

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ  
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ  
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

**С. В. Суліма, Л. С. Глоба**

# **WEB-ТЕХНОЛОГІЇ ІНТЕРНЕТ**

## **Підручник**

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського  
як підручник для здобувачів ступеня бакалавра  
за освітньою програмою «Інформаційно-комунікаційні технології»  
спеціальності G5 Електроніка, електронні комунікації, приладобудування та радіотехніка

Електронне мережеве навчальне видання

Київ  
КПІ ім. ІГОРЯ СІКОРСЬКОГО  
2026

УДК 004.738.5(075.8)

B19

Автори: Суліма Світлана Валеріївна, канд. техн. наук, доц.  
Глоба Лариса Сергіївна, д-р техн. наук, проф.  
Рецензенти: Єременко Олександра Сергіївна, д-р техн. наук, проф.  
Царьов Роман Юрійович, канд. техн. наук, доц.  
Відповідальний редактор Стрижак Олександр Євгенійович  
доктор технічних наук, професор, заступник директора з наукової роботи Національного центру "Мала академія наук України"

Гриф надано Вченою радою КПІ ім. Ігоря Сікорського  
(протокол № 12 від 15.12.2025 р.)  
за поданням вченої ради навчально-наукового інституту телекомунікаційних систем  
(протокол № 9 від 27.10.2025 р.)

**Суліма С. В.**

B19 Web-технології інтернет [Електронний ресурс] : підручник для здобувачів ступеня бакалавра за освіт. програмою «Інформаційно-комунікаційні технології» спец. G5 Електроніка, електронні комунікації, приладобудування та радіотехніка / С.В. Суліма, Л.С. Глоба; КПІ ім. Ігоря Сікорського. –Електрон. текст. дані (1 файл: 409 Кбайт). – Київ : КПІ ім. Ігоря Сікорського, 2026. – 780 с.

Підручник «Web-технології Інтернет» присвячений сучасним аспектам розвитку, функціонування та застосування вебтехнологій у контексті підготовки фахівців у галузі інформаційно-комунікаційних технологій. Видання охоплює широкий спектр питань — від історії становлення Інтернету та еволюції його поколінь (Web 1.0 – Web 4.0) до практичних засад веброзробки, зокрема створення та стилізації вебсторінок за допомогою HTML і CSS, програмування на JavaScript, а також використання інструментів для UX/UI-дизайну та оптимізації вебресурсів. Структура підручника побудована за модульним принципом, що забезпечує поступове засвоєння матеріалу: від базових понять до складних інтегрованих рішень. Значна увага приділена практичним прикладам, вправам і завданням, спрямованим на формування професійних компетентностей у сфері вебтехнологій та командної роботи в IT-проектах.

Видання призначене для здобувачів ступеня бакалавра за освітньою програмою «Інформаційно-комунікаційні технології» за спеціальністю G5 Електроніка, електронні комунікації, приладобудування та радіотехніка, проте може бути корисним також аспірантам, викладачам, фахівцям IT-сфери та всім, хто прагне набутися системних знань і практичних навичок у галузі розробки та експлуатації сучасних вебсистем.

УДК 004.738.5(075.8)

Реєстр. № НП 23/24-317. Обсяг 19,2 авт. арк.

Національний технічний університет України  
«Київський політехнічний інститут імені Ігоря Сікорського»  
проспект Берестейський, 37, м. Київ, 03056  
<https://kpi.ua>

Свідцтво про внесення до Державного реєстру видавців, виготовлювачів і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

© С.В. Суліма, Л.С. Глоба, 2026  
© КПІ ім. Ігоря Сікорського, 2026

## ЗМІСТ

|                                                                 |    |
|-----------------------------------------------------------------|----|
| ВСТУП .....                                                     | 11 |
| РОЗДІЛ 1. Історія розвитку програмних додатків в Інтернет ..... | 13 |
| 1.1 Етапи розвитку технологій Інтернет .....                    | 13 |
| 1.1.1 Web 1.0 - Мережа доставки контенту (1990-2000).....       | 13 |
| 1.1.2 Web 2.0 - Соціальна мережа (2000-2010).....               | 17 |
| 1.1.3 Web 3.0 - Інтернет як база даних (2010-2020) .....        | 23 |
| 1.1.4 Web 4.0 - Симбіотична взаємодія (2020+) .....             | 36 |
| 1.2 Стек протоколів Інтернету.....                              | 38 |
| 1.2.1 Основні принципи Інтернету .....                          | 38 |
| 1.2.2 Сімейство протоколів TCP/IP.....                          | 39 |
| 1.3 Переваги та недоліки TCP/IP .....                           | 41 |
| 1.3.1 Переваги .....                                            | 41 |
| 1.3.2 Недоліки .....                                            | 42 |
| 1.4 Майбутнє: IPv6 .....                                        | 43 |
| 1.5 Веб-сайт та Веб-додаток.....                                | 46 |
| 1.5.1 Веб-сайти .....                                           | 47 |
| 1.5.2 Веб-додатки.....                                          | 50 |
| 1.5.3 Технологічні відмінності.....                             | 55 |
| 1.5.4 Гібридні рішення у веб-розробці.....                      | 58 |
| 1.5.5 Тренди та майбутнє.....                                   | 61 |
| 1.5.6 Сервер: основа веб-простору .....                         | 63 |
| 1.5.7 Браузер.....                                              | 67 |
| 1.5.8 Статичний та динамічний контент .....                     | 70 |
| 1.6 Базові технології фронтенду та бекенду .....                | 71 |
| 1.6.1 Ключові технології та поняття.....                        | 71 |
| 1.6.2 HTML: ключова фронтенд-технологія.....                    | 79 |
| 1.6.3 CSS: стилізація .....                                     | 83 |
| 1.6.4 DOM: структура веб-сторінки .....                         | 88 |
| 1.6.5 JavaScript .....                                          | 89 |

|                             |                                                   |     |
|-----------------------------|---------------------------------------------------|-----|
| 1.6.6                       | Бекенд: програмування серверної частини .....     | 91  |
|                             | Висновки .....                                    | 104 |
|                             | Питання для самоперевірки .....                   | 105 |
| РОЗДІЛ 2. Основи HTML ..... |                                                   | 107 |
| 2.1                         | Базові поняття .....                              | 108 |
| 2.1.1                       | Одинарні теги HTML .....                          | 108 |
| 2.1.2                       | Парні теги HTML .....                             | 111 |
| 2.1.3                       | Теги для коментарів .....                         | 116 |
| 2.1.4                       | Структура HTML-документа .....                    | 120 |
| 2.1.5                       | Елемент <head> .....                              | 121 |
| 2.1.6                       | Елемент <body> .....                              | 122 |
| 2.1.7                       | Блокові елементи .....                            | 123 |
| 2.1.8                       | Рядкові елементи .....                            | 127 |
| 2.1.9                       | HTML текст .....                                  | 130 |
| 2.1.10                      | HTML посилання .....                              | 131 |
| 2.1.11                      | HTML зображення .....                             | 132 |
| 2.1.12                      | HTML списки .....                                 | 134 |
| 2.1.13                      | Практичні рекомендації та найкращі практики ..... | 136 |
| 2.2                         | Створення макета сторінки та верстка .....        | 137 |
| 2.2.1                       | Блокова верстка .....                             | 137 |
| 2.2.2                       | Вкладені плаваючі блоки .....                     | 140 |
| 2.2.3                       | Вирівнювання стовпців за висотою .....            | 144 |
| 2.2.4                       | Властивість display .....                         | 147 |
| 2.2.5                       | Створення навігаційної панелі .....               | 152 |
| 2.2.6                       | Вирівнювання плаваючих елементів .....            | 161 |
| 2.2.7                       | Створення найпростішого макета .....              | 167 |
| 2.2.8                       | Позиціювання .....                                | 180 |
| 2.3                         | Основи CSS .....                                  | 198 |
| 2.3.1                       | Додавання CSS до веб-сторінки .....               | 205 |
| 2.3.2                       | Селектори .....                                   | 236 |



|                                                                         |                                                        |     |
|-------------------------------------------------------------------------|--------------------------------------------------------|-----|
| 2.3.3                                                                   | Принцип каскадування і специфічність правила .....     | 242 |
| 2.3.4                                                                   | Практичні поради та кращі практики .....               | 245 |
| 2.4                                                                     | Трансформації, переходи та анімації.....               | 247 |
| 2.4.1                                                                   | CSS-трансформації.....                                 | 248 |
| 2.4.2                                                                   | CSS-переходи.....                                      | 253 |
| 2.4.3                                                                   | CSS-анімації.....                                      | 272 |
| 2.5                                                                     | Практичний приклад: Створення банера з анімацією ..... | 281 |
|                                                                         | Висновки .....                                         | 289 |
|                                                                         | Питання для самоперевірки .....                        | 290 |
| РОЗДІЛ 3. ВЕБПРОГРАМУВАННЯ НА JAVASCRIPT: МОВА,<br>ДОКУМЕНТ, ПОДІЇ..... |                                                        | 291 |
| 3.1                                                                     | Мова програмування JavaScript.....                     | 291 |
| 3.1.1                                                                   | Вступ до JavaScript.....                               | 291 |
| 3.1.2                                                                   | Інструменти розробника.....                            | 292 |
| 3.1.3                                                                   | Тег "script" .....                                     | 293 |
| 3.1.4                                                                   | Зовнішні скрипти .....                                 | 295 |
| 3.1.5                                                                   | Структура коду .....                                   | 297 |
| 3.1.6                                                                   | Змінні.....                                            | 309 |
| 3.1.7                                                                   | Типи даних .....                                       | 311 |
| 3.1.8                                                                   | Взаємодія: alert, prompt, confirm .....                | 316 |
| 3.1.9                                                                   | Математика .....                                       | 319 |
| 3.1.10                                                                  | Функції.....                                           | 323 |
| 3.1.11                                                                  | Функціональні вирази .....                             | 326 |
| 3.1.12                                                                  | Стрілкові функції в JavaScript .....                   | 334 |
| 3.1.13                                                                  | Об'єкти .....                                          | 343 |
| 3.2                                                                     | Браузер: документ .....                                | 348 |
| 3.2.1                                                                   | Браузерне середовище та специфікації.....              | 349 |
| 3.2.2                                                                   | Document Object Model (DOM).....                       | 357 |
| 3.2.3                                                                   | Browser Object Model (BOM) .....                       | 369 |
| 3.2.4                                                                   | DOM дерево .....                                       | 388 |

|        |                                                             |     |
|--------|-------------------------------------------------------------|-----|
| 3.2.5  | Навігація по DOM.....                                       | 392 |
| 3.2.6  | Пошук елементів: getElement* та querySelector*.....         | 397 |
| 3.2.7  | Властивості вузлів: тип, тег та вміст.....                  | 403 |
| 3.2.8  | Атрибути та властивості.....                                | 419 |
| 3.2.9  | Внесення змін в документ.....                               | 448 |
| 3.2.10 | Стили та класи.....                                         | 451 |
| 3.2.11 | Координати елементів у DOM.....                             | 455 |
| 3.3    | Браузер: події.....                                         | 468 |
| 3.3.1  | Вступ до подій браузера.....                                | 468 |
| 3.3.2  | Використання атрибута HTML.....                             | 469 |
| 3.3.3  | Використання властивостей DOM-об'єкта.....                  | 470 |
| 3.3.4  | Поширені помилки.....                                       | 471 |
| 3.3.5  | addEventListener.....                                       | 473 |
| 3.3.6  | Об'єкт події.....                                           | 478 |
| 3.3.7  | Об'єкт-обробник: handleEvent.....                           | 485 |
| 3.4    | Форми та інтерактивні елементи.....                         | 498 |
| 3.4.1  | Властивості та методи форми.....                            | 499 |
| 3.4.2  | Навігація: форми та елементи.....                           | 505 |
| 3.4.3  | Зворотне посилання: element.form.....                       | 509 |
| 3.4.4  | Елементи форми.....                                         | 513 |
| 3.4.5  | Події форм - реагування на дії користувача в реальному часі | 517 |
| 3.4.6  | Форми: подія та метод submit.....                           | 526 |
|        | Висновки.....                                               | 535 |
|        | Питання для самоперевірки.....                              | 536 |
|        | РОЗДІЛ 4. Графічний дизайн і UX-проектування вебсайтів..... | 538 |
| 4.1    | Графічний дизайн сайту.....                                 | 538 |
| 4.1.1  | Генерація ідей дизайну.....                                 | 539 |
| 4.1.2  | Розроблення макету головної сторінки сайту.....             | 543 |
| 4.1.3  | Фіксований та еластичний макет.....                         | 546 |

|        |                                                          |     |
|--------|----------------------------------------------------------|-----|
| 4.1.4  | Дизайн під мобільні пристрої .....                       | 549 |
| 4.1.5  | Модульна сітка.....                                      | 554 |
| 4.1.6  | Навігаційне меню .....                                   | 555 |
| 4.1.7  | Типографіка .....                                        | 559 |
| 4.1.8  | Порядок у вихідному макеті дизайну .....                 | 561 |
| 4.2    | Принципи композиції сайтів .....                         | 562 |
| 4.2.1  | Принципи веб дизайну .....                               | 563 |
| 4.2.2  | Веб дизайн з чого почати.....                            | 573 |
| 4.2.3  | Теорія кольору та колірні схеми .....                    | 577 |
| 4.2.4  | Як використовувати поєднання кольорів у дизайні сайту .. | 589 |
| 4.2.5  | Основи композиції сайтів.....                            | 592 |
| 4.2.6  | Принципи гештальту.....                                  | 600 |
| 4.2.7  | Практичне застосування принципів композиції .....        | 612 |
| 4.3    | Figma - інструмент сучасного веб-дизайну .....           | 625 |
| 4.3.1  | Навіщо потрібна Фігма.....                               | 625 |
| 4.3.2  | Які проекти можна створювати у Фігмі.....                | 626 |
| 4.3.3  | Що ще можна робити у Фігмі .....                         | 629 |
| 4.3.4  | Спільна робота у Фігмі .....                             | 632 |
| 4.3.5  | Початок роботи у Фігмі .....                             | 633 |
| 4.3.6  | Інструменти та можливості Фігми .....                    | 654 |
| 4.3.7  | Модульна сітка.....                                      | 660 |
| 4.3.8  | Векторні форми .....                                     | 664 |
| 4.3.9  | Криві .....                                              | 669 |
| 4.3.10 | Зображення.....                                          | 671 |
| 4.3.11 | Ефекти та маски .....                                    | 680 |
| 4.3.12 | Текст.....                                               | 685 |
| 4.3.13 | Додаткові можливості .....                               | 700 |
|        | Висновки .....                                           | 703 |
|        | Питання для самоперевірки .....                          | 704 |
|        | РОЗДІЛ 5. Оптимізація сайту .....                        | 706 |

|       |                                                                                   |     |
|-------|-----------------------------------------------------------------------------------|-----|
| 5.1   | Види оптимізації .....                                                            | 706 |
| 5.1.1 | Технічна оптимізація .....                                                        | 706 |
| 5.1.2 | SEO-оптимізація (пошукова оптимізація) .....                                      | 712 |
| 5.1.3 | UX/UI оптимізація: комплексний підхід до покращення користувачького досвіду ..... | 718 |
| 5.2   | Аудит сайту і внутрішня оптимізація.....                                          | 722 |
| 5.2.1 | Технічний аудит.....                                                              | 722 |
| 5.2.2 | Оптимізація контенту .....                                                        | 728 |
| 5.2.3 | Внутрішня перелінковка.....                                                       | 734 |
| 5.3   | Зовнішня оптимізація .....                                                        | 741 |
| 5.3.1 | Link Building (побудова посилань) .....                                           | 741 |
| 5.3.2 | Соціальні сигнали .....                                                           | 748 |
| 5.3.3 | Контент-маркетинг.....                                                            | 757 |
| 5.3.4 | Локальна SEO (для місцевого бізнесу) .....                                        | 770 |
|       | Висновки .....                                                                    | 775 |
|       | Питання для самоперевірки .....                                                   | 776 |
|       | Список рекомендованої літератури .....                                            | 777 |

## ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

|       |                                                                                 |
|-------|---------------------------------------------------------------------------------|
| API   | Application Programming Interface — програмний інтерфейс застосунків            |
| AR    | Augmented Reality — доповнена реальність                                        |
| ARIA  | Accessible Rich Internet Applications — технологія для доступності вебдодатків  |
| CMS   | Content Management System — система керування контентом                         |
| CSS   | Cascading Style Sheets — каскадні таблиці стилів                                |
| DHCP  | Dynamic Host Configuration Protocol — протокол динамічної конфігурації хостів   |
| DOM   | Document Object Model — об'єктна модель документа                               |
| HTML  | HyperText Markup Language — мова розмітки гіпертексту                           |
| HTTP  | HyperText Transfer Protocol — протокол передавання гіпертексту                  |
| HTTPS | HyperText Transfer Protocol Secure — захищений протокол передавання гіпертексту |
| IoT   | Internet of Things — інтернет речей                                             |
| IP    | Internet Protocol — інтернет-протокол                                           |
| IPv4  | Internet Protocol version 4 — інтернет-протокол версії 4                        |
| IPv6  | Internet Protocol version 6 — інтернет-протокол версії 6                        |
| IPsec | Internet Protocol Security — набір протоколів для захисту IP-трафіку            |
| JSON  | JavaScript Object Notation — текстовий формат обміну даними                     |
| QoS   | Quality of Service — якість обслуговування                                      |
| SaaS  | Software as a Service — програмне забезпечення як послуга                       |
| SEO   | Search Engine Optimization — пошукова оптимізація                               |
| SLAAC | Stateless Address Autoconfiguration — безстанова автоматична конфігурація адрес |

|      |                                                              |
|------|--------------------------------------------------------------|
| SQL  | Structured Query Language — мова структурованих запитів      |
| UI   | User Interface — користувацький інтерфейс                    |
| URL  | Uniform Resource Locator — уніфікований локатор ресурсу      |
| UX   | User Experience — користувацький досвід                      |
| VR   | Virtual Reality — віртуальна реальність                      |
| WSGI | Web Server Gateway Interface — шлюзовий інтерфейс вебсервера |
| XML  | eXtensible Markup Language — розширювана мова розмітки       |

## ВСТУП

Стрімкий розвиток інформаційно-комунікаційних технологій, що відбувається у XXI столітті, зумовлює зростання ролі Інтернету як базової інфраструктури сучасного суспільства. Від початкових статичних інформаційних ресурсів він трансформувався у глобальне інтерактивне середовище, яке забезпечує не лише доступ до знань, а й підтримку складних соціально-економічних процесів, наукових досліджень та освітньої діяльності.

В умовах цифровізації усіх сфер життєдіяльності набуває особливої актуальності формування у майбутніх фахівців системного уявлення про принципи побудови та функціонування Інтернету, сучасні вебтехнології, а також інструменти розроблення та супроводу вебсистем. Це зумовлює необхідність створення навчальних видань, які поєднують фундаментальні теоретичні засади з практичними аспектами застосування.

Метою даного підручника є забезпечення здобувачів вищої освіти комплексними знаннями щодо історії становлення та розвитку Інтернету, базових принципів його архітектури та протоколів, мов і засобів веброзробки, а також сучасних підходів до дизайну, інтерактивності, безпеки й оптимізації вебресурсів.

Структура видання побудована за модульним принципом, що дозволяє поетапно опановувати матеріал – від теоретичних положень і концептуальних моделей до прикладних рішень та практичних завдань. Особлива увага приділяється формуванню компетентностей, необхідних для здійснення повного циклу створення вебпродуктів – від розроблення концепції до тестування та впровадження.

Підручник призначений для здобувачів ступеня бакалавра за освітньою програмою «Інформаційно-комунікаційні технології», проте він може бути корисним також аспірантам, викладачам, фахівцям у галузі інформаційних технологій та усім, хто прагне науково обґрунтованого й системного розуміння сучасних Інтернет-технологій.





## РОЗДІЛ 1. ІСТОРІЯ РОЗВИТКУ ПРОГРАМНИХ ДОДАТКІВ В ІНТЕРНЕТ

### 1.1 Етапи розвитку технологій Інтернет

#### 1.1.1 Web 1.0 - Мережа доставки контенту (1990-2000)

Що таке "мережа доставки контенту"?

Уявіть Web 1.0 як величезну цифрову бібліотеку або телебачення через інтернет. Сайти працювали як односторонні канали передачі інформації - власник сайту публікував контент, а користувачі його тільки читали. Ніякої взаємодії.

#### Технічні особливості:

##### 1. Статичні сторінки

- Що це означає: Кожна веб-сторінка була як друкована сторінка в книзі - її зміст був фіксованим.
- Приклад: Якщо ви зайшли на сайт компанії, то побачите ту саму інформацію, що й інший користувач.
- Контраст з сьогоденням: Сучасні сайти показують різний контент залежно від того, хто ви, звідки прийшли, що шукаєте.

##### 2. Серверна обробка з файлової системи

Як це працювало:

- На сервері лежали готові HTML-файли.
- Коли користувач запитував сторінку, сервер просто відправляв файл "як є".
- Ніякої обробки або генерації контенту не відбувалося.

Аналогія: Як бібліотекар, який просто дає вам готову книгу з полиці.

##### 3. Базові технології

Server Side Includes (SSI):

```
html
```

```
<!--#include file="header.html"-->
<h1>Головна сторінка</h1>
<!--#include file="footer.html"-->
```

Дозволяв вставляти один файл в інший. Єдиний спосіб уникнути дублювання коду.

#### Common Gateway Interface (CGI):

```
perl
#!/usr/bin/perl
print "Content-type: text/html\n\n";
print "<h1>Привіт з CGI!</h1>";
```

Перший спосіб створення динамічного контенту. Дуже повільний і ресурсозатратний.

#### 4. Примітивний дизайн

##### Фреймсети (frames):

```
html
<frameset rows="100,*,50">
  <frame src="header.html">
  <frame src="content.html">
  <frame src="footer.html">
</frameset>
```

##### Таблиці для макетування:

```
html
<table width="100%">
  <tr>
    <td width="200">Меню</td>
    <td>Основний контент</td>
  </tr>
</table>
```

#### **Модель взаємодії: "Один до багатьох"**

##### Лінійна комунікація:

[Власник сайту] → [Веб-сайт] → [Користувач 1]  
 → [Користувач 2]  
 → [Користувач 3]

Конкретні приклади того часу:

#### 1. Корпоративні сайти:

- сторінка з інформацією про компанію;
- список товарів без можливості замовлення;
- контактні дані;
- жодної інтерактивності.

#### 2. Особисті домашні сторінки:

```
html
<html>
<body bgcolor="yellow">
<h1><blink>Ласкаво просимо на мою
сторінку!</blink></h1>
 Сайт в розробці 
<p>Мене звати Іван, мені 25 років...</p>
</body>
</html>
```

#### 3. Новинні сайти:

- статті публікувалися вручну;
- ніяких коментарів або обговорень;
- оновлення раз на день або рідше;
- обмеження зворотного зв'язку.

Єдині способи комунікації:

1. Email - <mailto:webmaster@site.com>
2. Телефон - номер на сторінці контактів

### 3. Гостьові книги (primitive гостевые книги) - найпростіша форма CGI

Приклад гостьової книги:

```
Петро | 15.03.1998 | Дуже цікавий сайт!
Марія | 12.03.1998 | Дякую за інформацію
```

**"Джерело диктує - читач виконує"**

Характеристики цієї моделі:

- авторитарність: власник сайту має абсолютну владу над контентом;
- пасивність: користувачі не можуть впливати на контент;
- однобічність: інформація тече тільки в одному напрямку;
- статичність: контент змінюється рідко, тільки власником;
- типові сайти Web 1.0:
  1. Yahoo! Directory - каталог сайтів, організований по категоріях.
  2. Ранній Amazon - просто каталог книг.
  3. CNN.com - новини без коментарів.
  4. Особисті домашні сторінки на GeoCities.

Попри всі обмеження, Web 1.0 був величезним проривом:

- глобальна доступність інформації;
- миттєве оновлення (порівняно з друкованими виданнями);
- низька вартість публікації;
- відсутність географічних обмежень.

Технічні обмеження того часу:

- повільний інтернет: dial-up модеми 56k;
- обмежена пам'ять та обчислювальна потужність;
- відсутність стандартів: браузерні війни між Netscape і Internet Explorer;
- примітивні інструменти розробки: текстові редактори, HTML вручну.

Web 1.0 був як цифрова версія друкованих видань - статична, одностороння, але революційна для свого часу. Він створив фундамент для всього, що прийшло потім.

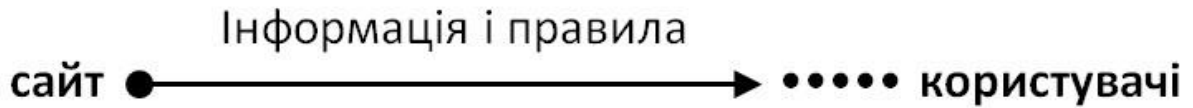


Рисунок 1.1 — Web 1.0

### 1.1.2 Web 2.0 - Соціальна мережа (2000-2010)

Web 2.0 - це справжня революція, яка змінила інтернет з односторонньої бібліотеки на живу, інтерактивну соціальну екосистему.

Що означає "цінувати контент користувача"? Це фундаментальна зміна філософії інтернету:

- Web 1.0: "Ми створюємо - ви споживаєте",
- Web 2.0: "Ви створюєте - ми надаємо платформу"

Приклад еволюції:

- Енциклопедія Britannica (Web 1.0) → Wikipedia (Web 2.0)
- Експерти пишуть статті → Будь-хто може створювати та редагувати

#### Ключові особливості в деталях

1. Користувацький контент - основа екосистеми

Програми стають платформами:

- Web 1.0: [Компанія] → [Контент] → [Користувач]
- Web 2.0: [Платформа] ↔ [Користувач створює контент] ↔ [Інші користувачі]

Конкретні приклади:

- YouTube (2005): "Broadcast Yourself" - кожен може стати телеканалом
- Flickr (2004): фотообмін замість особистих фотоальбомів
- MySpace/Facebook: профілі замість статичних домашніх сторінок

## 2. Активний розвиток нових форматів

### Подкасти (2004):

```
xml
<rss version="2.0">
  <channel>
    <title>Мій Подкаст</title>
    <item>
      <title>Епізод 1</title>
      <enclosure url="episode1.mp3" type="audio/mpeg"/>
    </item>
  </channel>
</rss>
```

- Революція: будь-хто може створити радіошоу
- Технологія: RSS + аудіофайли = автоматичне завантаження
- Приклад: Serial, This American Life стають глобальними

### Блоги (Blogger 1999, WordPress 2003):

- До блогів: потрібно знати HTML, FTP, мати сервер,
- З блогами: просто пиши та публікуй.

Вплив: народження блогосфери, цитування, трекбеки.

```
html
<!-- Простота публікації -->
<form action="/publish" method="post">
  <textarea name="content">Що у вас на
думці?</textarea>
  <button>Опублікувати</button>
</form>
```

### RSS-кураторство:

- Проблема Web 1.0: потрібно відвідувати кожен сайт окремо;
- Рішення Web 2.0: агрегація контенту в одному місці;

- Інструменти: Google Reader, Bloglines.

## Технічні інновації Web 2.0

### 1. Безкоштовний доступ до інформації

Нові бізнес-моделі:

- Freemium: базові функції безкоштовно, додаткові - за плату;
- Рекламна модель: користувачі = продукт, рекламодавці = клієнти;
- Вірусний розповсюдження: користувачі самі залучають інших.

Приклади:

- Gmail (2004): 1GB безкоштовно (коли конкуренти давали 2MB);
- Skype (2003): безкоштовні дзвінки через інтернет;
- Wikipedia (2001): безкоштовна енциклопедія.

### 2. Колективна класифікація (Folksonomy)

Теги замість ієрархічних каталогів:

- Web 1.0 (Yahoo Directory):  
Комп'ютери → Програмне забезпечення → Графіка →  
Фоторедактори
- Web 2.0 (Delicious, Flickr):  
#фото #редагування #креатив #дизайн #безкоштовно

Переваги тегування:

- Гнучкість: один ресурс може мати багато тегів
- Демократичність: користувачі самі вирішують, як класифікувати
- Еволюційність: популярні теги стають стандартом

Конкретний приклад - Flickr: фото заходу сонця може мати теги:

```
#sunset #nature #photography #canon #vacation #beach  
#romantic
```

### 3. Динамічний контент та AJAX

Технологічний прорив - AJAX (2005):

```
javascript  
// Оновлення без перезавантаження сторінки
```

```
function loadComments() {
  const xhr = new XMLHttpRequest();
  xhr.open('GET', '/comments');
  xhr.onload = function() {
    document.getElementById('comments').innerHTML =
xhr.responseText;
  };
  xhr.send();
}
```

Револьюційні зміни:

- Google Maps (2005): перетягування карти без перезавантаження
- Gmail: інтерфейс як в настільному додатку
- Facebook: миттєві оновлення стрічки новин

#### 4. API Revolution

Відкриті API створили екосистему:

Google Maps API (2005):

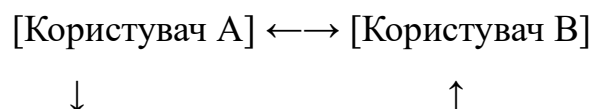
```
javascript
const map = new
google.maps.Map(document.getElementById('map'), {
  zoom: 13,
  center: { lat: 50.4501, lng: 30.5234 } // Київ
});
```

Приклади mashup-сервісів:

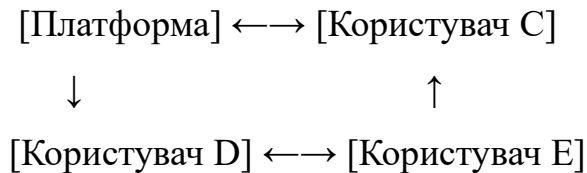
- HousingMaps.com: Craigslist оголошення на Google Maps
- Флікр + карти: фотографії прив'язані до місць
- Twitter API: тисячі додатків для роботи з твітами

#### Модель взаємодії: "Багато до багатьох"

Двовимірна взаємодія:







Конкретні приклади платформ:

1. MySpace (2003-2008):

- Перша масова соціальна мережа
- Користувачі могли кастомізувати профілі HTML/CSS
- Музиканти використовували для просування

2. Facebook (2004):

- Спочатку тільки для студентів
- Революційна стрічка новин (2006)
- API для сторонніх додатків (2007)

3. YouTube (2005):

- "Everyone is a broadcaster"
- Коментарі, лайки, підписки
- Монетизація для авторів

### Еволюція ролей

Власники платформ стають "модераторами":

- Встановлюють правила: що дозволено публікувати
- Надають інструменти: редактори, системи коментарів
- Забезпечують інфраструктуру: сервери, CDN, безпека

Користувачі стають "співтворцями":

- Створюють контент: пости, відео, фото, коментарі
- Кураторство: лайки, репости, рекомендації
- Модерація: повідомлення про порушення, community guidelines

### Технічні інновації в деталях

Rich Internet Applications (RIA):

Flash/Flex Applications:

- Інтерактивні додатки в браузері

- FarmVille, Angry Birds в браузері

Silverlight від Microsoft:

- Конкурент Flash для медіа-контенту
- Netflix використовував до переходу на HTML5

Web Services та SOA:

```
xml
<!-- SOAP веб-сервіс -->
<soap:Envelope>
  <soap:Body>
    <getUserInfo>
      <userId>12345</userId>
    </getUserInfo>
  </soap:Body>
</soap:Envelope>
```

### **Соціальні інновації**

#### **1. Виникнення нових професій:**

- Blogger - професійний блогер
- Community Manager - менеджер спільноти
- SEO Specialist - оптимізація для пошукових систем
- Social Media Manager - управління соціальними мережами

#### **2. Нові форми комунікації:**

- Hashtags - #тег для групування контенту
- @mentions - звертання до користувачів
- Likes/Hearts - швидка реакція на контент
- Sharing/Retweets - поширення контенту

### **Негативні наслідки Web 2.0**

Нові загрози:

#### **1. Інформаційні:**

- Filter Bubbles - ехо-камери думок

- Fake News - швидке поширення дезінформації
- Information Overload - надлишок інформації

## 2. Соціальні:

- Cyberbullying - онлайн цькування
- Privacy Issues - втрата приватності
- Addiction - залежність від соціальних мереж

## 3. Економічні:

- Digital Divide - розрив між тими, хто має доступ і не має
- Platform Monopolies - домінування великих платформ

### **Ключові компанії та сервіси Web 2.0:**

Соціальні мережі: MySpace (2003), Facebook (2004), LinkedIn (2003).

Медіа-платформи: YouTube (2005), Flickr (2004), Vimeo (2004).

Блогінг: Blogger (1999), WordPress (2003), LiveJournal (1999).

Мікроблогінг: Twitter (2006) - революція 140 символів.

Закладки: del.icio.us (2003), StumbleUpon (2001).

Колаборація: Wikipedia (2001), Google Docs (2006).

Web 2.0 перетворив інтернет з бібліотеки в живу соціальну екосистему, де кожен користувач одночасно є і споживачем, і творцем контенту. Це заклало основу для сучасного соціального Інтернету.



Рисунок 1.2 — Web 2.0

### **1.1.3 Web 3.0 - Інтернет як база даних (2010-2020)**

Web 3.0 - це справжня технологічна революція, яка перетворила інтернет з соціальної платформи на розумну, самонавчальну систему. "Інтернет як база даних" - що це означає?

Це кардинальна зміна сприйняття інтернету:

- Web 1.0: Інтернет = колекція документів

- Web 2.0: Інтернет = соціальна платформа
- Web 3.0: Інтернет = розумна база даних, яка "розуміє" дані

Концептуальна різниця:

Web 2.0: "iPhone 15" → просто текст

```
Web 3.0: "iPhone 15" → {
  тип: "смартфон",
  виробник: "Apple",
  ціна: "$999",
  дата_випуску: "2023-09-22",
  операційна_система: "iOS 17"
}
```

### Ключові особливості в деталях

#### 1. Семантика сайтів - від тексту до розуміння

Семантична розмітка (Schema.org):

```
html
<!-- Web 2.0 -->
<div>
  Ресторан "Борщ і Сало"
  Рейтинг: 4.5 зірки
  Київ, вул. Хрещатик, 1
</div>
```

```
<!-- Web 3.0 -->
<div itemscope itemtype="http://schema.org/Restaurant">
  <h1 itemprop="name">Ресторан "Борщ і Сало"</h1>
  <div itemprop="aggregateRating" itemscope
itemtype="http://schema.org/AggregateRating">
    <span itemprop="ratingValue">4.5</span> з
    <span itemprop="bestRating">5</span> зірок
  </div>
```

```

<div itemprop="address" itemscope
itemtype="http://schema.org/PostalAddress">
  <span itemprop="streetAddress">вул. Хрещатик,
1</span>
  <span itemprop="addressLocality">Київ</span>
</div>
</div>

```

Результат для пошукових систем:

- Google може показати рейтинг безпосередньо в результатах пошуку
- Карти автоматично розпізнають адресу
- Голосові помічники можуть відповісти на питання про режим роботи

## 2. "Розумний пошук" - еволюція Google

Еволюція пошукових запитів:

- Web 1.0 (1998): "pizza Kyiv restaurant"
- Результат: сторінки зі словами pizza, Kyiv, restaurant
- Web 2.0 (2005): "найкраща піца в Києві"
- Результат: врахування соціальних сигналів, відгуків
- Web 3.0 (2015): "де можна поїсти піцу поблизу зараз?"
- Результат: розуміння контексту, місцезнаходження, часу

Технології, що це забезпечили:

Knowledge Graph (Google, 2012):

Пошук: "Тарас Шевченко"

Результат не просто посилання, а факти:

- Народився: 9 березня 1814, Моринці
- Професія: поет, художник
- Відомі твори: "Кобзар", "Заповіт"
- Зв'язки: українська література, романтизм

Entity Recognition:

```

javascript
// Розпізнавання сутностей

```

```
const text = "Тарас Шевченко народився в Морінцях";
const entities = {
  "Тарас Шевченко": { type: "PERSON", category: "poet"
},
  "Морінці": { type: "LOCATION", category: "village" }
};
```

### 3. Штучний інтелект та NLP

Natural Language Processing в дії:

Google Translate еволюція:

- 2006 (Web 2.0): Statistical Machine Translation

"Я люблю програмування" → "I love programming" (60% точність)

- 2016 (Web 3.0): Neural Machine Translation

"Я займаюся розробкою мобільних додатків" →

"I develop mobile applications" (95% точність)

Голосові помічники:

Siri (2011) - перші кроки:

- Користувач: "Встанови будильник на 7 ранку"
- Siri: Розпізнає команду → Встановлює будильник

Google Assistant (2016) - контекстне розуміння:

- Користувач: "Яка погода в Києві?"
- Assistant: "В Києві сьогодні +15°C, хмарно"
- Користувач: "А завтра?"
- Assistant: Розуміє, що "завтра" стосується Києва, не питає знову місто

### 4. 3D графіка та іммерсивний досвід

WebGL та 3D в браузері:

```
javascript
// Three.js приклад
const scene = new THREE.Scene();
```

```

const camera = new THREE.PerspectiveCamera(75,
window.innerWidth/window.innerHeight, 0.1, 1000);
const renderer = new THREE.WebGLRenderer();

// Створення 3D об'єкта
const geometry = new THREE.BoxGeometry();
const material = new THREE.MeshBasicMaterial({color:
0x00ff00});
const cube = new THREE.Mesh(geometry, material);
scene.add(cube);

```

#### Практичні застосування:

##### E-commerce:

- IKEA Place (AR додаток): віртуальне розміщення меблів у кімнаті
- Nike: 3D кастомізація взуття
- Amazon: 360° огляд товарів

##### Ігри в браузері:

- Angry Birds переходить з Flash на WebGL
- Unity WebGL - повноцінні 3D ігри в браузері

##### Освіта:

- Google Earth: 3D подорожі світом
- Virtual museum tours: Лувр, Ермітаж онлайн

#### 5. Взаємозв'язок контенту через метадані

##### Linked Data принцип:

```

turtle
# RDF тройці (Resource Description Framework)
<http://example.com/person/john>
  <http://xmlns.com/foaf/0.1/name> "Джон Сміт" ;
  <http://xmlns.com/foaf/0.1/knows>
<http://example.com/person/jane> ;

```

```
<http://xmlns.com/foaf/0.1/workplaceHomepage>
<http://example.com/company/tech> .
```

Практичні результати:

- Facebook Graph API: зв'язки між людьми, сторінками, подіями
- Google Knowledge Graph: автоматичне з'єднання фактів про персони, місця, речі
- DBpedia: структуровані дані з Wikipedia

Приклад взаємозв'язку:

Стаття про iPhone → зв'язана з:

- └─ Apple Inc. (виробник)
- └─ iOS (операційна система)
- └─ Steve Jobs (засновник)
- └─ Смартфони (категорія)
- └─ Конкуренти (Samsung Galaxy, Google Pixel)

## 6. Універсальна доступність - Internet of Things

Пристрої, що підключилися до інтернету:

- 2010: ~12.5 млрд пристроїв
- 2020: ~50 млрд пристроїв

Категорії IoT пристроїв:

Розумний дім:

```
javascript
// Приклад керування розумним будинком
const homeAPI = {
  thermostat: { temperature: 22, mode: "auto" },
  lights: { living_room: "on", bedroom: "dim" },
  security: { cameras: "active", alarm: "off" }
};

// Голосова команда: "Зроби темніше в спальні"
processVoiceCommand("dim bedroom lights")
```



```
→ homeAPI.lights.bedroom = "dim";
```

Носимі пристрої:

- Apple Watch (2015): фітнес-трекінг, сповіщення
- Google Glass: доповнена реальність (хоча не став масовим)

Автомобільна індустрія:

- Tesla: автомобілі як комп'ютери на колесах
- Android Auto/Apple CarPlay: інтеграція смартфона і авто

## Три основні "С" Web 3.0

### 1. Самофінансування

Нові моделі монетизації:

Freemium до Premium:

Spotify:

- Безкоштовно: реклама, обмеження
- Premium: \$9.99/місяць, без реклами, офлайн режим

Subscription Economy:

- Netflix: від DVD-по-пошті до стрімінгового гіганта
- SaaS: Software as a Service (Salesforce, Office 365)

Creator Economy:

YouTube Partner Program (2007):

Автор → Створює контент → YouTube показує рекламу →

Прибуток ділиться 55% автору, 45% YouTube

### 2. Самоорганізація

Алгоритмічна організація контенту:

Facebook News Feed (2006-2016 еволюція):

2006: Хронологічна стрічка

2009: EdgeRank алгоритм - враховує взаємодію

2013: Machine Learning - персоналізація

2016: AI-driven curation - глибоке навчання

YouTube Recommendations:

```
python
# Спрощений алгоритм рекомендацій
def recommend_videos(user_history, video_metadata):
    similar_users = find_similar_users(user_history)
    popular_among_similar =
get_popular_videos(similar_users)

# Враховуємо:
# - Час перегляду (Watch Time)
# - CTR (Click Through Rate)
# - Лайки/дизлайки
# - Коментарі

return ranked_recommendations
```

### 3. Самокерованість

Автоматизація та автономність:

Google Search автоматичне оновлення:

- Panda (2011): боротьба з низькоякісним контентом
- Penguin (2012): боротьба з спамними посиланнями
- Hummingbird (2013): розуміння семантики запитів
- RankBrain (2015): машинне навчання в пошуку

Автоматична модерація:

```
python
# Приклад автомодерації YouTube
def moderate_content(video):
    # Аудіо аналіз
    audio_flags = detect_copyrighted_music(video.audio)

    # Відео аналіз
```

```

    visual_flags =
detect_inappropriate_content(video.frames)

    # Текст аналіз
    text_flags =
analyze_title_description(video.metadata)

    if any([audio_flags, visual_flags, text_flags]):
        return "FLAGGED_FOR_REVIEW"
    return "APPROVED"

```

### Додаткові ознаки Web 3.0

#### 1. Мобільність

Mobile-First дизайн:

Responsive Web Design (2010):

```

css
/* Адаптивний дизайн */
@media (max-width: 768px) {
    .container {
        width: 100%;
        padding: 10px;
    }
}

@media (min-width: 1200px) {
    .container {
        width: 1140px;
        margin: 0 auto;
    }
}

```

Progressive Web Apps (PWA):

```

javascript
// Service Worker для офлайн функціональності
self.addEventListener('fetch', event => {
  if (event.request.url.includes('/api/')) {
    event.respondWith(
      caches.match(event.request)
        .then(response => response ||
fetch(event.request))
    );
  }
});

```

Мобільні статистики:

- 2010: 4% трафіку з мобільних пристроїв
- 2016: 51% трафіку з мобільних (переломний момент)
- 2020: 54.8% трафіку з мобільних

## 2. Масштабованість

Cloud Computing революція:

AWS еволюція:

2006: EC2 (Elastic Compute Cloud) - віртуальні сервери

2012: Over 1 trillion objects in S3

2020: Майже \$50 млрд річного доходу

Microservices Architecture:

- Монолітна архітектура (Web 2.0):

[Одна велика програма]

- Мікросервісна архітектура (Web 3.0):

[Auth Service]  $\longleftrightarrow$  [User Service]  $\longleftrightarrow$  [Payment Service]

↑

↑

↑

[API Gateway]  $\longleftrightarrow$  [Content Service]  $\longleftrightarrow$  [Analytics Service]

Auto-scaling:

```

yaml

```

```
# Kubernetes автомасштабування
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: web-app-hpa
spec:
  minReplicas: 2
  maxReplicas: 10
  targetCPUUtilizationPercentage: 70
```

### 3. Ігровікованість

Гейміфікація веб-сервісів:

Foursquare/Swarm:

- Check-ins: відзначення відвідування місць
- Badges: нагороди за активність
- Mayor: статус за найчастіші відвідування
- Leaderboard: рейтинг серед друзів

Duolingo:

```
javascript
// Система нагород
const user_progress = {
  streak_days: 45,           // Серія днів навчання
  xp_points: 12500,          // Очки досвіду
  level: 15,                 // Рівень
  badges: ['early_bird', 'perfectionist', 'scholar'],
  league: 'diamond'          // Ліга змагань
};
```

LinkedIn гейміфікація:

- Profile completeness: прогрес-бар заповнення профілю
- Skill endorsements: підтвердження навичок від колег
- Connection milestones: досягнення в кількості зв'язків

## Технологічні досягнення Web 3.0

JavaScript еволюція:

ES6+ (2015+) нові можливості:

```
javascript
// Стрілочні функції
const fetchData = async (url) => {
  const response = await fetch(url);
  return response.json();
};

// Деструктурування
const {name, age, city} = user;

// Template literals
const message = `Привіт, ${name}! Тобі ${age} років.`;
```

CSS3 потужності:

```
css
/* 3D трансформації */
.card {
  transform: perspective(1000px) rotateY(45deg);
  transition: transform 0.3s ease;
}

/* Flexbox для макетування */
.container {
  display: flex;
  justify-content: space-between;
  align-items: center;
}
```

```
/* CSS Grid */
.grid {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(300px,
1fr));
  gap: 20px;
}
```

### Соціальні зміни Web 3.0

Нові професії:

- Data Scientist: аналіз великих даних
- UX/UI Designer: досвід користувача
- DevOps Engineer: розгортання та підтримка
- Machine Learning Engineer: розробка ШІ систем

Зміна поведінки користувачів:

- Attention Economy: боротьба за увагу користувачів
- FOMO (Fear of Missing Out): страх пропустити щось важливе
- Echo Chambers: алгоритми створюють інформаційні бульбашки

### Виклики Web 3.0:

1. Privacy Paradox: користувачі хочуть персоналізацію, але турбуються про приватність.
2. Algorithm Bias: упередження в алгоритмах ШІ.
3. Information Overload: надлишок персоналізованого контенту.
4. Digital Addiction: залежність від персоналізованих стрічок.

Web 3.0 перетворив інтернет на розумну, самонавчальну систему, яка розуміє контекст, передбачає потреби користувачів та автоматично адаптується до змін. Це створило основу для сучасного ШІ-керованого Інтернету.



Рисунок 1.3 — Web 3.0

### 1.1.4 Web 4.0 - Симбіотична взаємодія (2020+)

Це етап розвитку Інтернету, коли користувач і машина працюють як єдине ціле. Машини не просто виконують команди, а розуміють контекст, наміри й навіть емоції людини.

#### 1. Симбіотична взаємодія людини та машини

Машини не лише обробляють дані, а й прогнозують потреби користувача.

Приклад: розумні окуляри, які автоматично перекладають написані або усні тексти, підказують маршрут, нагадують про завдання.

#### 2. Розмиті межі між людиною та машиною

Людина і технологія зливаються — завдяки нейроінтерфейсам, AR/VR, імплантам.

Приклад: мозково-комп'ютерні інтерфейси (BCI), що дозволяють керувати пристроями силою думки.

#### 3. Web-ОС — Інтернет як операційна система

Інтернет стає єдиною платформою, де працюють усі програми.

Приклад: замість встановлення софту на ПК, усе доступно через хмару (Google Workspace, Figma, Canva).

#### 4. Глобальна інформаційна система

Миттєва передача інформації з будь-якої точки світу.

Приклад: хмарні сховища + 5G/6G зв'язок → відеоконференція у VR з людьми з різних країн без затримок.

#### 5. Покращений ШІ

Штучний інтелект стає контекстно-орієнтованим — розуміє емоції, адаптується під користувача.

Приклад: віртуальний асистент, який розпізнає тон голосу і пропонує допомогу, навіть якщо ви цього не просили.

#### 6. Природне спілкування

Можливість говорити з Інтернетом як з людиною — голосові та жестові інтерфейси.



Приклад: ChatGPT у голосовому режимі, Siri, Alexa, які підтримують живий діалог.

7. Прозорість

Усі дії в Інтернеті простежуються і фіксуються — як у блокчейні.

Приклад: історія транзакцій у криптовалюті або публічні реєстри.

8. Висока швидкість та надійність

- Завдяки 5G/6G, квантовим обчисленням та розподіленим мережам.
- Приклад: автономні авто отримують дані від сенсорів і серверів у реальному часі, приймаючи миттєві рішення.

Суть Web 4.0: це вже не просто "Інтернет речей" (IoT), а Інтернет інтеграції, де людина й машина взаємодіють на рівних, без бар'єрів, у режимі реального часу, з максимально природним інтерфейсом.

Таблиця 1.1 — Порівняльна таблиця розвитку Інтернету від Web 1.0 до Web 4.0

| Покоління | Період     | Основна концепція                                   | Характер взаємодії                                   | Ключові особливості                                                           | Приклади технологій/сервісів                           | Приклади використання                                                                                            |
|-----------|------------|-----------------------------------------------------|------------------------------------------------------|-------------------------------------------------------------------------------|--------------------------------------------------------|------------------------------------------------------------------------------------------------------------------|
| Web 1.0   | ~1990–2000 | Статичний Інтернет – лише читання                   | Одностороння: сайт → користувач                      | Статичні HTML-сторінки, відсутність динамічного контенту, мінімум мультимедіа | Yahoo!, Altavista, Geocities                           | Перегляд інформаційних сторінок, онлайн-каталоги, новинні портали                                                |
| Web 2.0   | ~2000–2010 | Інтерактивний Інтернет – читання + написання        | Двостороння: користувач ↔ сайт                       | Соціальна взаємодія, користувацький контент, AJAX, RSS                        | Facebook, Twitter, YouTube, Wikipedia                  | Соцмережі, блоги, відеохостинги, колективні енциклопедії                                                         |
| Web 3.0   | ~2010–2020 | Семантичний Інтернет – розуміння значення даних     | Персоналізація, контекстна взаємодія                 | ІІІ, семантичний пошук, блокчейн, IoT, API-інтеграції                         | Google Knowledge Graph, Ethereum, Siri                 | Рекомендаційні системи, голосові помічники, криптовалюти, розумний дім                                           |
| Web 4.0   | 2020+      | Симбіотичний Інтернет – людина+машина як єдине ціле | Природне спілкування (голос, жести, нейроінтерфейси) | 5G/6G, AR/VR, BCI (Neuralink), автономні пристрої, повна інтеграція з ІІІ     | ChatGPT, Alexa, автономні авто Tesla, Apple Vision Pro | Розумні асистенти, дистанційна медицина в реальному часі, повноцінні VR-наради, керування пристроями силою думки |

## 1.2 Стек протоколів Інтернету

### 1.2.1 Основні принципи Інтернету

Що таке "мережа доставки контенту"?

#### 1. Універсальність

Цей принцип означає, що Інтернет побудований так, щоб усі пристрої могли знаходити і спілкуватися один з одним, незалежно від того, де вони фізично знаходяться або яке обладнання використовують.

Ключові моменти:

- Універсальні ідентифікатори (адреси)  
Усі підключені пристрої мають унікальні IP-адреси (IPv4 або IPv6), завдяки яким їх можна ідентифікувати та знайти у мережі. Приклад: якщо у вас є смартфон у Києві та сервер у США, вони можуть "побачити" один одного завдяки IP.
- Можливість взаємодії будь-яких двох машин  
Не важливо, чи це ПК з Windows, чи сервер на Linux — якщо вони підключені до Інтернету, вони можуть обмінюватися даними.
- Незалежність користувацького інтерфейсу від фізичної мережі  
Те, як ви працюєте з сайтом чи додатком, не залежить від того, чи він передає дані через оптоволокно, Wi-Fi чи мобільний зв'язок.

#### 2. Віртуальна єдність

Цей принцип робить Інтернет для користувача єдиним простором, навіть якщо технічно він складається з мільйонів різних фізичних мереж.

Ключові моменти:

- Єдина віртуальна мережа для кінцевих користувачів  
Користувач бачить Інтернет як один цілісний ресурс, хоча насправді це безліч різних мереж і серверів.
- Рівнозначність всіх фізичних мереж

Незалежно від того, як саме підключений пристрій (через Wi-Fi, 4G, супутник), він є повноправною частиною Інтернету.

- Незалежність від фізичних параметрів та географічного масштабу  
Не важливо, чи з'єднання прокладене між сусідніми будинками або між континентами — принципи роботи ті самі.

Простими словами:

- Універсальність — це коли будь-який пристрій може знайти інший пристрій в Інтернеті.
- Віртуальна єдність — це коли всі ці пристрої виглядають для користувача як частини одного великого “світового комп’ютера”.

### 1.2.2 Сімейство протоколів TCP/IP

TCP/IP — це набір протоколів, які визначають, як комп’ютери обмінюються даними в Інтернеті. Вони працюють у вигляді 4-рівневої моделі комунікацій.

#### Рівень 1: Мережевий інтерфейс (Network Interface Layer)

Функція:

- Забезпечує фізичне підключення до конкретної мережі (кабель, Wi-Fi тощо).
- Керує передачею бітів та кадрів у межах локальної мережі.

Компоненти:

- Драйвер мережевого пристрою в операційній системі.
- Мережева плата (Ethernet-адаптер, Wi-Fi-модуль).

Приклад протоколу:

- Ethernet (дротова передача даних у локальній мережі).

Аналогія: це як дорога між будинками, по якій фізично їздять автомобілі з вантажем.

#### Рівень 2: Мережевий рівень (IP) (Internet Layer)

Основа TCP/IP: реалізує міжмережеве з'єднання — тобто об'єднує різні фізичні мережі в один Інтернет.

Функції:

- Маршрутизація пакетів (вибір найкращого шляху до отримувача).
- Передача даних без встановлення з'єднання (connectionless).
- Кожен пакет обробляється незалежно.

Проблеми: пакети можуть загубитися, дублюватися або прийти в неправильному порядку.

Аналогія: це як пошта, яка доставляє листи без гарантії, що вони дійдуть, і може доставити їх у різній послідовності.

### **Рівень 3: Транспортний рівень (Transport Layer)**

Відповідає за доставку даних між двома пристроями.

Два основних протоколи:

1. TCP (Transmission Control Protocol) — надійний

- Встановлює логічне з'єднання перед передачею.
- Ділить дані на пакети, відстежує їх отримання.
- Повторно надсилає втрачені пакети.
- Гарантує правильний порядок доставки.

Приклад: передача файлів через FTP, завантаження вебсторінок.

2. UDP (User Datagram Protocol) — швидкий, але ненадійний

- Не встановлює з'єднання.
- Не перевіряє доставку.
- Працює швидше, але без гарантій.

Приклад: онлайн-ігри, VoIP-дзвінки, стрімінг відео.

### **Рівень 4: Прикладний рівень (Application Layer)**

Тут працюють конкретні програми, що користуються нижніми рівнями.

Не цікавиться, як передаються дані — важливий лише формат і логіка обміну.

Приклади протоколів:

- TELNET — віддалена робота з терміналом.
- FTP — передача файлів.
- SMTP — електронна пошта.
- SNMP — керування мережевими пристроями.
- HTTP — передача вебсторінок.

Аналогія: це як сервіси доставки (піца, меблі, пошта), які використовують одну і ту ж транспортну систему, але доставляють різні речі.

### **Як працює модель TCP/IP на прикладі відкриття вебсторінки:**

1. Прикладний рівень (HTTP) — браузер запитує сторінку.
2. Транспортний рівень (TCP) — дані діляться на пакети та готуються до відправки.
3. Мережевий рівень (IP) — вибирається маршрут до сервера.
4. Мережевий інтерфейс (Ethernet/Wi-Fi) — пакети фізично передаються по мережі.

|                                        |                                 |
|----------------------------------------|---------------------------------|
| Прикладний рівень                      | WWW, FTP, Telnet E-mail та інші |
| Транспортний рівень                    | TCP, UDP                        |
| Мережевий рівень                       | IP, ICMP, IGMP                  |
| Канальний рівень (мережевий інтерфейс) | ARP, RARP                       |

Рисунок 1.4 — Сімейство протоколів TCP/IP

## **1.3 Переваги та недоліки TCP/IP**

### **1.3.1 Переваги**

#### **1. Відкритість**

TCP/IP — це відкритий стандарт, описаний у публічних специфікаціях (RFC). Будь-хто може реалізувати його у своєму програмному або апаратному продукті без ліцензійних обмежень.

Приклад: різні операційні системи (Windows, Linux, macOS) можуть обмінюватися даними через TCP/IP без додаткових налаштувань.

## 2. Масштабованість

Працює як у маленьких локальних мережах, так і в глобальному Інтернеті. Кількість підключених пристроїв може зростати практично без обмежень (особливо з IPv6).

Приклад: протокол працює однаково добре як для офісної мережі з 10 ПК, так і для світової мережі з мільярдами пристроїв.

## 3. Універсальність

Не залежить від типу комп'ютера, операційної системи чи фізичної мережі.

Приклад: смартфон на Android може передавати дані серверу на Linux через Wi-Fi або мобільний зв'язок.

## 4. Простота використання

Користувачі та розробники можуть працювати з TCP/IP через прості API та інтерфейси, не заглиблюючись у складні деталі.

Приклад: веброзробнику не потрібно знати, як фізично передаються пакети — він просто пише HTTP-запити.

### 1.3.2 Недоліки

#### 1. Проблеми захисту інформації (особливо в e-commerce)

У базовій реалізації TCP/IP немає вбудованого шифрування або автентифікації. Це особливо небезпечно при передаванні конфіденційних даних (наприклад, банківських карт).

Рішення: використовувати TLS/SSL (HTTPS).

#### 2. Невпорядкованість передачі пакетів

Протокол IP може доставити пакети у неправильному порядку (TCP виправляє це, але UDP — ні).

Проблема: у відеопотоці чи онлайн-іграх затримки можуть бути помітними.

### 3. Неможливість відстеження маршруту

Сам протокол IP не зберігає всю історію проходження пакета.

Рішення: використовувати додаткові інструменти, наприклад traceroute.

### 4. Обмеження для мультимедіа в реальному часі

TCP занадто повільний для потокового відео чи VoIP через механізм підтверджень. UDP швидший, але ненадійний.

Рішення: спеціалізовані протоколи (RTP, WebRTC).

### 5. Обмежений адресний простір IPv4

У IPv4 — лише близько 4,3 мільярда унікальних адрес, і вони майже вичерпані.

Рішення: перехід на IPv6, де адрес майже нескінченна кількість.

Висновок: TCP/IP — універсальний і масштабований стандарт, який зробив Інтернет можливим, але для сучасних вимог (захист, мультимедіа, адресація) його потрібно доповнювати новими технологіями та протоколами.

## 1.4 Майбутнє: IPv6

IPv6 — це нова версія Інтернет-протоколу, створена для заміни IPv4, у якого закінчується простір адрес. Він вирішує обмеження попередньої версії та додає нові можливості.

### 1. Чотириразове збільшення розміру адреси

IPv4 використовує 32 біти для адреси ( $\approx 4,3$  млрд унікальних адрес). IPv6 використовує 128 біт (в 4 рази більше бітів, але адрес більше в  $2^{32}$  разів).

Приклад: IPv4-адреса виглядає так — 192.168.0.1  
IPv6-адреса виглядає так — 2001:0db8:85a3:0000:0000:8a2e:0370:7334.

### 2. 4 квадрильйони адрес замість 4 мільярдів

Точніше, IPv6 підтримує  $3,4 \times 10^{38}$  адрес — це число настільки велике, що вистачить для кожного піщинку на Землі, якщо кожній виділити кілька адрес.

Це знімає потребу у NAT (підмережах із «сірою» адресацією).

### 3. Вбудовані функції захисту

IPv6 спочатку розроблявся з IPsec — технологією шифрування та перевірки автентичності пакетів.

Це підвищує безпеку передачі даних без додаткових надбудов.

### 4. Підтримка мультимедіа в реальному часі

IPv6 оптимізований для потокових відео, голосових дзвінків, онлайн-ігор і VR/AR. Має вбудовані механізми для пріоритету трафіку (QoS — Quality of Service).

### 5. Автоматичне реконфігурування адрес

Пристрій може самостійно отримати IPv6-адресу без DHCP-сервера (SLAAC — Stateless Address Autoconfiguration). Це спрощує підключення нових пристроїв до мережі, особливо в IoT.

Висновок: IPv6 — це не просто «більше адрес», а фундамент для нового покоління Інтернету, де кожен пристрій має свій унікальний глобальний IP, передача даних безпечніша, а мультимедіа працює швидше й стабільніше.

Таблиця 1.2 — Порівняльна таблиця IPv4 та IPv6

| Характеристика           | IPv4                            | IPv6                                                                                |
|--------------------------|---------------------------------|-------------------------------------------------------------------------------------|
| Довжина адреси           | 32 біти                         | 128 біт                                                                             |
| Кількість можливих адрес | $\approx 4,3$ млрд ( $2^{32}$ ) | $\approx 3,4 \times 10^{38}$ ( $2^{128}$ )                                          |
| Формат запису            | Десятковий, розділений крапками | Шістнадцятковий, розділений двокрапками (наприклад, 2001:0db8:85a3::8a2e:0370:7334) |



| Характеристика                            | IPv4                                                                 | IPv6                                                      |
|-------------------------------------------|----------------------------------------------------------------------|-----------------------------------------------------------|
|                                           | (наприклад, 192.168.0.1)                                             |                                                           |
| <b>Підтримка безпеки</b>                  | Не вбудована, потребує додаткових протоколів (IPSec не обов'язковий) | Вбудована підтримка IPSec                                 |
| <b>Підтримка QoS (Quality of Service)</b> | Обмежена                                                             | Розширена, з пріоритетами для мультимедіа                 |
| <b>Налаштування</b>                       | Переважно вручну або через DHCP                                      | Автоматична конфігурація (SLAAC)                          |
| <b>Фрагментація пакетів</b>               | Може виконувати як відправник, так і маршрутизатор                   | Виконує тільки відправник                                 |
| <b>Сумісність з іншими протоколами</b>    | Широко підтримується, але застаріває                                 | Сумісний через механізми переходу (Dual Stack, Tunneling) |
| <b>Обмеження</b>                          | Вичерпання адрес, складнощі з безпекою та мультимедіа                | Потребує оновлення обладнання та ПЗ для повної підтримки  |

## 1.5 Веб-сайт та Веб-додаток

У сучасному цифровому світі розуміння різниці між веб-сайтами та веб-додатками критично важливе для розробників — вибір правильної архітектури та технологій, бізнесу — визначення потреб і бюджету проекту, користувачів — розуміння можливостей і обмежень ресурсу.

Веб-сайт (Website) — набір статичних веб-сторінок, які містять інформацію для презентації без складної взаємодії з користувачем. Метафора: цифрова брошура або книга — ви читаєте, переглядаєте, але не змінюєте контент.

Веб-додаток (Web Application) — інтерактивна програма, що працює в браузері та дозволяє користувачеві виконувати певні дії й обробляти дані. Метафора: цифровий інструмент — ви працюєте з ним, виконуєте завдання, отримуєте результати.

Таблиця 1.3 — Порівняльна таблиця характеристик Веб-сайт vs Веб-додаток

| Критерій               | Веб-сайт                      | Веб-додаток                   |
|------------------------|-------------------------------|-------------------------------|
| Контент                | Статичний, інформаційний      | Динамічний, функціональний    |
| Інтерактивність        | Мінімальна (навігація, форми) | Висока (CRUD операції)        |
| Персоналізація         | Відсутня або обмежена         | Повна персоналізація          |
| База даних             | Не обов'язкова                | Зазвичай потрібна             |
| Складність розробки    | Низька-середня                | Середня-висока                |
| Швидкість завантаження | Висока                        | Залежить від функціональності |
| SEO-оптимізація        | Легка                         | Складніша                     |
| Вартість розробки      | Нижча                         | Вища                          |

### 1.5.1 Веб-сайти

Характеристики:

1. Статичність контенту — інформація змінюється рідко.
2. Односпрямована комунікація — від сайту до користувача.
3. Презентаційна функція — показати, розповісти, інформувати.
4. Простота структури — HTML, CSS, мінімум JavaScript.

Коли ми говоримо про статичність контенту, це означає, що інформація на сайті змінюється дуже рідко і зазвичай залишається незмінною протягом тривалого часу. Уявіть собі сайт юридичної фірми - інформація про партнерів компанії, їхні послуги та контактні дані можуть залишатися актуальними місяцями або навіть роками. Власники таких сайтів оновлюють контент лише тоді, коли відбуваються суттєві зміни в бізнесі, наприклад, додається новий партнер або змінюється адреса офісу.

Односпрямована комунікація означає, що інформація тече тільки в одному напрямку - від сайту до відвідувача. Це схоже на читання газети чи перегляд телевізійних новин - ви отримуєте інформацію, але не можете безпосередньо вплинути на те, що бачите. Власник сайту створює контент, а користувачі його споживають. Навіть якщо на сайті є форма зворотного зв'язку, це все одно не змінює основний контент для інших відвідувачів.

Презентаційна функція веб-сайтів полягає в тому, що їхня головна мета - це показати, розповісти або проінформувати аудиторію про щось конкретне. Сайт працює як цифрова вітрина або інформаційний стенд, де компанія або особа представляє себе, свої послуги чи ідеї широкому загалу.

Простота структури веб-сайтів виражається у використанні базових веб-технологій. Основу складають HTML для структури сторінки, CSS для оформлення та мінімальна кількість JavaScript для простих інтерактивних елементів, як-от випадаючі меню або слайдери зображень.

Типи веб-сайтів:

1. Корпоративні сайти — про компанію, послуги, контакти.

2. Блоги — особисті або тематичні публікації.
3. Портфоліо — демонстрація робіт творчих людей.
4. Інформаційні портали — новини, довідкова інформація.
5. Лендінг-сторінки — для конкретної рекламної кампанії.

Корпоративні сайти створюються компаніями для представлення свого бізнесу в Інтернеті. Такий сайт розповідає про історію компанії, її місію та цінності, описує послуги або товари, представляє команду співробітників та надає контактну інформацію. Це цифровий еквівалент корпоративної брошури, яка допомагає потенційним клієнтам познайомитися з бізнесом та прийняти рішення про співпрацю.

Блоги служать платформою для регулярної публікації статей на певну тематику або особистих роздумів автора. Вони можуть бути особистими щоденниками в інтернеті або спеціалізованими ресурсами, присвяченими конкретній темі - від кулінарії до технологій. Читачі приходять на блог за новою інформацією, порадами або розвагами.

Сайти-портфоліо використовують творчі люди та фахівці для демонстрації своїх робіт та досягнень. Фотографи показують свої найкращі знімки, дизайнери - проекти, а письменники - публікації. Це цифрова галерея, яка допомагає залучити нових клієнтів або роботодавців.

Інформаційні портали створюються для поширення новин, довідкової інформації або освітнього контенту. Вони функціонують як цифрові енциклопедії або газети, де відвідувачі можуть знайти актуальну інформацію з різних тем.

Лендінг-сторінки - це спеціальні одно-сторінкові сайти, створені для конкретної рекламної кампанії або маркетингової мети. Вони зосереджені на одній конкретній пропозиції та спрямовані на те, щоб спонукати відвідувача виконати певну дію, наприклад, купити товар або залишити контактні дані.

Переваги веб-сайтів:

- швидка розробка та запуск;

- низька вартість підтримки;
- відмінна SEO-оптимізація;
- висока швидкість завантаження;
- простота хостингу.

Швидкість розробки та запуску веб-сайтів пояснюється їхньою відносною простотою. Оскільки не потрібно створювати складну логіку обробки даних чи інтеграції з базами даних, розробник може сконцентруватися на створенні привабливого дизайну та організації контенту. Це означає, що простий корпоративний сайт можна створити за кілька тижнів, а не місяців.

Низька вартість підтримки обумовлена тим, що статичні сайти потребують мінімального технічного обслуговування. Немає необхідності регулярно оновлювати програмне забезпечення, створювати резервні копії баз даних або стежити за безпекою складних систем. Основні витрати обмежуються оплатою хостингу та періодичними оновленнями контенту.

Відмінна SEO-оптимізація досягається завдяки тому, що статичні сторінки легко індексуються пошуковими системами. Контент доступний відразу після завантаження сторінки, а структура HTML чітка та зрозуміла для пошукових роботів. Це допомагає сайтам краще ранжуватися в результатах пошуку.

Висока швидкість завантаження пояснюється простотою структури - браузеру потрібно завантажити лише HTML, CSS та зображення, без необхідності очікувати обробки даних на сервері або складних JavaScript-обчислень.

Простота хостингу означає, що такі сайти можна розмістити на будь-якому базовому веб-сервері без специфічних налаштувань або додаткового програмного забезпечення.

Недоліки веб-сайтів:

- обмежена функціональність;
- відсутність персоналізації;

- неможливість обробки складних даних.

Обмежена функціональність є природним наслідком простоти веб-сайтів. Вони не можуть виконувати складні операції з даними, такі як обробка платежів, управління користувацькими акаунтами або створення персоналізованого контенту. Все, що виходить за межі простої презентації інформації, вимагає переходу до більш складних рішень.

Відсутність персоналізації означає, що всі користувачі бачать однаковий контент незалежно від своїх інтересів, попереднього досвіду взаємодії з сайтом або особистих потреб. Сайт не може "запам'ятати" користувача або адаптуватися під його вподобання.

Неможливість обробки складних даних проявляється в тому, що веб-сайти не можуть виконувати обчислення, аналізувати інформацію або взаємодіяти з зовнішніми системами в реальному часі. Вони призначені для статичної подачі інформації, а не для її активної обробки.

### 1.5.2 Веб-додатки

Характеристики:

- динамічність — контент змінюється залежно від користувача;
- двостороння взаємодія — користувач впливає на систему;
- функціональність — виконання конкретних завдань;
- складна архітектура — фронтенд, бекенд, база даних.

Динамічність веб-додатків означає, що контент адаптується та змінюється залежно від конкретного користувача та його дій. Коли ви заходите в свій акаунт Gmail, ви бачите саме свої листи, а не чужі, коли переходите на Facebook - стрічка новин формується індивідуально на основі ваших друзів та інтересів. Це кардинально відрізняється від статичного сайту, де всі бачать однакову інформацію. Веб-додаток "розуміє", хто ви, що вас цікавить, які дії ви виконували раніше, та на основі цього створює унікальний контент спеціально для вас. Двостороння взаємодія полягає в тому, що користувач не

просто споживає інформацію, а активно впливає на роботу системи. Коли ви створюєте пост в соціальній мережі, редагуєте документ в Google Docs або робите покупку в інтернет-магазині, ваші дії змінюють стан системи та впливають на те, що бачитимуть інші користувачі. Система реагує на ваші дії миттєво, оновлюючи інтерфейс та зберігаючи зміни в базі даних. Функціональність веб-додатків виражається в тому, що вони створені для виконання конкретних завдань та вирішення практичних проблем користувачів. Замість простого показу інформації, вони дозволяють виконувати роботу - обробляти документи, керувати фінансами, спілкуватися з людьми, купувати товари або організовувати проекти. Кожен веб-додаток має чітко визначену мету та набір інструментів для її досягнення. Складна архітектура веб-додатків включає кілька взаємопов'язаних компонентів. Фронтенд - це те, що бачить користувач в браузері, інтерактивний інтерфейс, створений за допомогою сучасних технологій. Бекенд - це серверна частина, яка обробляє запити, виконує бізнес-логіку та керує даними. База даних зберігає всю інформацію користувачів та системи. Ці компоненти працюють разом, постійно обмінюючись даними для забезпечення безперебійної роботи додатку.

Типи веб-додатків:

- SPA (Single Page Application) — React, Vue, Angular додатки;
- CMS (Content Management System) — WordPress admin, Drupal;
- E-commerce платформи — інтернет-магазини з кошиком;
- Соціальні мережі — Facebook, Instagram, LinkedIn;
- Productivity tools — Google Workspace, Office 365;
- Фінансові додатки — інтернет-банкінг, платіжні системи.

SPA (Single Page Application) представляють собою сучасний підхід до створення веб-додатків, де вся взаємодія відбувається в межах однієї веб-сторінки. Замість переходу між різними сторінками, як у традиційних сайтах, SPA динамічно оновлюють контент на поточній сторінці. Такі додатки,

побудовані на фреймворках React, Vue або Angular, забезпечують швидку та плавну роботу, схожу на десктопні програми. Прикладом може служити Gmail, де ви можете читати пошту, писати листи та організовувати папки без перезавантаження сторінки.

Системи управління контентом (CMS) призначені для того, щоб люди без технічних знань могли легко створювати та редагувати вміст веб-сайтів. WordPress, який використовується мільйонами сайтів, надає зручний адміністративний інтерфейс, де можна писати статті, завантажувати зображення та налаштовувати зовнішній вигляд сайту без програмування.

E-commerce платформи створені для ведення онлайн-торгівлі та включають всю необхідну функціональність для інтернет-магазинів. Вони дозволяють клієнтам переглядати каталоги товарів, додавати покупки в кошик, оформляювати замовлення та здійснювати платежі, а власникам - керувати товарами, обробляти замовлення та аналізувати продажі.

Соціальні мережі представляють складні системи, що дозволяють людям спілкуватися, ділитися контентом та будувати онлайн-спільноти. Facebook, Instagram та LinkedIn створюють персоналізовані стрічки новин, дозволяють публікувати контент, коментувати та реагувати на дописи інших користувачів, формуючи складну мережу соціальних зв'язків.

Інструменти продуктивності замінюють традиційне офісне програмне забезпечення веб-версіями, що працюють в браузері. Google Workspace та Office 365 дозволяють створювати та редагувати документи, таблиці та презентації онлайн, працювати в команді над спільними проектами та зберігати файли в хмарі.

Фінансові додатки забезпечують безпечний доступ до банківських послуг та фінансових операцій через інтернет. Інтернет-банкінг дозволяє переглядати баланс рахунків, здійснювати перекази, оплачувати рахунки та керувати інвестиціями, забезпечуючи при цьому високий рівень безпеки та захисту персональних даних.

Переваги веб-додатків:



- високий рівень інтерактивності;
- персоналізований досвід користувача;
- можливість обробки складних даних;
- розширювана функціональність;
- монетизація через підписки/функції.

Високий рівень інтерактивності дозволяє користувачам не просто переглядати інформацію, а активно взаємодіяти з системою в реальному часі. Ви можете перетягувати елементи на екрані, отримувати миттєві відповіді на свої дії, бачити зміни, внесені іншими користувачами, та співпрацювати над спільними проектами так, ніби всі працюють в одній кімнаті.

Персоналізований досвід користувача означає, що кожна людина отримує унікальний інтерфейс та контент, адаптований під її потреби, вподобання та попередню активність. Spotify створює персональні плейлисти на основі ваших музичних уподобань, Amazon пропонує товари, які можуть вас зацікавити, а Netflix рекомендує фільми відповідно до вашого глядацького досвіду.

Можливість обробки складних даних дозволяє веб-додаткам виконувати серйозні обчислення, аналізувати великі обсяги інформації та інтегруватися з іншими системами. Фінансові додатки можуть розраховувати ризики інвестицій, CRM системи - аналізувати поведінку клієнтів, а проектні менеджери - оптимізувати розподіл ресурсів.

Розширювана функціональність означає, що веб-додатки можуть постійно еволюціонувати, додаючи нові можливості та адаптуючись до потреб користувачів. Розробники можуть випускати оновлення, додавати нові модулі та інтегрувати додаток з іншими сервісами без необхідності встановлення нового програмного забезпечення на комп'ютери користувачів.

Монетизація через підписки та функції відкриває різноманітні можливості для отримання прибутку від веб-додатку. Компанії можуть пропонувати базовий функціонал безкоштовно, а за преміум-можливості брати

помірну плату, створювати підписні моделі або отримувати комісії від транзакцій, що проходять через платформу.

Недоліки веб-додатків:

- висока складність розробки;
- значні витрати на розробку та підтримку;
- проблеми з SEO (особливо SPA);
- вища навантаження на сервер;
- потреба в безпеці та резервному копіюванні.

Висока складність розробки пояснюється тим, що створення веб-додатку вимагає глибоких знань багатьох технологій, розуміння архітектури систем та здатності вирішувати складні технічні проблеми. Розробникам потрібно освоїти не тільки мови програмування, а й принципи роботи баз даних, безпеки, масштабування та інтеграції різних компонентів.

Значні витрати на розробку та підтримку включають не тільки початкові інвестиції в створення додатку, а й постійні витрати на його обслуговування. Потрібно оплачувати роботу програмістів, системних адміністраторів, дизайнерів та тестувальників, підтримувати сервери, оновлювати безпеку, створювати резервні копії та забезпечувати безперебійну роботу системи.

Проблеми з SEO особливо гостро стоять для односторінкових додатків, оскільки пошукові системи традиційно краще індексують статичний HTML-контент. SPA часто генерують контент динамічно за допомогою JavaScript, що може бути складно для пошукових роботів, хоча сучасні технології поступово вирішують цю проблему.

Вище навантаження на сервер виникає через необхідність обробляти запити багатьох користувачів одночасно, виконувати складні обчислення, взаємодіяти з базами даних та забезпечувати швидкість відповіді. Це вимагає потужніших серверів та більших витрат на інфраструктуру.

Потреба в безпеці та резервному копіюванні є критично важливою, оскільки веб-додатки зберігають персональні дані користувачів та часто

обробляють фінансову інформацію. Необхідно постійно оновлювати системи безпеки, захищатися від хакерських атак, створювати резервні копії даних та мати план відновлення у випадку збоїв чи катастроф.

### **1.5.3 Технологічні відмінності**

#### **Технології веб-сайтів**

Frontend веб-сайтів базується на трьох основних технологіях, які є фундаментом будь-якої веб-сторінки. HTML (HyperText Markup Language) створює структуру та зміст сторінки, визначаючи де розташовані заголовки, параграфи, зображення та посилання. CSS (Cascading Style Sheets) відповідає за візуальне оформлення - кольори, шрифти, розміщення елементів та загальний дизайн сторінки. Базовий JavaScript додає мінімальну інтерактивність, наприклад, випадаючі меню, слайдери зображень або простеньку анімацію при наведенні курсору. Цей JavaScript зазвичай обмежується кількома рядками коду і не виконує складних операцій.

Backend для веб-сайтів часто взагалі відсутній, особливо у випадку статичних сайтів. Це означає, що всі файли HTML, CSS та JavaScript зберігаються на сервері в готовому вигляді і просто передаються браузеру користувача без будь-якої обробки. Коли ви відкриваєте таку веб-сторінку, сервер просто знаходить потрібний HTML-файл і відправляє його вашому браузеру, не виконуючи жодних додаткових операцій. У деяких випадках може використовуватися простий backend для обробки контактних форм або базової аналітики, але це не є обов'язковою вимогою.

База даних для простих веб-сайтів зазвичай не потрібна, оскільки вся інформація вже "вбудована" в HTML-файли. Контент сайту - тексти, зображення, структура сторінок - зберігається безпосередньо в файлах на сервері. Якщо потрібно змінити інформацію на сайті, веб-розробник редагує HTML-файли та завантажує оновлені версії на сервер.

Хостинг для веб-сайтів може бути надзвичайно простим, оскільки потрібно лише місце для зберігання файлів та можливість їх передачі користувачам через інтернет. Багато провайдерів пропонують статичний хостинг, де ви просто завантажуєте свої HTML, CSS та JavaScript файли в певну папку на сервері, і сайт стає доступним всім бажаним. Такий хостинг зазвичай дуже дешевий та надійний, оскільки не потребує складного програмного забезпечення або спеціальних налаштувань.

### **Технології веб-додатків**

Frontend веб-додатків використовує потужні JavaScript фреймворки, які дозволяють створювати складні інтерактивні інтерфейси. React, розроблений Facebook, дозволяє будувати інтерфейси з компонентів, які можуть динамічно оновлюватися залежно від дій користувача. Vue.js надає елегантний спосіб створення реактивних інтерфейсів з мінімальною кількістю коду. Angular, створений Google, пропонує повноцінну платформу для розробки великих корпоративних додатків. Ці фреймворки дозволяють створювати односторінкові додатки, які працюють швидко та плавно, надаючи користувачам досвід, схожий на роботу з десктопними програмами.

Backend веб-додатків представляє собою складну серверну систему, яка обробляє бізнес-логіку, керує даними та забезпечує безпеку. Node.js дозволяє писати серверний код на JavaScript, що спрощує розробку для команд, які вже знайомі з цією мовою. Python з фреймворками Django або Flask популярний завдяки своїй простоті та потужності для швидкої розробки. PHP залишається популярним вибором для веб-розробки, особливо в поєднанні з фреймворками Laravel або Symfony. Java та C# використовуються для великих корпоративних систем, де потрібна максимальна надійність та масштабованість. Backend відповідає за аутентифікацію користувачів, обробку запитів, взаємодію з базою даних та інтеграцію з зовнішніми сервісами.

База даних є обов'язковим компонентом веб-додатків, оскільки потрібно зберігати інформацію про користувачів, їхні дії, налаштування та всі дані, з якими працює система. MySQL та PostgreSQL є реляційними базами даних, які

відмінно підходять для зберігання структурованої інформації з чіткими зв'язками між різними типами даних. MongoDB представляє NoSQL підхід, який дозволяє зберігати дані у гнучкому JSON-подібному форматі, що особливо зручно для додатків з мінливими вимогами до структури даних. Вибір конкретної бази даних залежить від специфіки проекту, обсягів даних та особливостей їх використання.

Хостинг веб-додатків вимагає значно більш складної інфраструктури, оскільки потрібно не просто віддавати файли, а й виконувати серверний код, забезпечувати роботу баз даних та гарантувати безпеку. Такий хостинг включає операційну систему сервера, веб-сервер (Apache або Nginx), інтерпретатор або компілятор для backend мови програмування, систему управління базами даних, засоби резервного копіювання та моніторингу. Крім того, потрібно налаштовувати безпеку, SSL-сертифікати, брандмауери та регулярно оновлювати програмне забезпечення для захисту від вразливостей. Це робить хостинг веб-додатків значно дорожчим та складнішим в обслуговуванні порівняно зі статичними сайтами.



Рисунок 1.5 — Веб-сайт проти веб-додатку

1.5.4 Гібридні рішення у веб-розробці

Сучасні веб-ресурси все частіше розмивають межі між традиційними сайтами та повноцінними додатками, створюючи гібридні рішення, які поєднують кращі риси обох підходів. Такі рішення виникають через зростаючі очікування користувачів, які хочуть отримувати інформацію так само швидко, як на статичних сайтах, але водночас мати доступ до інтерактивних функцій, характерних для додатків.

Прогресивні веб-додатки (PWA)

Прогресивні веб-додатки представляють собою революційний підхід до веб-розробки, який дозволяє створювати ресурси, що поєднують найкращі характеристики традиційних веб-сайтів та нативних мобільних додатків. PWA

використовують сучасні веб-технології для створення швидких, надійних та захоплюючих користувацьких інтерфейсів, які працюють однаково добре як в браузері на комп'ютері, так і на мобільних пристроях.

Унікальність PWA полягає в тому, що вони працюють офлайн або при поганому інтернет-з'єднанні завдяки використанню *service workers* - спеціальних скриптів, які кешують важливий контент та ресурси. Коли користувач відвідує PWA вперше при стабільному інтернет-з'єднанні, додаток зберігає ключову інформацію локально на пристрої. Пізніше, навіть без доступу до мережі, користувач може переглядати збережений контент та виконувати базові функції. Наприклад, додаток новин може показувати раніше завантажені статті, а Twitter PWA дозволяє читати твіти, що були завантажені під час попереднього сеансу.

Ще однією революційною характеристикою PWA є можливість встановлення на пристрій без використання магазинів додатків. Користувач може додати PWA на домашній екран свого смартфона або створити ярлик на робочому столі комп'ютера, після чого додаток запускатиметься у повноекранному режимі, майже неможливо відрізнити від звичайного нативного додатку. При цьому PWA автоматично отримує оновлення через веб, не потребуючи ручного завантаження нових версій з магазину додатків.

Яскравими прикладами успішних PWA є Twitter Lite, який працює навіть при повільному інтернеті та споживає значно менше трафіку, Pinterest, що показав збільшення часу перебування користувачів на 40% після впровадження PWA технологій, та Starbucks PWA, який дозволяє замовляти каву навіть офлайн, синхронізуючи замовлення з сервером при відновленні з'єднання.

### **Сайти з елементами додатків**

Багато сучасних веб-ресурсів починають як традиційні інформаційні сайти, але поступово інтегрують інтерактивні елементи, характерні для веб-додатків. Такий підхід дозволяє зберегти переваги простих сайтів - швидкість завантаження, відмінну SEO-оптимізацію та низькі витрати на розробку -

водночас додаючи функціональність, яка покращує користувацький досвід.

Базовий інформаційний сайт залишається основою такого гібридного рішення. Головні сторінки містять статичну інформацію про компанію, послуги або продукти, що забезпечує швидке завантаження та ефективну роботу пошукових систем. Користувачі можуть легко знайти потрібну інформацію через Google або інші пошукові системи та швидко ознайомитися з основним контентом.

Інтерактивні форми виходять далеко за межі простих контактних форм, пропонуючи складні багатоетапні процеси. Наприклад, сайт страхової компанії може мати форму розрахунку вартості поліса, де користувач поетапно вводить інформацію про себе, своє майно та бажане покриття, а система в реальному часі розраховує вартість та показує різні варіанти. Такі форми зберігають прогрес користувача, дозволяють повертатися до попередніх кроків та надають персоналізовані рекомендації.

Чат-боти інтегруються в сайти для надання миттєвої підтримки клієнтам та відповідей на типові питання. Сучасні чат-боти можуть розуміти природну мову, навчатися з попередніх розмов та навіть передавати складні запити живим операторам. Наприклад, чат-бот на сайті банку може допомогти клієнту знайти найближче відділення, пояснити умови кредиту або навіть допомогти заповнити заявку на відкриття рахунку, при цьому весь процес відбувається без залишення основного сайту.

Калькулятори та віджети додають практичну цінність до інформаційних сайтів, дозволяючи користувачам виконувати корисні розрахунки або отримувати персоналізовані рекомендації. Сайт банку може включати калькулятор іпотеки, який враховує різні параметри кредиту та показує графік платежів. Сайт фітнес-тренера може мати калькулятор ВМІ та розробник дієти на основі індивідуальних параметрів користувача. Ці інструменти працюють миттєво, не потребують реєстрації, але надають достатньо цінності, щоб користувачі повертались на сайт знову та рекомендували його іншим.



Такі гібридні рішення особливо ефективні для бізнесу, оскільки дозволяють поступово нарощувати функціональність відповідно до потреб користувачів та бюджетних можливостей. Компанія може почати з простого корпоративного сайту, а потім додавати інтерактивні елементи на основі аналізу поведінки відвідувачів та їхніх запитів.

### **1.5.5 Тренди та майбутнє**

JAMstack архітектура представляє собою революційний підхід до створення веб-ресурсів, який поєднує швидкість та безпеку статичних сайтів з гнучкістю та функціональністю сучасних веб-додатків. Аббревіатура JAM розшифровується як JavaScript, APIs та Markup, що відображає основні компоненти цієї технології. Сутність підходу полягає в тому, що основний контент сайту генерується заздалегідь у вигляді статичних HTML-файлів, що забезпечує блискавичну швидкість завантаження та відмінну SEO-оптимізацію. Водночас динамічна функціональність додається через JavaScript та зовнішні API сервіси. Наприклад, блог може мати статичні сторінки зі статтями, але коментарі завантажуються динамічно через API, форма зворотного зв'язку обробляється через сторонній сервіс, а пошук реалізується через JavaScript. Такий підхід дозволяє отримати найкраще з двох світів - швидкість статичних сайтів та інтерактивність веб-додатків.

Headless CMS концепція кардинально змінює традиційний підхід до управління контентом, розділяючи процес створення контенту та його презентації користувачам. У традиційних CMS, таких як WordPress, контент та його відображення тісно пов'язані - коли ви створюєте статтю, система одразу визначає, як вона виглядатиме на сайті. Headless CMS натомість зберігає контент у "чистому" вигляді без прив'язки до конкретного дизайну або платформи. Контент-менеджери створюють та редагують матеріали через зручний інтерфейс, але цей контент може потім відображатися на веб-сайті, в мобільному додатку, на цифрових екранах або навіть передаватися голосовим

асистентам. Це особливо цінно для компаній, які хочуть підтримувати присутність на різних платформах - один і той же контент може автоматично адаптуватися для сайту, iOS додатку, Android додатку та соціальних мереж.

Serverless технології дозволяють розробникам створювати та запускати код без необхідності керувати власними серверами або складною інфраструктурою. Замість того, щоб підтримувати цілодобово працюючі сервери, які споживають ресурси навіть коли нічого не відбувається, serverless підхід запускає код лише тоді, коли це потрібно. Наприклад, функція обробки платежів активується лише в момент здійснення покупки, функція відправки електронної пошти - лише при заповненні форми зворотного зв'язку. Хмарні провайдери, такі як AWS Lambda, Google Cloud Functions або Azure Functions, автоматично керують всією інфраструктурою, масштабуванням та безпекою. Розробники платять лише за фактичне використання ресурсів, що робить такий підхід надзвичайно економічним для проектів з нерівномірним навантаженням.

Мікрофронтови застосовують принципи мікросервісної архітектури до фронтенд розробки, дозволяючи розбивати великі веб-додатки на менші, незалежно розроблювані та розгортвані частини. Замість монолітного додатку, який розробляється однією великою командою, кожна частина інтерфейсу може створюватися окремою командою з використанням різних технологій. Наприклад, у великому e-commerce додатку команда каталогу може розробляти секцію товарів на React, команда платежів - секцію оплати на Vue.js, а команда профілю користувача - особистий кабінет на Angular. Ці частини об'єднуються в єдиний досвід для користувача, але технічно залишаються незалежними. Такий підхід дозволяє командам працювати паралельно, використовувати найкращі інструменти для конкретних завдань та швидше випускати оновлення.

Майбутні напрямки:

1. Розмивання меж між сайтами та додатками.
2. Зростання ролі ШІ в персоналізації.

3. Розвиток голосових інтерфейсів.
4. Інтеграція з IoT пристроями.

### 1.5.6 Сервер: основа веб-простору

Щоб веб-сторінка або веб-додаток стали доступними всім користувачам у світі, вони мають зберігатися на спеціальному комп'ютері — сервері. Сервер працює безперервно, підключений до Інтернету та готовий приймати запити від клієнтів (переважно браузерів).

Головні функції сервера:

#### 1. Зберігання даних

Сервер зберігає усі необхідні файли:

- HTML — структура сторінки;
- CSS — стилі оформлення;
- JavaScript — логіка та інтерактивність;
- медіафайли (зображення, відео);
- серверний код (якщо є).

#### 2. Обробка запитів

Коли користувач вводить адресу сайту (URL), браузер відправляє запит на відповідний сервер.

#### 3. Надсилання відповіді

Сервер аналізує запит і надсилає назад дані — наприклад, готову веб-сторінку, яку браузер показує користувачу.

Без сервера сайт чи додаток не можуть існувати в Інтернеті. Він виконує роль посередника між користувачем і збереженою інформацією, хоча користувач найчастіше його "не бачить".

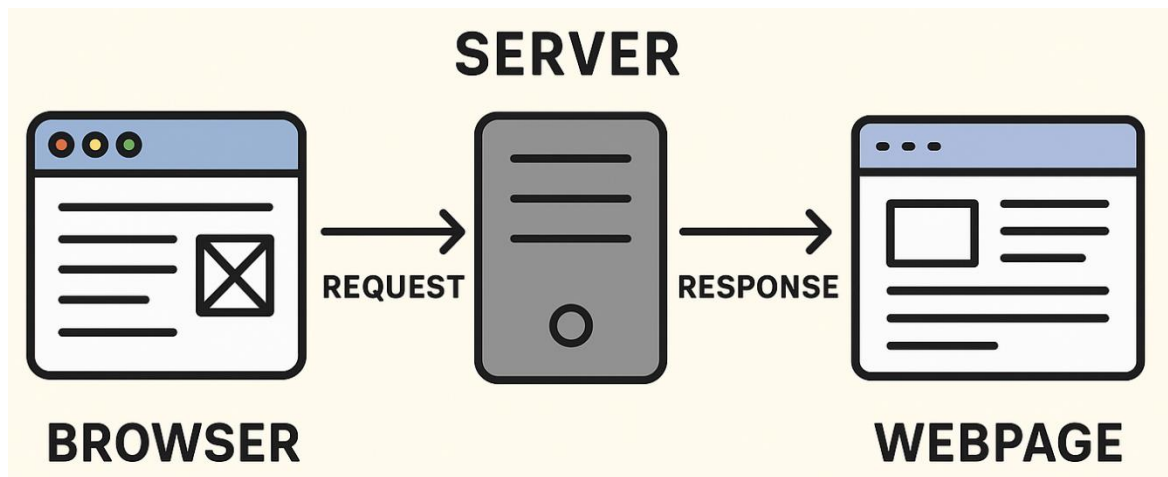


Рисунок 1.6 — Як працює сервер

Коли ми говоримо про веб-сервер, ми маємо на увазі не лише фізичне обладнання, а й програмне забезпечення, яке на ньому працює. Це програмне забезпечення називається веб-серверною програмою (або просто веб-сервером). Воно відповідає за прийом, обробку та надсилання HTTP-запитів.

Функції веб-серверної програми:

- приймає запити (наприклад, коли користувач відкриває сайт);
- обробляє їх (передає в потрібні скрипти, бази даних, API);
- відправляє відповідь (HTML-сторінку, JSON-дані, медіафайл тощо).

### Основні веб-серверні програми

#### 1. Apache HTTP Server

- Найстаріший та один із найпоширеніших серверів.
- Переваги:
  - стабільність і надійність;
  - гнучка система модулів (можна підключати додаткові можливості);
  - кросплатформність (Windows, Linux, macOS).
- Використовується для:
  - статичного контенту (HTML, CSS, JS);
  - динамічних сайтів (PHP, Python, Perl тощо).

#### 2. Nginx (Engine X)

Славиться високою продуктивністю і швидкістю, особливо при великій кількості одночасних з'єднань.

Часто використовується:

- як зворотний проксі-сервер (розподіл навантаження між серверами);
- для статичного контенту;
- у поєднанні з Apache або іншими серверами.

### 3. Microsoft Internet Information Services (IIS)

- Продукт компанії Microsoft.
- Тісно інтегрований із Windows Server.
- Основна сфера використання: корпоративні середовища на основі технологій Microsoft (.NET, ASP.NET).

### 4. LiteSpeed

Комерційний веб-сервер, розроблений як швидша альтернатива Apache.

Переваги:

- сумісність із конфігураціями Apache (простий перехід);
- краща оптимізація продуктивності та безпеки.

Вибір залежить від: типу веб-додатка (статичний сайт, динамічний портал, e-commerce); навантаження (кількість відвідувачів); операційної системи (Linux, Windows); потреб у безпеці та швидкодії.

Таблиця 1.4 — Порівняльна таблиця веб-серверів

| Критерій       | Apache<br>HTTP Server       | Nginx<br>(Engine X)    | Microsoft IIS            | LiteSpeed                                  |
|----------------|-----------------------------|------------------------|--------------------------|--------------------------------------------|
| Платформа      | Linux,<br>Windows,<br>macOS | Linux,<br>Windows      | Windows<br>Server        | Linux<br>(основна),<br>частково<br>Windows |
| Продуктивність | Висока, але<br>менш         | Дуже висока,<br>чудово | Середня,<br>оптимізована | Дуже висока,<br>оптимізована               |

| Критерій     | Apache HTTP Server                                 | Nginx (Engine X)                                               | Microsoft IIS                                               | LiteSpeed                                                |
|--------------|----------------------------------------------------|----------------------------------------------------------------|-------------------------------------------------------------|----------------------------------------------------------|
|              | ефективна під великим навантаженням                | працює з великою кількістю одночасних запитів                  | для роботи в середовищі Microsoft                           | для інтенсивних навантажень                              |
| Призначення  | Статичний та динамічний контент (PHP, Python тощо) | Статичний контент, зворотний проксі, балансування навантаження | Додатки на основі .NET, ASP.NET, корпоративні рішення       | Статичний і динамічний контент (WordPress, e-commerce)   |
| Гнучкість    | Дуже гнучкий, підтримує безліч модулів             | Легко масштабується, менш модульний ніж Apache                 | Обмежена поза екосистемою Microsoft                         | Сумісний з Apache-конфігураціями, підтримка сучасних CMS |
| Ліцензія     | Відкрите ПЗ (Apache License 2.0)                   | Відкрите ПЗ (2-clause BSD license)                             | Пропрієтарна (Microsoft)                                    | Комерційна (є безкоштовна OpenLiteSpeed)                 |
| Використання | Універсальний вибір для сайтів різного масштабу    | Великі проекти, що потребують високої швидкодії та проксі      | Переважно корпоративний сектор (інтеграція з Windows Active | Великі сайти, онлайн-магазини, хостинг WordPress/Magento |

| Критерій                              | Apache HTTP Server                                                                                                              | Nginx (Engine X)                                                                                               | Microsoft IIS                                                                                  | LiteSpeed                                                                                                          |
|---------------------------------------|---------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------|
|                                       |                                                                                                                                 |                                                                                                                | Directory, Exchange)                                                                           |                                                                                                                    |
| <b>Складність налаштування</b>        | Середня                                                                                                                         | Середня, потребує знань конфігураційних файлів                                                                 | Легка у Windows-інфраструктурі                                                                 | Середня                                                                                                            |
| <b>Рекомендовано обирати, якщо...</b> | - Потрібна гнучкість і підтримка різних мов програмування- Сайт середнього навантаження- Потрібні численні модулі та розширення | - Необхідна висока продуктивність- Велике навантаження, балансування трафіку- Використання як зворотний проксі | - Ви працюєте у корпоративному середовищі Microsoft- Плануєте використовувати .NET або ASP.NET | - Потрібна висока швидкодія- Планується велика кількість відвідувачів- Використовується WordPress, Joomla, Magento |

### 1.5.7 Браузер

Браузер — це не просто програма для перегляду веб-сторінок, а справжній посередник між користувачем і величезним інформаційним простором Інтернету. Коли людина вводить адресу сайту або натискає на посилання, браузер починає складний процес, який для користувача виглядає миттєвим. Спершу він формує запит і надсилає його до веб-сервера, на якому зберігається необхідний контент. Сервер у відповідь передає набір файлів, що

складається з HTML-коду, стилів оформлення CSS і сценаріїв JavaScript. У цей момент браузер ще не показує готову сторінку, а виконує низку операцій, які можна порівняти з перекладом тексту іноземною мовою: він аналізує отриманий код, створює внутрішню структуру документа, визначає, як мають виглядати окремі елементи, і лише потім формує для користувача зрозумілий інтерфейс.

Особливу роль у цьому процесі відіграє рендеринг — побудова візуальної моделі веб-сторінки. Браузер розкладає отримані дані на складові, застосовує стилі, визначає розміщення текстів і зображень, відтворює анімацію та інтерактивні елементи. Саме завдяки цьому звичайний набір тегів і коду перетворюється на сайт, яким можна користуватися. Коли на сторінці є динамічні елементи, вступає в дію JavaScript-рушій, що виконує програмний код, реагує на дії користувача, змінює вміст без повторного завантаження.

Браузер також дбає про те, щоб робота з Інтернетом була швидкою і безпечною. Він зберігає у кеші частину даних, які вже завантажувалися раніше, щоб наступного разу відкривати сторінки швидше. Крім того, він може пам'ятати налаштування сайтів і тимчасову інформацію у вигляді cookies, що полегшує роботу з веб-додатками. Кожен браузер має власний "рушій", тобто механізм, який відповідає за обробку та відображення вмісту: у Google Chrome це V8 і Blink, у Mozilla Firefox — SpiderMonkey і Gecko, у Safari — WebKit, а сучасний Microsoft Edge базується на Chromium і використовує ті ж технології, що й Chrome. Попри різницю у внутрішніх механізмах, усі вони мають одну мету — зробити взаємодію з веб-простором максимально зручною, швидкою та надійною.

Основні функції браузера:

1. Надсилання запитів

- Коли ви вводите адресу сайту (URL) або натискаєте посилання, браузер формує HTTP/HTTPS-запит.
- Запит відправляється на веб-сервер, що зберігає потрібний сайт.



## 2. Відображення (рендеринг) контенту

Отримавши відповідь (HTML, CSS, JavaScript), браузер:

- парсить HTML (аналізує структуру сторінки);
- застосовує CSS (стилі: кольори, шрифти, розташування);
- будує DOM-дерево (структуру сторінки);
- відображає готовий результат.

## 3. Виконання скриптів

- JavaScript оживляє сторінку: анімації, обробка натискань, форми, інтерактивні елементи.
- Браузер має вбудований JS-рушій (наприклад, V8 у Chrome, SpiderMonkey у Firefox).

## 4. Зберігання даних

- Cookies – маленькі файли з налаштуваннями або даними користувача.
- Кеш – тимчасове збереження елементів сторінок для швидшого завантаження.
- LocalStorage / SessionStorage – локальне сховище для веб-додатків.

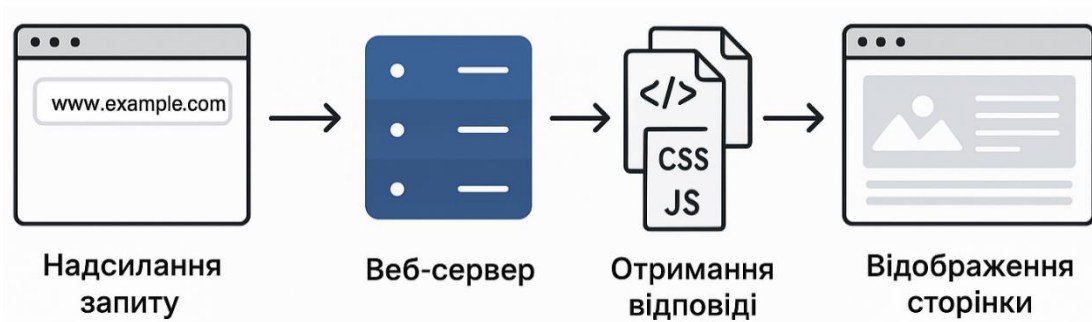


Рисунок 1.7 — Робота браузера

Популярні браузери та їх рушії:

- Google Chrome — V8 (JavaScript) + Blink (рендеринг);
- Mozilla Firefox — SpiderMonkey + Gecko;
- Microsoft Edge — Blink + V8 (новий Edge базується на Chromium);
- Safari — WebKit (JavaScript: JavaScriptCore).

Чому браузери відрізняються? Різні рушії можуть по-різному інтерпретувати веб-код (через що іноді сторінки виглядають інакше). Рівень підтримки сучасних технологій (HTML5, CSS3, WebAssembly) також залежить від браузера.

### 1.5.8 Статичний та динамічний контент

Розуміння різниці між статичним і динамічним контентом є ключовим для розуміння роботи вебу.

Статичний контент — це інформація, яка не змінюється, якщо її не змінить вручну розробник. Коли ви запитуєте таку сторінку, сервер просто відправляє вам готовий HTML-файл, як він є. Наприклад, сторінка "Про нас" або "Контакти" на звичайному сайті. Тут немає взаємодії з базою даних чи складних обчислень на сервері.

Динамічний контент — це контент, який генерується "на льоту" (on the fly) при кожному запиті. Сервер не просто віддає готовий файл, а виконує певний код, звертається до бази даних, обробляє інформацію і вже після цього створює HTML-сторінку для користувача. Приклади: стрічка новин у соціальних мережах, результати пошуку на Google, персоналізований кошик в онлайн-магазині. Кожен користувач може бачити різний контент, навіть якщо вони відвідують одну й ту саму URL-адресу.

Статичні додатки (Static Web Apps) складаються переважно з HTML, CSS та JavaScript, які вже скомпільовані та розміщені на сервері. Вони не мають складної серверної логіки та не використовують бази даних. Уся інтерактивність відбувається на стороні клієнта (в браузері) за допомогою JavaScript.

Переваги статичних додатків:

- Швидкість: Вони завантажуються дуже швидко, оскільки серверу не потрібно нічого обробляти.
- Безпека: Відсутність серверної логіки знижує ризик атак.

- Простота розгортання: Їх можна розмістити на звичайних хостингах або CDN (Content Delivery Network).

Динамічні додатки (Dynamic Web Apps) використовують сервер для генерації контенту. Вони складаються з клієнтської частини (фронтенд) і серверної (бекенд). Бекенд відповідає за:

- Зберігання та обробку даних: Взаємодіє з базами даних (MySQL, PostgreSQL, MongoDB).
- Аутентифікацію та авторизацію: Управляє обліковими записами користувачів.
- Генерацію контенту: Формує HTML-сторінки, які потім відправляються клієнту.

Приклади динамічних додатків: більшість сучасних веб-сервісів — інтернет-банкінг, системи управління відносинами з клієнтами (CRM), платформи електронної комерції. Вони є основою сучасної інтерактивної мережі.

## 1.6 Базові технології фронтенду та бекенду

### 1.6.1 Ключові технології та поняття

Клієнт (браузер, мобільний застосунок) надсилає HTTP-запит на сервер, сервер виконує бізнес-логіку, звертається до БД чи зовнішніх сервісів і повертає HTTP-відповідь (HTML/CSS/JS або JSON).

Користувач → Браузер

└─(1) DNS пошук домену

└─(2) TLS рукоштовування (HTTPS)

└─(3) HTTP запит → [CDN/Reverse Proxy] → App Server → DB/Cache

←─(4) HTTP відповідь (HTML/JSON, статичні файли через CDN)

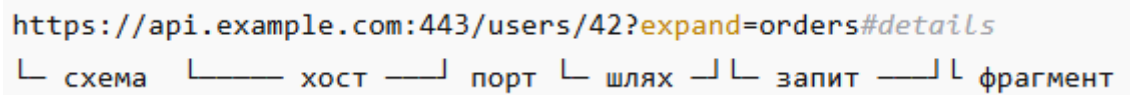
←─(5) Рендер UI у браузері

HTTP — безстановий (stateless) протокол: кожен запит самодостатній. «Стан» користувача ми зберігаємо через cookies/сесії/JWT на клієнті або в БД/кеші на сервері.

CDN прискорює доставку статичних файлів (зображення, CSS, JS).

Reverse проху (наприклад, Nginx) термінує TLS, кешує, балансує навантаження.

URL визначає, куди і що саме запитує клієнт. DNS перетворює домен example.com на IP-адресу сервера. Порт за замовчуванням: 80 (HTTP), 443 (HTTPS).



https://api.example.com:443/users/42?expand=orders#details

└─ схема ─┬── хост ─┬── порт ─┬── шлях ─┬── запит ─┬── фрагмент

Рисунок 1.8 — URL та DNS

### HTTP: методи, коди та заголовки

Методи: GET (читання), POST (створення), PUT/PATCH (оновлення), DELETE (видалення), HEAD, OPTIONS.

Коди стану (прикладі):

- 200 OK, 201 Created, 204 No Content
- 301/302 Redirect, 304 Not Modified
- 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found
- 500 Internal Server Error, 503 Service Unavailable

Заголовки говорять як обробляти вміст:

- Content-Type: application/json — тип даних у тілі.
- Асепт: application/json — що клієнт готовий прийняти.
- Authorization: Bearer <token> — автентифікація.
- Cache-Control, ETag, Last-Modified — кешування.
- Set-Cookie, Cookie — сесії/стан.
- Origin, Access-Control-Allow-Origin — **CORS** (доступ між доменами).

Міні-приклад сирого HTTP-запиту/відповіді:

```
GET /api/users/42 HTTP/1.1
```

```
Host: example.com
Accept: application/json
Authorization: Bearer <JWT>

HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: max-age=60

{"id":42,"name":"Olena","roles":["user"]}
```

Уявіть собі Інтернет як величезну бібліотеку, де мільярди документів, зображень, відео та інших файлів постійно передаються між комп'ютерами по всьому світу. Але як же різні пристрої та програми розуміють, що саме їм передали і як з цим працювати?

Почнемо з найосновніших технологій, які формують основу веб-сторінок. HTML, або HyperText Markup Language, – це мова розмітки, яка визначає структуру та зміст веб-документів. Коли ви відкриваєте будь-яку веб-сторінку, браузер отримує HTML-код, який містить інформацію про те, де розміщені заголовки, абзаци тексту, зображення, посилання та інші елементи. HTML працює як скелет веб-сторінки – він говорить браузеру, що є що, але не визначає, як це має виглядати.

За зовнішній вигляд веб-сторінок відповідає CSS, або Cascading Style Sheets. Якщо HTML – це скелет, то CSS – це одяг та макіяж веб-сторінки. Ця технологія дозволяє веб-розробникам визначати кольори, шрифти, розміщення елементів, анімації та багато іншого. CSS працює в тандемі з HTML, отримуючи інформацію про структуру документа та застосовуючи до неї стилістичні правила. Завдяки CSS одна й та сама HTML-структура може виглядати як діловий сайт банку або як яскравий блог про подорожі.

Але сучасні веб-сторінки не просто показують статичну інформацію – вони взаємодіють з користувачами, реагують на їхні дії, завантажують нові дані без перезавантаження сторінки. За всю цю динаміку відповідає JavaScript

– мова програмування, яка виконується безпосередньо в браузері. JavaScript може змінювати вміст сторінки в реальному часі, обробляти натискання кнопок, перевіряти правильність заповнення форм, створювати складні інтерактивні елементи. Сьогодні JavaScript настільки потужний, що на ньому створюють повноцінні додатки, які працюють прямо в браузері.

Тепер поговоримо про формати обміну даними між різними системами. Коли ваш браузер потребує отримати інформацію з сервера – наприклад, список новин, дані про погоду або результати пошуку – він надсилає запит, а сервер повертає відповідь у певному форматі. Найпопулярнішим форматом для такого обміну є JSON, або JavaScript Object Notation.

JSON став де-факто стандартом для API саме завдяки своїй простоті та зручності. Цей формат використовує синтаксис, подібний до того, як в JavaScript описують об'єкти, що робить його інтуїтивно зрозумілим для розробників. JSON може передавати різні типи даних – числа, рядки, булеві значення, масиви та вкладені об'єкти. Наприклад, інформація про користувача може включати його ім'я, вік, список хобі та адресу, яка сама складається з вулиці, міста та поштового коду. JSON елегантно справляється з такими складними структурами даних.

Хоча JSON домінує в сучасному веб-розробленні, варто згадати і про XML – eXtensible Markup Language. XML був дуже популярний на зорі розвитку веб-сервісів і досі використовується в деяких галузях, особливо у великих корпоративних системах та урядових структурах. XML більш багатослівний за JSON, але він пропонує більше можливостей для валідації даних та має розвинену екосистему стандартів. У XML кожен елемент має відкриваючий та закриваючий теги, що робить структуру даних дуже явною, але й збільшує розмір повідомлень.

Окремо варто розглянути ситуації, коли потрібно передавати файли – зображення, документи, відео та інший бінарний вміст. Для цього використовується спеціальний формат під назвою FormData або multipart/form-data. Цей формат дозволяє в одному запиті об'єднати як звичайні текстові поля

форми, так і файли. Коли ви завантажуєте світлину в соціальну мережу разом з описом, браузер формує multipart/form-data запит, який містить і текст опису, і бінарні дані зображення.

Щоб все це різноманіття форматів правильно обробляється, існує система MIME-типів, або типів вмісту. MIME-тип – це своєрідна етикетка, яка вказує, що саме міститься в переданих даних. Коли сервер надсилає файл браузеру, він обов'язково вказує MIME-тип у заголовках відповіді. Наприклад, text/html означає HTML-документ, application/json – дані в форматі JSON, image/jpeg – зображення в форматі JPEG. Браузер, отримавши ці дані разом з MIME-типом, знає, як їх обробити – показати як веб-сторінку, розпарсити як JSON-об'єкт або відобразити як картинку.

MIME-типи особливо важливі, коли мова йде про завантаження файлів. Сервер може перевірити MIME-тип завантажуваного файлу та переконатися, що користувач дійсно намагається завантажити зображення, а не виконуваний код, який може бути небезпечним.

Розуміння цих форматів та їх призначення критично важливе для сучасного веб-розробника. HTML, CSS та JavaScript утворюють основу фронтенд-розробки, JSON забезпечує ефективний обмін даними між клієнтом та сервером, а правильне використання MIME-типів гарантує безпеку та коректну обробку різноманітного вмісту. Усі ці технології працюють разом, створюючи той багатий, інтерактивний досвід, який ми сприймаємо як сучасний Інтернет.

JSON-приклад для API:

```
{
  "id": 42,
  "name": "Olena",
  "orders": [{ "number": "A123", "total": 399.00 }]
}
```

Рендеринг інтерфейсу: MPA, SPA, CSR, SSR, SSG, ISR.

MPA (Multi-Page App): сервер віддає повноцінні HTML-сторінки для кожного переходу.

SPA (Single-Page App): одна HTML-сторінка, далі маршрути — у браузері; дані — через API.

CSR (Client-Side Rendering): HTML формується в браузері JS-кодом.

SSR (Server-Side Rendering): HTML формується на сервері (краще SEO/Time-to-First-Byte).

SSG (Static Site Generation): HTML генерується під час білду (дуже швидко, чудово кешується).

ISR (Incremental Static Regeneration): гібрид — статичні сторінки з контрольованим оновленням.

Стан: клієнтський і серверний. Клієнтський: оперативна пам'ять додатка (React/Vue), LocalStorage, SessionStorage, IndexedDB, cookies. Серверний: БД (PostgreSQL/MySQL/MongoDB), кеш (Redis, Memcached). Джерело істини (single source of truth) зазвичай — сервер/БД; клієнт синхронізується через API.

Приклад збереження невеликого стану:

```
localStorage.setItem('theme', 'dark');
const theme = localStorage.getItem('theme'); // "dark"
```

Стилі API: REST, GraphQL, WebSockets, gRPC. REST — робота з ресурсами через URL та HTTP-методи. Ідемпотентність: GET/PUT/DELETE бажано не змінюють результат при повторі. GraphQL — клієнт сам описує форму даних; один ендпоінт, менше «over/under-fetching». WebSockets/Server-Sent Events — двосторонній/односторонній стрімінг для реального часу (чати, лайв-таблиці). gRPC — високопродуктивні двійкові RPC поверх HTTP/2; популярний між сервісами.

REST-приклад (curl):

```
curl -H "Accept: application/json"
https://api.example.com/users/42
```

GraphQL-приклад (запит):



```
query {
  user(id: 42) { id name orders { number total } }
}
```

Безпека — базовий набір:

- HTTPS/TLS: шифрування трафіку і перевірка автентичності сервера (сертифікати).
- AuthN vs AuthZ: автентифікація (хто ти?) проти авторизації (що тобі дозволено?).
- Сесії та токени: cookie-сесії (часто з CSRF-захистом) або JWT у Authorization: Bearer.
- CORS: контроль доступу до ресурсів з інших доменів.
- Валідація/санітизація даних: протидія XSS, SQL-ін'єкціям, CSRF.
- Політики браузера: Content-Security-Policy, SameSite для cookies тощо.

Міні-приклад заголовків безпеки у відповіді:

```
Content-Security-Policy: default-src 'self'
Strict-Transport-Security: max-age=31536000;
includeSubDomains
X-Content-Type-Options: nosniff
```

Продуктивність і масштабування:

- Кеші: браузерний (Cache-Control, ETag), CDN, reverse proxy, Redis на рівні додатка, кеш запитів БД.
- Стиснення: gzip/brotli; HTTP/2/3 (мультиплексування, швидші встановлення з'єднання).
- Оптимізація фронтенду: мінімізація JS/CSS, code splitting, lazy-loading, критичний CSS, оптимізація зображень.
- Горизонтальне масштабування: балансувальник → кілька інстансів застосунку (контейнери).
- Вертикальне: потужніші ресурси на одному вузлі (RAM/CPU/SSD).

- Індеси БД: прискорюють запити; важливе правильне проєктування схем.

Розгортання та середовища:

- Dev → Stage → Prod: чітке розділення для якості.
- CI/CD: автоматичні тести, білд, деплой (GitHub Actions, GitLab CI, Jenkins).
- Контейнери: Docker (ізоляція), оркестрація: Kubernetes (масштабування/самовідновлення).
- Serverless/FaaS: функції за подією (AWS Lambda, Cloud Functions) — менше операційки, оплата за використання.

Спостережуваність (Observability):

- Логи (структуровані), метрики (латентність, помилки, використання ресурсів), трейсинг (end-to-end шлях запиту).
- SLA/SLO/SLI: контракт рівня сервісу, цілі та показники якості.

Часто плутають — розрізняємо терміни:

- Бібліотека vs фреймворк: бібліотека додає функції; фреймворк диктує архітектуру («інверсія управління»).
- API vs SDK: API — контракт/протокол; SDK — набір інструментів/код для роботи з API.
- Моноліт vs мікросервіси: один застосунок vs набір дрібних сервісів з чіткими контрактами.
- RDBMS vs NoSQL: таблиці/SQL/ACID vs документи/ключ-значення/граф/колонки (гнучкість/масштаб).
- ORM: прошарок, що відображає об'єкти коду на таблиці БД (ActiveRecord, Data Mapper).

Міні-демонстрації:

1) Перегляд сирової відповіді сервера

```
curl -i https://example.com
```

2) Запит із JSON-тілом

```
curl -X POST https://api.example.com/users \
  -H "Content-Type: application/json" \
  -d '{"name":"Olena","email":"olena@example.com"}'
```

### 3) Найпростіший REST-ендпоінт на Express (ідея, без залежностей)

```
import express from 'express';
const app = express();
app.use(express.json());

app.get('/api/health', (req, res) => res.json({ status:
'ok' }));
app.listen(3000, () => console.log('Listening on
:3000'));
```

## 1.6.2 HTML: ключова фронтенд-технологія

Якщо ми порівнювали веб-додаток з рестораном, то HTML (від англ. HyperText Markup Language) — це його архітектурний план, креслення, що визначає, де будуть стіни, вікна та двері. Простими словами, HTML — це основна мова розмітки, яка відповідає за структуру будь-якої веб-сторінки. Вона не говорить, як щось має виглядати, а лише визначає, чим воно є: це заголовок, це абзац, це посилання, а це — зображення.

HTML розшифровується як HyperText Markup Language, що в перекладі означає мова гіпертекстової розмітки. Слово "гіпертекст" тут ключове – це текст, який містить посилання на інші документи, створюючи ту саму павутину зв'язків, завдяки якій World Wide Web отримав свою назву. А "розмітка" означає, що ми не просто пишемо звичайний текст, а структуруємо його, вказуючи браузеру, які частини є заголовками, які – абзацами, де розміщені зображення, а де – інтерактивні елементи.

Ключова особливість HTML полягає в тому, що це декларативна мова розмітки, а не мова програмування в класичному розумінні. Це означає, що ми описуємо, що має бути на сторінці та як це структуровано, але не пишемо

алгоритми чи логіку. HTML складається з елементів, кожен з яких позначається спеціальними мітками, які називаються тегами. Теги зазвичай йдуть парами – відкриваючий тег вказує початок елемента, а закриваючий тег, який містить слеш, позначає його кінець.

Кожен HTML-документ починається з кореневого елемента `<html>`, який, подібно до фундаменту, охоплює все інше. Усередині цього фундаменту є дві ключові секції. Перша — це `<head>`, яка є своєрідним "заштунками" сторінки. У цьому розділі ми не бачимо нічого на екрані, але тут міститься важлива службова інформація: метадані, які розповідають пошуковим системам, про що сторінка, а також підключення зовнішніх файлів, як-от стилі CSS для оформлення чи скрипти JavaScript для інтерактивності. Друга секція — це `<body>`, і це єдине місце, де розташовується весь видимий вміст сторінки: тексти, картинки, кнопки, форми, таблиці. Усе, що користувач може побачити та з чим може взаємодіяти, знаходиться саме тут.

Для створення структури сторінки в HTML використовуються так звані теги. Кожен тег має своє призначення. Наприклад, щоб створити заголовок, ми використовуємо теги від `<h1>` до `<h6>`, де `h1` позначає найважливіший заголовок, а `h6` — найменший. Для звичайного тексту використовується тег `<p>`, що створює абзац. Коли ми хочемо додати посилання, щоб перейти на іншу сторінку, ми застосовуємо тег `<a>`. Для вставлення зображення ми використовуємо тег `<img>`. А для створення інтерактивних елементів, як-от поля для введення тексту чи кнопки, ми застосовуємо теги `<form>`, `<input>` та `<button>`.

Отже, HTML є тим самим "каркасом", який надає скелет веб-сторінці. Без нього не було б, що прикрашати стилями та оживляти скриптами. Це — фундамент, на якому будується весь сучасний веб-інтерфейс.

Семантичність — одна з найважливіших концепцій сучасного HTML. Це означає, що ми вибираємо HTML-елементи не лише на основі того, як вони виглядають, а виходячи з їх значення та призначення в контексті документа. Наприклад, для основного заголовка сторінки використовується елемент `h1`,

для підзаголовків – h2, h3 та так далі, для звичайних абзаців – елемент p, для списків – ul або ol залежно від того, чи потрібна нумерація. Такий підхід робить веб-сторінки більш доступними для людей з обмеженими можливостями, які використовують екранні читачі, а також покращує розуміння структури сторінки пошуковими системами.

HTML5 ввів нові семантичні елементи, які дозволяють ще точніше описувати структуру веб-сторінок. Елемент header зазвичай містить логотип, навігаційне меню та іншу інформацію, яка розміщується у верхній частині сторінки. Елемент nav призначений для навігаційних меню. Main позначає основний вміст сторінки, article використовується для самостійних блоків контенту, як-от статті блогу чи новини, section допомагає розділити вміст на тематичні секції, aside підходить для бічної панелі з додатковою інформацією, а footer традиційно розміщується внизу сторінки та містить контактну інформацію, копірайт та інші допоміжні дані.

Атрибути додають HTML-елементам додаткову функціональність та можливості налаштування. Кожен елемент може мати один або кілька атрибутів, які надають браузеру додаткову інформацію про те, як обробляти цей елемент. Наприклад, атрибут src у елемента img вказує шлях до файлу зображення, атрибут href у елемента a визначає адресу, на яку веде посилання, а атрибут alt надає альтернативний опис зображення для випадків, коли воно не може бути завантажено або для екранних читачів.

Глобальні атрибути можуть використовуватися з будь-яким HTML-елементом. Атрибут id надає елементу унікальний ідентифікатор, який може використовуватися для посилань, стилізації через CSS або маніпуляцій через JavaScript. Атрибут class дозволяє присвоїти елементу одну або кілька класів, що особливо корисно для групування елементів з подібними характеристиками та застосування до них спільних стилів. Атрибут title надає додаткову інформацію про елемент, яка зазвичай відображається у вигляді спливаючої підказки при наведенні курсора.

Форми в HTML заслуговують особливої уваги, оскільки вони забезпечують інтерактивність та дозволяють користувачам надсилати дані на сервер. Елемент `form` створює контейнер для різних полів вводу, кнопок та інших інтерактивних елементів. Усередині форми можуть розміщуватися текстові поля, поля для паролів, радіо-кнопки, чекбокси, спадні списки, текстові області та багато інших типів елементів вводу. Кожне поле форми має ім'я, яке використовується для ідентифікації даних при їх відправці на сервер.

HTML5 значно розширив можливості роботи з формами, додавши нові типи полів вводу, такі як `email`, `url`, `tel`, `date`, `number` та багато інших. Ці спеціалізовані поля не лише надають браузеру інформацію про тип очікуваних даних, але й автоматично забезпечують базову валідацію введеної інформації. Наприклад, поле з типом `email` автоматично перевіряє, чи містить введений текст символ `@` та має правильний формат електронної адреси.

Доступність веб-сторінок – це критично важливий аспект сучасної веб-розробки, і HTML надає багато інструментів для створення доступного контенту. Правильне використання семантичних елементів автоматично покращує доступність, але існують і спеціальні атрибути, такі як `aria-label`, `aria-describedby` та `role`, які надають додаткову інформацію для допоміжних технологій. Альтернативні описи для зображень через атрибут `alt`, правильна структура заголовків та логічний порядок табуляції також критично важливі для користувачів з обмеженими можливостями.

HTML як ключова фронтенд-технологія працює в тісній інтеграції з CSS та JavaScript. Хоча HTML визначає структуру та зміст, CSS відповідає за візуальне представлення, а JavaScript додає інтерактивність та динамічну поведінку. Ця тріада технологій утворює основу сучасної веб-розробки, і глибоке розуміння HTML є необхідною передумовою для ефективної роботи з іншими фронтенд-технологіями.

Сучасні тенденції в HTML-розробці включають активне використання прогресивних веб-технологій, семантичної розмітки для покращення SEO, адаптивного дизайну для різних пристроїв та дотримання стандартів веб-

доступності. HTML продовжує розвиватися, і розуміння його принципів та можливостей залишається фундаментальним навиком для будь-якого веб-розробника.

Базовий шаблон зі службовими метаданими:

```
<!doctype html>
<html lang="uk">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width,
initial-scale=1" />
    <title>Привіт, світ!</title>
  </head>
  <body>
    <h1>Привіт, світ!</h1>
    <p>Мій перший базовий документ HTML.</p>
  </body>
</html>
```

### 1.6.3 CSS: стилізація

CSS (Cascading Style Sheets) — технологія, яка визначає вигляд елементів, створених за допомогою HTML. Дозволяє налаштовувати кольори, шрифти, відступи, анімації, макети. CSS створює «зовнішній одяг» для веб-сторінки, роблячи її привабливою та зручною.

Якщо HTML можна порівняти зі скелетом будівлі, то CSS – це архітектурний дизайн, який визначає, як ця будівля виглядатиме зсередини та ззовні.

CSS розшифровується як Cascading Style Sheets, що означає каскадні таблиці стилів. Слово "каскадні" тут має особливе значення – воно вказує на те, що стилі можуть наслідуватися та перекриватися згідно з певними правилами пріоритетності. Цей механізм дозволяє створювати складні та

гнучкі системи стилізації, де загальні стилі можуть застосовуватися до великих груп елементів, а специфічні стилі можуть перекривати їх для окремих випадків.

Основою роботи з CSS є концепція селекторів та правил. Селектор вказує браузеру, до яких саме HTML-елементів має застосовуватися певний стиль, а правило описує, які саме властивості та їх значення мають бути застосовані. Найпростіші селектори посиляються на HTML-теги безпосередньо, але CSS надає набагато більше можливостей для точного таргетування елементів. Селектори класів дозволяють застосовувати стилі до груп елементів з однаковим атрибутом `class`, селектори ідентифікаторів націлюються на унікальні елементи з певним `id`, а складні селектори можуть комбінувати різні критерії для досягнення дуже специфічного таргетування.

Каскадність CSS означає, що коли кілька правил застосовуються до одного елемента, браузер використовує систему пріоритетів для визначення, який стиль буде остаточно застосований. Інлайн-стилі, записані безпосередньо в HTML-атрибуті `style`, мають найвищий пріоритет, за ними йдуть ідентифікатори, потім класи, а найнижчий пріоритет мають селектори тегів. Цей механізм дозволяє розробникам створювати загальні стилі для всього сайту та перекривати їх більш специфічними правилами для окремих сторінок чи елементів.

Блочна модель CSS є фундаментальною концепцією, яка визначає, як браузер обчислює розміри та розміщення елементів на сторінці. Кожен елемент складається з кількох шарів: самого вмісту, внутрішніх відступів (`padding`), рамки (`border`) та зовнішніх відступів (`margin`). Розуміння того, як ці шари взаємодіють між собою, критично важливе для створення точних та передбачуваних макетів. Властивість `box-sizing` дозволяє змінювати спосіб обчислення загального розміру елемента, що особливо корисно при створенні адаптивних дизайнів.

Позиціонування елементів в CSS надає розробникам потужні інструменти для контролю над розміщенням контенту на сторінці. Статичне



позиціонування є стандартним і розміщує елементи в природному потоці документа. Відносне позиціонування дозволяє зсувати елементи відносно їх природної позиції, не впливаючи на інші елементи. Абсолютне позиціонування виймає елемент з природного потоку і розміщує його відносно найближчого позиціонованого батьківського елемента. Фіксоване позиціонування прикріплює елемент до вікна браузера, що робить його видимим навіть при прокручуванні сторінки.

Flexbox революціонізував спосіб створення макетів в CSS, надавши інтуїтивний та потужний механізм для розподілу простору між елементами. Гнучкий контейнер може автоматично розподіляти доступний простір між дочірніми елементами, вирівнювати їх по горизонталі та вертикалі, змінювати порядок відображення без зміни HTML-структури. Flexbox особливо ефективний для створення навігаційних меню, карток контенту та інших компонентів, які мають адаптуватися до різних розмірів екрана.

CSS Grid представляє ще більш потужну систему макетування, яка дозволяє створювати двовимірні сітки для розміщення контенту. На відміну від Flexbox, який працює в одному напрямку, Grid може одночасно контролювати як рядки, так і колонки. Ця технологія ідеально підходить для створення складних макетів веб-сторінок, де різні елементи мають займати певні позиції в загальній сітці. Grid дозволяє створювати адаптивні макети, які автоматично перестроюються залежно від розміру екрана.

Адаптивний дизайн став обов'язковим стандартом сучасної веб-розробки, оскільки користувачі заходять на веб-сайти з найрізноманітніших пристроїв – від смартфонів до широкоформатних моніторів. CSS надає інструменти для створення дизайнів, які елегантно адаптуються до різних розмірів екрана та можливостей пристроїв. Медіа-запити дозволяють застосовувати різні стилі залежно від характеристик пристрою, таких як ширина екрана, роздільна здатність або орієнтація. Гнучкі одиниці виміру, такі як відсотки, viewport units та em, дозволяють створювати макети, які масштабуються пропорційно.

Типографіка в CSS охоплює всі аспекти роботи з текстом – від вибору шрифтів до налаштування міжрядкових відстаней. Сучасні браузері підтримують веб-шрифти, що дозволяє використовувати будь-які шрифти, не покладаючись на те, які шрифти встановлені на комп'ютері користувача. Властивості для налаштування розміру тексту, товщини, стилю, кольору та інших характеристик надають повний контроль над типографічним оформленням. Правильна типографіка значно впливає на читабельність контенту та загальне сприйняття сайту користувачами.

Кольори та градієнти в CSS відкривають безмежні можливості для візуального оформлення. Кольори можна задавати різними способами – через назви, шістнадцяткові коди, RGB, HSL значення, кожен з яких має свої переваги в різних ситуаціях. Градієнти дозволяють створювати плавні переходи між кольорами, що додає глибини та сучасності дизайну. CSS підтримує як лінійні, так і радіальні градієнти, а також можливість створення складних багатокольорових переходів.

Анімації та переходи надають веб-сторінкам динамічності та інтерактивності. CSS-переходи дозволяють створювати плавні зміни властивостей елементів при взаємодії з користувачем, наприклад, при наведенні курсора або фокусі. Ключові кадри дають можливість створювати складні анімації, які можуть тривати певний час, повторюватися та мати різні функції часу для створення реалістичних ефектів руху.

Препроцесори CSS, такі як Sass або Less, розширюють можливості звичайного CSS, додаючи функціональність програмування – змінні, функції, вкладеність селекторів та інші корисні можливості. Ці інструменти дозволяють писати більш організований та підтримуваний код, особливо в великих проектах. Змінні в препроцесорах дозволяють визначити кольори, розміри та інші значення в одному місці та використовувати їх по всьому проекту, що значно спрощує підтримку та зміну дизайну.

Методології організації CSS, такі як BEM (Block Element Modifier), допомагають структурувати код та робити його більш зрозумілим та

підтримуваним. Ці підходи особливо важливі в командній розробці, де кілька розробників працюють над одним проектом. Правильна організація CSS-коду запобігає конфліктам стилів та робить код більш передбачуваним.

Продуктивність CSS також є важливим аспектом, особливо для великих веб-сайтів. Оптимізація селекторів, мінімізація коду, використання ефективних властивостей та уникнення надмірно складних стилів допомагають покращити швидкість завантаження та відображення сторінок. Сучасні інструменти збірки можуть автоматично оптимізувати CSS-код, видаляючи неиспользуемые стилі та стискаючи файли для зменшення їх розміру.

CSS як технологія стилізації є невід'ємною частиною сучасної веб-розробки, і її можливості продовжують розширюватися з кожною новою версією стандарту. Глибоке розуміння CSS дозволяє створювати не лише красиві, але й функціональні, доступні та продуктивні веб-інтерфейси, які забезпечують відмінний користувацький досвід на всіх типах пристроїв.

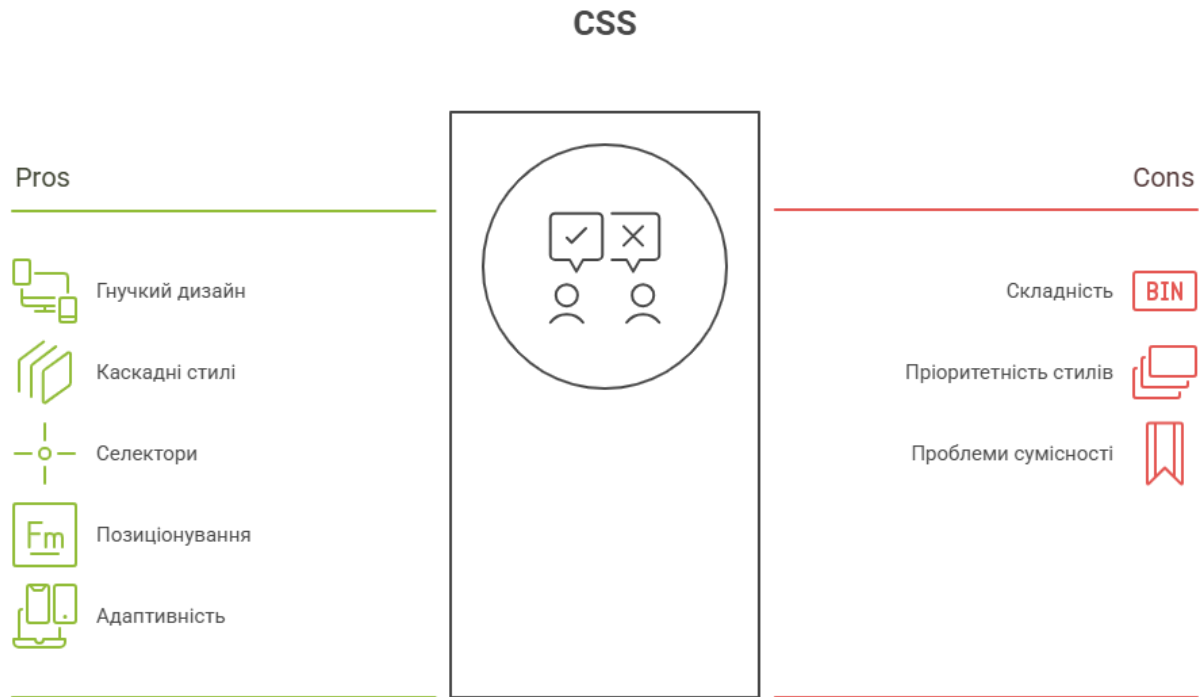


Рисунок 1.9 — Сильні та слабкі сторони CSS

### 1.6.4 DOM: структура веб-сторінки

DOM (Document Object Model) — це стандартна модель представлення структури веб-сторінки, яка дозволяє браузеру і програмам (зокрема JavaScript) працювати з HTML або XML документом, змінювати його вміст, стиль і навіть структуру в реальному часі.

Коли браузер завантажує HTML-документ, він не просто читає його як текст. Він перетворює його в об'єктну структуру — дерево вузлів (nodes). Кожен тег HTML стає елементом (element), текст — текстовим вузлом (text node), а атрибути — частиною вузлів.

DOM є інтерфейсом між програмістом і веб-сторінкою. Завдяки йому JavaScript може:

- знаходити елементи (`document.getElementById("header");`);
- змінювати їх вміст (`element.textContent = "Новий текст";`);
- додавати або видаляти елементи (`element.remove()` або `document.createElement()`).

Розглянемо просту веб-сторінку:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Приклад DOM</title>
  </head>
  <body>
    <h1>Заголовок</h1>
    <p>Це абзац тексту.</p>
  </body>
</html>
```

Після завантаження браузером, DOM буде виглядати так (у вигляді дерева):

```
Document
```

```

└─ html
    └─ head
        └─ title
            └─ "Приклад DOM"
    └─ body
        └─ h1
            └─ "Заголовок"
        └─ p
            └─ "Це абзац тексту."

```

Основні компоненти DOM:

1. Document — корінь дерева, що представляє весь документ.
2. Element — HTML-теги (<html>, <body>, <p> і т.д.).
3. Text nodes — текст всередині тегів.
4. Attributes — властивості елементів (наприклад, class="main").
5. Comments — коментарі в HTML також є частиною DOM.

Чому DOM важливий? Це модель даних, яка дозволяє змінювати сторінку після її завантаження (динамічні сайти). Це стандарт, підтримуваний усіма браузерами. Він робить можливими JavaScript-фреймворки (React, Vue, Angular), які постійно маніпулюють DOM.

### 1.6.5 JavaScript

Розглянемо, як JavaScript працює з DOM на практиці. Це один із ключових моментів динамічного веб-розроблення.

#### 1. Доступ до елементів сторінки

JavaScript може знайти елементи на сторінці різними способами:

```

<body>
  <h1 id="title">Заголовок</h1>
  <p class="text">Це абзац тексту.</p>
  <button>Натисни мене</button>
</body>

```

### JS приклади:

```
// По ID
const heading = document.getElementById("title");

// По класу (повертає колекцію)
const paragraphs =
document.getElementsByClassName("text");

// По тегу (повертає колекцію)
const buttons =
document.getElementsByTagName("button");

// Нові методи з querySelector
const firstParagraph = document.querySelector(".text");
// перший елемент
const allParagraphs =
document.querySelectorAll(".text"); // всі елементи
```

### 2. Зміна вмісту та атрибутів

```
// Зміна тексту
heading.textContent = "Новий заголовок";

// Зміна HTML
firstParagraph.innerHTML = "Цей абзац
<strong>виділений</strong>";

// Зміна стилю
heading.style.color = "blue";
heading.style.fontSize = "30px";
```

```
// Робота з атрибутами
heading.setAttribute("data-type", "main");
console.log(heading.getAttribute("data-type")); //
"main"
```

### 3. Додавання і видалення елементів

```
// Створити новий елемент
const newPara = document.createElement("p");
newPara.textContent = "Цей абзац додано за допомогою
JS.";

// Додати його в body
document.body.appendChild(newPara);

// Видалити елемент
newPara.remove();
```

### 4. Реакція на події

```
const button = document.querySelector("button");

button.addEventListener("click", () => {
  heading.textContent = "Ви натиснули кнопку!";
  heading.style.color = "red";
});
```

Тепер при натисканні на кнопку заголовок змінюється і стає червоним.

## 1.6.6 Бекенд: програмування серверної частини

Бекенд - це серверна частина веб-додатку, яка працює на сервері і відповідає за обробку бізнес-логіки, взаємодію з базами даних, автентифікацію користувачів, безпеку та надання даних фронтенду через API. Якщо фронтенд - це те, що бачить користувач (інтерфейс, кнопки, форми), то бекенд - це

"мозок" додатку, який приймає рішення, обробляє запити і керує всіма внутрішніми процесами.

Уявіть собі ресторан: фронтенд - це зал, де сидять відвідувачі і офіціанти приймають замовлення, а бекенд - це кухня, де кухарі готують страви, перевіряють наявність інгредієнтів на складі і координують весь процес приготування їжі.

Сучасна бекенд-архітектура зазвичай побудована за принципом багат шарової архітектури, де кожен шар має свою відповідальність і взаємодіє з сусідніми шарами через чітко визначені інтерфейси.

### Багат шарова архітектура бекенду



Рисунок 1.10 — Багат шарова архітектура бекенду

Шар представлення (Presentation Layer) - це REST API або GraphQL ендпоінти, які отримують HTTP-запити від клієнтів і повертають відповіді у форматі JSON, XML або інших структурованих форматах. Цей шар відповідає за валідацію вхідних даних, автентифікацію та авторизацію запитів.

Шар бізнес-логіки (Business Logic Layer) - тут зосереджена вся основна логіка додатку: обчислення, правила валідації даних, алгоритми обробки інформації та координація між різними компонентами системи. Цей шар не залежить від способу отримання даних чи їх зберігання.



Шар доступу до даних (Data Access Layer) - відповідає за взаємодію з базами даних, файловими системами, зовнішніми API та іншими джерелами даних, інкапсулюючи всі деталі роботи з конкретними технологіями зберігання.

Ось приклад простої архітектури на Node.js з Express:

```
// Шар представлення - API роути
const express = require('express');
const app = express();

app.get('/api/users/:id', async (req, res) => {
  try {
    const user = await
userService.getUserById(req.params.id);
    res.json(user);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

// Шар бізнес-логіки - сервіс
class UserService {
  async getUserById(id) {
    const user = await userRepository.findById(id);
    if (!user) throw new Error('User not found');
    return this.transformUserData(user);
  }

  transformUserData(user) {
    return {
      id: user.id,
```

```

        name: user.full_name,
        email: user.email_address
    };
}
}

// Шар доступу до даних - репозиторій
class UserRepository {
    async findById(id) {
        const query = 'SELECT * FROM users WHERE id =
?';

        return await database.query(query, [id]);
    }
}

```

Коли користувач натискає кнопку в браузері або мобільному додатку, розпочинається складний процес обробки запиту, який проходить через кілька етапів, кожен з яких має критичне значення для правильного функціонування всієї системи.

Етап 1: Отримання запиту - веб-сервер (наприклад, Nginx або Apache) отримує HTTP-запит від клієнта і передає його до додатку через протоколи FastCGI, WSGI або прямо через HTTP у випадку Node.js чи Go.

Етап 2: Маршрутизація - роутер аналізує URL запиту і HTTP-метод, щоб визначити, який контролер і яка функція повинні обробити цей запит, часто використовуючи патерни URL та регулярні вирази.

Етап 3: Middleware обробка - запит проходить через ланцюжок middleware функцій, які можуть виконувати автентифікацію, логування, валідацію, обмеження швидкості запитів та інші допоміжні операції.

Етап 4: Контролер обробка - основна логіка обробки запиту виконується в контролері, який координує взаємодію між різними сервісами і компонентами.

Етап 5: Бізнес-логіка - викликаються відповідні сервіси, які містять основну логіку додатку, обчислення та правила валідації.

Етап 6: Доступ до даних - якщо потрібно, виконуються запити до бази даних, зовнішніх API або файлової системи.

Етап 7: Формування відповіді - результат обробки серіалізується в JSON або інший формат і повертається клієнту з відповідним HTTP статус-кодом.

## Послідовність обробки запиту

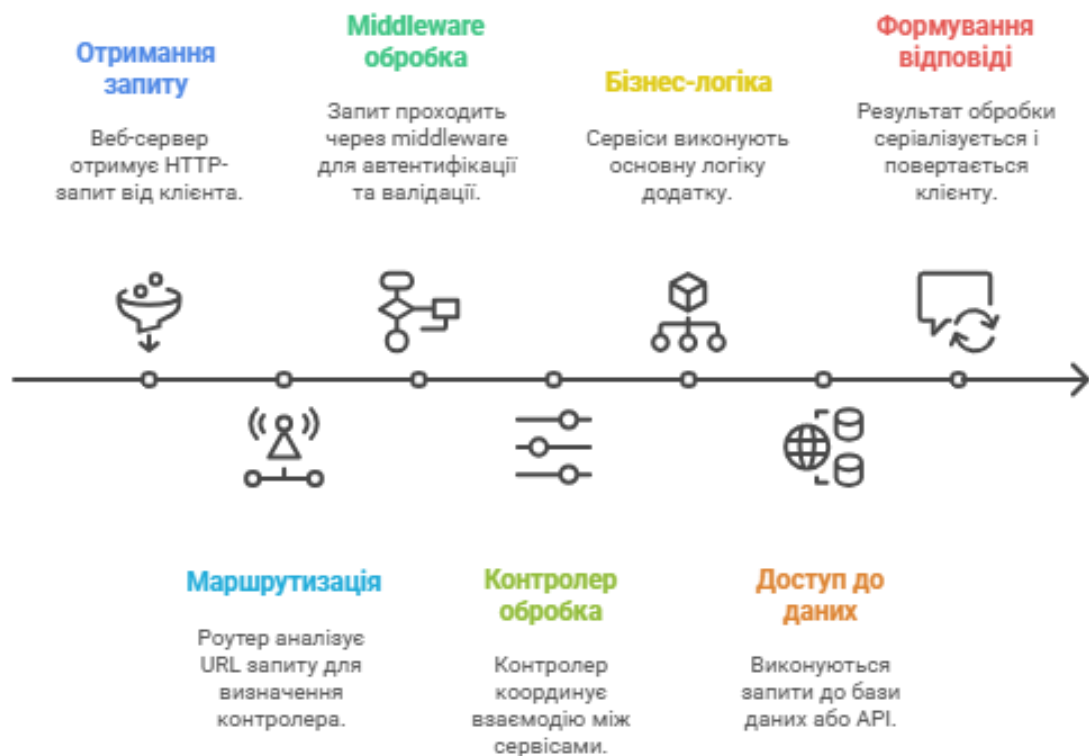


Рисунок 1.11 — Послідовність обробки запиту

Приклад повного циклу на Python з Flask:

```
from flask import Flask, request, jsonify
import sqlite3
import bcrypt

app = Flask(__name__)

# Middleware для логування
@app.before_request
```

```

def log_request():
    print(f"Request: {request.method} {request.url}")

# Маршрутизація і контролер
@app.route('/api/auth/login', methods=['POST'])
def login():
    data = request.get_json()

    # Валідація вхідних даних
    if not data.get('email') or not
data.get('password'):
        return jsonify({'error': 'Email and password
required'}), 400

    # Бізнес-логіка автентифікації
    user = authenticate_user(data['email'],
data['password'])
    if user:
        token = generate_token(user['id'])
        return jsonify({'token': token, 'user': user})
    else:
        return jsonify({'error': 'Invalid
credentials'}), 401

# Бізнес-логіка
def authenticate_user(email, password):
    conn = sqlite3.connect('app.db')
    cursor = conn.execute('SELECT * FROM users WHERE
email = ?', (email,))
    user = cursor.fetchone()

```

```

conn.close()

if user and bcrypt.checkpw(password.encode(),
user[3]):
    return {'id': user[0], 'email': user[1],
'name': user[2]}
    return None

def generate_token(user_id):
    # Спрощена генерація токена
    import uuid
    return str(uuid.uuid4())

```

База даних і системи зберігання - серце будь-якого бекенду, де зберігається вся інформація додатку. Реляційні бази даних (PostgreSQL, MySQL) використовуються для структурованих даних з складними зв'язками, тоді як NoSQL бази (MongoDB, Redis) краще підходять для неструктурованих даних і високонавантажених додатків.

API і веб-сервіси - інтерфейси, через які фронтенд та інші системи взаємодіють з бекендом, найчастіше реалізовані як RESTful API з використанням JSON для обміну даними, або більш сучасні GraphQL API для складних запитів.

Автентифікація і авторизація - системи безпеки, які перевіряють ідентичність користувачів (хто ви?) і контролюють доступ до ресурсів (що вам дозволено робити?), часто реалізовані через JWT токени, OAuth2 або сесії.

Кешування - механізми зберігання часто запитуваних даних у швидкій пам'яті (Redis, Memcached) для підвищення продуктивності і зменшення навантаження на базу даних.

Приклад інтеграції всіх компонентів на Node.js:

```

const express = require('express');
const jwt = require('jsonwebtoken');

```

```

const redis = require('redis');
const mysql = require('mysql2/promise');

const app = express();
const cache = redis.createClient();
const db = mysql.createPool({
  host: 'localhost',
  user: 'root',
  database: 'myapp'
});

// Middleware автентифікації
const authenticateToken = (req, res, next) => {
  const token = req.headers['authorization']?.split('
')[1];
  if (!token) return res.sendStatus(401);

  jwt.verify(token, process.env.JWT_SECRET, (err,
user) => {
    if (err) return res.sendStatus(403);
    req.user = user;
    next();
  });
};

// API з кешуванням
app.get('/api/posts', authenticateToken, async (req,
res) => {
  const cacheKey = `posts:user:${req.user.id}`;

```

```

// Перевіряємо кеш
const cachedPosts = await cache.get(cacheKey);
if (cachedPosts) {
    return res.json(JSON.parse(cachedPosts));
}

// Запит до БД
const [posts] = await db.execute(
    'SELECT * FROM posts WHERE user_id = ? ORDER BY
created_at DESC',
    [req.user.id]
);

// Зберігаємо в кеш на 5 хвилин
await cache.setex(cacheKey, 300,
JSON.stringify(posts));
res.json(posts);
});

// CRUD операції
app.post('/api/posts', authenticateToken, async (req,
res) => {
    const { title, content } = req.body;

    try {
        const [result] = await db.execute(
            'INSERT INTO posts (user_id, title,
content, created_at) VALUES (?, ?, ?, NOW())',
            [req.user.id, title, content]
        );
    }

```

```

        // Очищуємо кеш після створення нового поста
        await cache.del(`posts:user:${req.user.id}`);

        res.status(201).json({ id: result.insertId,
message: 'Post created' });
    } catch (error) {
        res.status(500).json({ error: 'Failed to create
post' });
    }
});

```

У реальних проектах бекенд-розробник повинен постійно думати про масштабованість, безпеку, продуктивність та підтримуваність коду, що вимагає дотримання певних принципів і патернів проектування.

Принцип єдиної відповідальності означає, що кожен клас або функція повинні мати лише одну причину для змін, що робить код більш модульним і легким для тестування та підтримки.

Обробка помилок і логування - критично важливі аспекти, які дозволяють швидко діагностувати проблеми в продакшені і забезпечувати надійну роботу системи навіть у випадку несподіваних ситуацій.

Тестування - автоматизовані тести (unit, integration, end-to-end) забезпечують впевненість у тому, що код працює правильно і не ламається при внесенні змін.

Приклад комплексного сервісу з усіма практиками:

```

import logging
import pytest
from typing import Optional
from dataclasses import dataclass

# Логування

```



```

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Модель даних
@dataclass
class User:
    id: int
    email: str
    name: str

# Винятки
class UserNotFoundError(Exception):
    pass

class UserService:
    def __init__(self, repository):
        self.repository = repository

    async def get_user(self, user_id: int) -> User:
        """Отримати користувача за ID з логуванням і
        обробкою помилок"""
        try:
            logger.info(f"Fetching user {user_id}")
            user = await
self.repository.find_by_id(user_id)
            if not user:
                raise UserNotFoundError(f"User
{user_id} not found")
            logger.info(f"Successfully fetched user
{user_id}")

```

```

        return user
    except Exception as e:
        logger.error(f"Error fetching user
{user_id}: {str(e)}")
        raise

    async def create_user(self, email: str, name: str)
-> User:
        """Створити нового користувача з валідацією"""
        if not self._validate_email(email):
            raise ValueError("Invalid email format")

        try:
            user_id = await self.repository.create({
                'email': email,
                'name': name
            })
            logger.info(f"Created user {user_id} with
email {email}")
            return User(id=user_id, email=email,
name=name)
        except Exception as e:
            logger.error(f"Error creating user:
{str(e)}")
            raise

    def _validate_email(self, email: str) -> bool:
        import re
        pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-
]+\.[a-zA-Z]{2,}$'
```

```

        return re.match(pattern, email) is not None

# Тестування
class TestUserService:
    @pytest.fixture
    def mock_repository(self):
        class MockRepository:
            async def find_by_id(self, user_id):
                if user_id == 1:
                    return User(id=1,
email="test@example.com", name="Test User")
                return None

            async def create(self, data):
                return 2
        return MockRepository()

    @pytest.fixture
    def user_service(self, mock_repository):
        return UserService(mock_repository)

    async def test_get_existing_user(self,
user_service):
        user = await user_service.get_user(1)
        assert user.id == 1
        assert user.email == "test@example.com"

    async def test_get_nonexistent_user(self,
user_service):
        with pytest.raises(UserNotFoundError):

```

```

        await user_service.get_user(999)

    async def test_create_user_valid_data(self,
user_service):
        user = await
user_service.create_user("new@example.com", "New User")
        assert user.id == 2
        assert user.email == "new@example.com"

```

Бекенд-розробка - це мистецтво балансування між функціональністю, продуктивністю і підтримуваністю коду, де кожне рішення впливає на досвід мільйонів користувачів, які навіть не здогадуються про складність процесів, що відбуваються на сервері щоразу, коли вони натискають кнопку або відправляють форму в інтернеті.

## Висновки

У першому розділі систематизовано знання про еволюцію веб-технологій від Web 1.0 до Web 4.0, проаналізовано фундаментальні принципи роботи Інтернету та розглянуто ключові технології розробки сучасних веб-додатків.

Дослідження стеку протоколів TCP/IP підтвердило його фундаментальну роль у забезпеченні універсальності та масштабованості Інтернету. Виявлено, що принципи універсальності та віртуальної єдності залишаються актуальними навіть при переході до IPv6, який вирішує проблеми обмеженості адресного простору та забезпечує вбудовану підтримку безпеки.

Представлений матеріал демонструє, що успішна веб-розробка потребує не лише знання конкретних технологій, але й розуміння принципів їх взаємодії, архітектурних патернів та тенденцій розвитку індустрії.

Отримані знання створюють необхідний фундамент для подальшого вивчення спеціалізованих аспектів веб-розробки та формують системний

підхід до створення сучасних веб-рішень, які відповідають вимогам масштабованості, безпеки та користувацького досвіду.

### **Питання для самоперевірки**

- 1) Охарактеризуйте основні принципи роботи Web 1.0. Чому цей період називають "мережею доставки контенту"?
- 2) Які технічні обмеження Web 1.0 перешкоджали розвитку інтерактивних веб-додатків?
- 3) Поясніть, як зміна моделі взаємодії з "один до багатьох" на "багато до багатьох" вплинула на розвиток Web 2.0.
- 4) Назвіть три ключові технологічні інновації Web 2.0 та поясніть їх вплив на користувацький досвід.
- 5) Що означає поняття "семантичний веб" у контексті Web 3.0? Наведіть приклади практичного застосування.
- 6) Порівняйте переваги та недоліки кожного покоління веб-технологій. Які проблеми вирішувало кожне нове покоління?
- 7) Сформулюйте та поясніть два основні принципи Інтернету: універсальність та віртуальну єдність.
- 8) Опишіть функції кожного рівня моделі TCP/IP. Як вони взаємодіють між собою?
- 9) У чому полягає різниця між протоколами TCP та UDP? Коли доцільно використовувати кожен з них?
- 10) Поясніть поняття "безстановий протокол" на прикладі HTTP. Як це впливає на архітектуру веб-додатків?
- 11) Визначте ключові відмінності між веб-сайтами та веб-додатками за критеріями функціональності, інтерактивності та технічної складності.
- 12) Поясніть різницю між статичним та динамічним контентом. Як це впливає на архітектуру веб-ресурсу?
- 13) Які переваги та недоліки мають статичні та динамічні веб-додатки?

- 14) Опишіть роль веб-сервера в архітектурі веб-додатків. Які основні функції він виконує?
- 15) Поясніть процес обробки HTTP-запиту від клієнта до сервера та назад.
- 16) Які основні функції виконує браузер при роботі з веб-контентом?
- 17) Поясніть різницю між різними браузерними рушіями та їх вплив на відображення веб-сторінок.
- 18) Опишіть роль HTML у створенні веб-сторінок. Чому важлива семантична розмітка?
- 19) Поясніть принцип каскадності в CSS. Як визначається пріоритетність стилів?
- 20) Що таке DOM і як JavaScript взаємодіє з ним для створення динамічних веб-сторінок?
- 21) Порівняйте різні способи позиціонування елементів у CSS (Flexbox, Grid, традиційне позиціонування).
- 22) Порівняйте JSON та XML як формати обміну даними. Які переваги та недоліки має кожен?
- 23) Опишіть багат шарову архітектуру бекенду. Яка відповідальність кожного шару?
- 24) Поясніть цикл обробки HTTP-запиту в бекенд-додатку від маршрутизації до формування відповіді.
- 25) Які принципи безпеки мають враховуватися при розробці бекенд-систем?

## РОЗДІЛ 2. ОСНОВИ HTML

HyperText Markup Language (HTML) є основою Всесвітньої павутини та фундаментальною технологією для створення веб-сторінок. Ця мова розмітки дозволяє структурувати та організовувати контент у веб-документах, визначаючи семантичне значення кожного елемента сторінки. HTML використовує систему тегів для опису різних частин документа, від заголовків та абзаців до зображень та посилань.

HTML не є мовою програмування у класичному розумінні, а радше мовою розмітки, яка інструктує веб-браузер про те, як відобразити контент користувачеві. Кожен HTML-документ представляє собою текстовий файл з розширенням .html або .htm, який містить теги, атрибути та контент, що разом формують структуру веб-сторінки.

HTML слугує своєрідним каркасом, на який потім накладаються стилі (CSS) і логіка (JavaScript). Коли браузер отримує HTML-документ від сервера, він послідовно читає його від початку до кінця, аналізуючи кожен тег і створюючи структуру, яка називається DOM (Document Object Model). Саме ця модель стає основою для подальшого відображення сторінки на екрані.

Основою HTML є теги. Вони описують, які елементи повинні бути на сторінці: заголовки, абзаци, зображення, посилання, списки тощо. Кожен тег має свій початок і кінець, а також може містити атрибути — додаткову інформацію для браузера. Наприклад, атрибут src вказує шлях до зображення, а href — адресу посилання.

Найпростіший HTML-документ складається зі службових тегів <html>, <head> і <body>. У <head> зберігається службова інформація — назва сторінки, метадані, підключення стилів і скриптів. У <body> розташовуються всі видимі для користувача елементи.

Ось приклад найпростішої веб-сторінки:

```
<!DOCTYPE html>
<html lang="uk">
```

```
<head>
  <meta charset="UTF-8">
  <title>Моя перша сторінка</title>
</head>
<body>
  <h1>Вітаю у світі HTML!</h1>
  <p>Це мій перший абзац.</p>
</body>
</html>
```

Коли браузер відкриває цей документ, він відображає великий заголовок і текст під ним. Усе інше, що знаходиться у коді, впливає на структуру та поведінку сторінки, навіть якщо не видно на екрані.

## 2.1 Базові поняття

### 2.1.1 Одинарні теги HTML

Одинарні теги в HTML представляють собою особливу категорію елементів, які кардинально відрізняються від звичайних парних тегів своєю структурою та призначенням. На відміну від більшості HTML-елементів, які складаються з відкриваючого та закриваючого тегів з контентом між ними, одинарні теги є самодостатніми елементами, які не потребують закриття та не можуть містити будь-який контент всередині себе.

Ці елементи також називають порожніми або самозакриваючими тегами, оскільки вони виконують свою функцію одразу після появи в коді, без необхідності в додатковому контенті чи закриваючому тезі. Їхнє призначення полягає у вставці специфічних елементів, створенні розривів або додаванні службової інформації до HTML-документа.

Тег `<br>` є одним з найчастіше використовуваних одинарних елементів і служить для створення розриву рядка в тексті. Коли браузер зустрічає цей тег, він переносить наступний текст на новий рядок, але при цьому не створює



нового абзацу з відповідними відступами. Це робить `<br>` незамінним інструментом для форматування тексту, де потрібен точний контроль над розташуванням рядків, наприклад, при відображенні віршів, де кожен рядок повинен починатися з нової позиції, або при форматуванні поштових адрес, де кожна частина адреси традиційно розміщується на окремому рядку.

Горизонтальна лінія `<hr>` створює візуальний розділювач між різними секціями контенту на веб-сторінці. Цей елемент автоматично малює лінію поперек всієї доступної ширини свого батьківського контейнера і традиційно використовується для тематичного поділу контенту, коли потрібно чітко відокремити один розділ від іншого. Наприклад, в статті `<hr>` може розділяти різні підтеми, а в блозі - відокремлювати основний текст від коментарів або додаткової інформації.

Тег `<img>` є критично важливим елементом для вставки зображень у веб-документи і обов'язково повинен супроводжуватися атрибутом `src`, який вказує браузеру, звідки завантажити файл зображення. Без цього атрибута елемент `<img>` не матиме жодного сенсу, оскільки браузер не знатиме, яке зображення потрібно відобразити. Крім того, рекомендується завжди включати атрибут `alt` з описом зображення для покращення доступності сайту.

Елемент `<input>` відкриває широкі можливості для створення різноманітних полів введення в веб-формах. Залежно від значення атрибута `type`, цей тег може створювати текстові поля для введення імені або електронної пошти, поля для паролів, прапорці для множинного вибору, радіокнопки для одиночного вибору з групи опцій, кнопки для відправки форм та багато інших елементів інтерфейсу. Гнучкість цього елемента робить його основою для створення інтерактивних веб-форм.

Тег `<meta>` містить метадані - інформацію про сам HTML-документ, яка не відображається безпосередньо на сторінці, але є критично важливою для браузерів та пошукових систем. Через цей тег можна встановити кодування символів документа, що забезпечує правильне відображення тексту різними мовами, додати опис сторінки для пошукових систем, вказати ключові слова,

встановити параметри відображення на мобільних пристроях та передати багато іншої службової інформації.

Важливою особливістю сучасного HTML5 є те, що одинарні теги більше не потребують символу косої риски "/" в кінці, який був обов'язковим у попередній специфікації XHTML. Тоді теги мали виглядати як `<br />` або `<img />`, щоб показати їхній самозакриваючий характер, але в HTML5 достатньо просто `<br>` або `<img>`, що спрощує написання коду та робить його більш читабельним.

Нижче — таблиця HTML-тегів, які є одинарними (void elements) — вони не мають закриваючого тегу і не містять в собі контенту.

Таблиця 2.1 — Таблиця одинарних (void) тегів HTML

| Тег   | Опис                                                     | Приклад                                                                                 | Основні атрибути                                            | Примітки                                                                        |
|-------|----------------------------------------------------------|-----------------------------------------------------------------------------------------|-------------------------------------------------------------|---------------------------------------------------------------------------------|
| area  | Область у зображенні-карті (image map).                  | <code>&lt;area shape="rect" coords="0,0,100,100" href="page.html" alt="link"&gt;</code> | shape, coords, href, alt, target                            | Працює в <code>&lt;map&gt;</code> .                                             |
| base  | Вказує базову URL чи ціль для всіх посилань на сторінці. | <code>&lt;base href="https://example.com/"&gt;</code>                                   | href, target                                                | Має розміщуватись у <code>&lt;head&gt;</code> ; лише один на документ.          |
| br    | Перенос рядка (line break).                              | Перше речення. <code>&lt;br&gt;</code> Друге речення.                                   | —                                                           | Часто вживають для коротких переносів; не замінює структурні блоки.             |
| col   | Описує колонку в таблиці для стилізації/ширини.          | <code>&lt;col span="1" style="width:50%"&gt;</code>                                     | span                                                        | Використовується всередині <code>&lt;colgroup&gt;</code> .                      |
| embed | Вставка зовнішнього ресурсу (плеєр, плагін).             | <code>&lt;embed src="file.swf" type="application/x-shockwave-flash"&gt;</code>          | src, type, width, height                                    | Застосування залежить від браузера; часто замінено <code>iframe/object</code> . |
| hr    | Горизонтальна лінія (розділювач).                        | <code>&lt;hr&gt;</code>                                                                 | size, width, align (deprecated)                             | Семантично означає тематичний розрив.                                           |
| img   | Вставка зображення.                                      | <code>&lt;img src="logo.png" alt="Логотип" width="100"&gt;</code>                       | src, alt, width, height, srcset, sizes                      | alt — обов'язковий для доступності.                                             |
| input | Поле введення (форма).                                   | <code>&lt;input type="text" name="user" placeholder="Ім'я"&gt;</code>                   | type, name, value, placeholder, required, disabled, checked | Дуже багатофункціональний; type визначає поведінку.                             |
| link  | Зв'язок з зовнішнім ресурсом (CSS, іконки тощо).         | <code>&lt;link rel="stylesheet" href="style.css"&gt;</code>                             | rel, href, type, media                                      | Використовується в <code>&lt;head&gt;</code> .                                  |

| Тег    | Опис                                                         | Приклад                                                               | Основні атрибути                   | Примітки                                   |
|--------|--------------------------------------------------------------|-----------------------------------------------------------------------|------------------------------------|--------------------------------------------|
| meta   | Метадані сторінки (charset, viewport).                       | <meta charset="utf-8">                                                | charset, name, content, http-equiv | Важливий для SEO та кодування.             |
| param  | Параметр для <object>.                                       | <param name="movie" value="movie.swf">                                | name, value                        | Використовується разом з <object>.         |
| source | Джерело для мультимедіа (<video>/<audio>) або для <picture>. | <source src="video.mp4" type="video/mp4">                             | src, type, media                   | Дозволяє вказати кілька форматів/версій.   |
| track  | Субтитри, капшени або опис для медіа.                        | <track kind="subtitles" src="uk.vtt" srclang="uk" label="Українська"> | kind, src, srclang, label, default | Використовується в <video>/<audio>.        |
| wbr    | Місце потенційного переносу слова (word break opportunity).  | longword<wbr>able                                                     | —                                  | Допомагає браузеру переносити довгі слова. |

У HTML5 ці теги називаються **void elements**: вони не мають закриваючого тегу і не можуть містити вміст між відкриваючим і закриваючим тегом. У XHTML іноді пишуться як самозакриваючі: `<br />`, ``. В HTML5 це не обов'язково — достатньо `<br>`, `<img>`.

## 2.1.2 Парні теги HTML

Парні теги в HTML представляють собою фундаментальну концепцію мови розмітки, яка дозволяє створювати контейнери для контенту та організовувати структуру веб-документів. На відміну від одинарних тегів, які виконують свою функцію самостійно, парні теги працюють як своєрідні "обгортки" або "контейнери", що охоплюють певну частину контенту і надають їй специфічні властивості або значення.

Структура парних тегів є досить простою, але водночас потужною: кожен парний елемент починається з відкриваючого тега, який виглядає як `<назва_тега>`, і закінчується закриваючим тегом `</назва_тега>`, де косая риска перед назвою сигналізує браузеру про закінчення елемента. Між цими двома тегами розміщується контент, який може включати звичайний текст, інші HTML-елементи, або навіть комбінацію різних типів контенту.

Ця структура створює те, що програмісти називають "контейнерною моделлю", де кожен елемент може містити інші елементи, формуючи

деревоподібну ієрархію документа. Така організація дозволяє браузеру розуміти, які частини контенту пов'язані між собою, як вони повинні відображатися, і які стилі або поведінка мають застосовуватися до кожної групи елементів.

Правильне вкладення елементів є критично важливим аспектом роботи з парними тегамі. Це означає, що якщо один елемент починається всередині іншого, він повинен бути повністю завершений до того, як закінчиться зовнішній елемент. Наприклад, якщо ви відкрили тег `<div>`, а потім всередині нього відкрили тег `<p>`, то спочатку потрібно закрити `</p>`, і тільки потім `</div>`. Порушення цього принципу може призвести до того, що браузер неправильно інтерпретує структуру документа і відобразить сторінку не так, як задумано розробником.

Тег `<html>` є найвищим у ієрархії HTML-документа і служить кореневим контейнером, який повідомляє браузеру, що весь контент між відкриваючим `<html>` та закриваючим `</html>` тегамі є HTML-кодом. Цей елемент зазвичай містить атрибут `lang`, який вказує основну мову документа, що допомагає браузерам та допоміжним технологіям правильно обробляти текст.

Елемент `<head>` функціонує як "голова" документа, де розміщується вся службова інформація, яка не відображається безпосередньо користувачеві, але є життєво важливою для правильного функціонування сторінки. Сюди входять метадані про кодування тексту, посилання на файли стилів CSS, які визначають зовнішній вигляд сторінки, скрипти JavaScript для інтерактивності, та інша технічна інформація, необхідна браузеру для правильного відображення та функціонування сторінки.

Тег `<title>` має особливе значення як для користувачів, так і для пошукових систем. Текст, розміщений між `<title>` та `</title>`, стає назвою веб-сторінки, яка відображається у вкладці браузера, використовується як назва закладки при збереженні сторінки, і служить головним заголовком у результатах пошукових систем. Правильно написаний заголовок може значно

вплинути на те, чи знайдуть користувачі вашу сторінку через пошукові системи.

Елемент `<body>` є контейнером для всього видимого контенту веб-сторінки - це та частина HTML-документа, яку користувач безпосередньо бачить і з якою взаємодіє. Всі тексти, зображення, посилання, форми, кнопки та інші елементи інтерфейсу повинні розміщуватися саме всередині цього контейнера. Браузер відображає тільки той контент, який знаходиться в межах `<body>`, ігноруючи все, що розміщено поза ним.

Тег `<div>` є універсальним блоковим контейнером, який не має власного семантичного значення, але надзвичайно корисний для організації та групування контенту. Розробники використовують `<div>` для створення розділів сторінки, застосування спільних стилів до групи елементів, або для маніпулювання групами елементів за допомогою JavaScript. Цей елемент займає всю доступну ширину свого батьківського контейнера та автоматично переноситься на новий рядок.

На противагу `<div>`, елемент `<span>` є рядковим контейнером, який використовується для роботи з невеликими частинами тексту без порушення потоку документа. Якщо `<div>` створює окремий блок, то `<span>` залишається частиною рядка тексту і дозволяє застосовувати стилі або скрипти лише до обраних слів або фраз. Це робить `<span>` ідеальним інструментом для виділення окремих слів кольором, зміни шрифту частини речення, або додавання інтерактивності до конкретних фрагментів тексту.

Таблиця 2.2 — Таблиця парних тегів HTML

| Тег                       | Опис                                                  | Приклад                                             | Основні атрибути  | Примітки                       |
|---------------------------|-------------------------------------------------------|-----------------------------------------------------|-------------------|--------------------------------|
| <code>&lt;html&gt;</code> | Кореневий елемент HTML-документа.                     | <code>&lt;html lang="uk"&gt;...&lt;/html&gt;</code> | <code>lang</code> | Містить увесь код сторінки.    |
| <code>&lt;head&gt;</code> | Службова частина документа: метадані, стилі, скрипти. | <code>&lt;head&gt;...&lt;/head&gt;</code>           | —                 | Не відображається на сторінці. |

| Тег         | Опис                                 | Приклад                                   | Основні атрибути                 | Примітки                                    |
|-------------|--------------------------------------|-------------------------------------------|----------------------------------|---------------------------------------------|
| <body>      | Основний вміст сторінки.             | <body>...</body>                          | —                                | Усе, що бачить користувач.                  |
| <div>       | Блоковий контейнер.                  | <div class="box">...</div>                | class, id, style                 | Використовується для групування елементів.  |
| <span>      | Рядковий контейнер.                  | <span style="color:red">Текст</span>      | class, id, style                 | Для виділення частини тексту.               |
| <p>         | Абзац.                               | <p>Це абзац.</p>                          | class, id                        | Блочний елемент.                            |
| <h1>...<h6> | Заголовки різних рівнів.             | <h1>Головний заголовок</h1>               | class, id                        | Використовуються для структурування тексту. |
| <a>         | Гіперпосилання.                      | <a href="page.html">Посилання</a>         | href, target, rel                | Може бути як текст, так і кнопка.           |
| <ul>        | Нумерований список (unordered list). | <ul><li>Елемент</li></ul>                 | type (deprecated)                | Використовується з <li>.                    |
| <ol>        | Нумерований список (ordered list).   | <ol><li>Перший</li></ol>                  | start, type                      | Також працює з <li>.                        |
| <li>        | Елемент списку.                      | <li>Пункт</li>                            | value                            | Використовується всередині <ul>/<ol>.       |
| <table>     | Таблиця.                             | <table>...</table>                        | border, cellpadding (deprecated) | Створює табличну структуру.                 |
| <tr>        | Рядок таблиці.                       | <tr>...</tr>                              | —                                | Усередині <table>.                          |
| <td>        | Комірка таблиці.                     | <td>Текст</td>                            | colspan, rowspan                 | Для даних у таблиці.                        |
| <th>        | Заголовок комірки таблиці.           | <th>Назва</th>                            | scope                            | Виділяється жирним.                         |
| <form>      | Форма для введення даних.            | <form action="/submit">...</form>         | action, method, enctype          | Контейнер для елементів введення.           |
| <label>     | Підпис до поля введення.             | <label for="name">Ім'я</label>            | for                              | Зв'язує текст і поле введення.              |
| <textarea>  | Багаторядкове поле введення.         | <textarea rows="4" cols="50"></textarea>  | rows, cols, placeholder          | Парний аналог <input>.                      |
| <select>    | Випадаючий список.                   | <select><option>Варіант</option></select> | multiple, size                   | Використовується з <option>.                |
| <option>    | Елемент випадаючого списку.          | <option value="1">Перший</option>         | value, selected                  | Усередині <select>.                         |
| <script>    | Вставка JavaScript-коду або          | <script src="app.js"></script>            | src, type, defer, async          | Розміщується в <head> або перед </body>.    |

| Ter          | Опис                                | Приклад                                                           | Основні атрибути         | Примітки                            |
|--------------|-------------------------------------|-------------------------------------------------------------------|--------------------------|-------------------------------------|
|              | підключення зовнішнього.            |                                                                   |                          |                                     |
| <style>      | Вбудовані CSS-стилі.                | <style>body {color:red;}</style>                                  | media                    | Рекомендовано зовнішні CSS.         |
| <header>     | Шапка сайту або секції.             | <header>...</header>                                              | —                        | Семантичний елемент HTML5.          |
| <footer>     | Нижній колонтитул.                  | <footer>...</footer>                                              | —                        | Містить контакти, авторські права.  |
| <section>    | Семантичний розділ контенту.        | <section>...</section>                                            | —                        | Логічно групує зміст.               |
| <article>    | Незалежна стаття або блок контенту. | <article>...</article>                                            | —                        | Використовується для блогів, новин. |
| <nav>        | Блок навігації.                     | <nav>...</nav>                                                    | —                        | Для меню сайту.                     |
| <main>       | Основний контент сторінки.          | <main>...</main>                                                  | —                        | Тільки один на сторінці.            |
| <aside>      | Блок додаткової інформації.         | <aside>...</aside>                                                | —                        | Для сайдбарів, реклами.             |
| <figure>     | Блок для зображення з підписом.     | <figure><figcaption>Опис</figcaption></figure> | —                        | Семантична пара з <figcaption>.     |
| <figcaption> | Підпис до <figure>.                 | <figcaption>Пояснення</figcaption>                                | —                        | Може бути зверху або знизу.         |
| <video>      | Відеофайл на сторінці.              | <video src="movie.mp4"></video> controls<source>                  | controls, autoplay, loop | Може містити кілька <source>.       |
| <audio>      | Аудіофайл на сторінці.              | <audio src="sound.mp3"></audio> controls<source>                  | controls, autoplay, loop | Аналогічно до <video>.              |

## Елементи HTML



Рисунок 2.1 — Парні теги HTML5

### 2.1.3 Теги для коментарів

Уявіть собі, що ви будujete будинок за кресленнями. Ви можете робити на них нотатки, позначки чи короткі пояснення, які зрозумілі лише будівельникам, але невидимі для тих, хто потім житиме в цьому будинку. В HTML-коментарі виконують саме таку роль. Це невидимі для кінцевого користувача елементи, які слугують своєрідними "пояснювальними записками" для розробників. Браузер повністю ігнорує будь-який текст, що знаходиться всередині коментаря, і не показує його на сторінці. Однак, якщо подивитися у вихідний код сторінки, наприклад, через інструменти розробника, ви побачите їх у повному обсязі.

HTML-коментарі є невидимими для користувачів елементами, які дозволяють розробникам додавати пояснення, нотатки або тимчасово приховувати частини коду без видалення їх з документа. Коментарі є надзвичайно корисними для документування коду, особливо при роботі в команді або при поверненні до проекту після тривалої перерви.



Синтаксис HTML-коментарів виглядає наступним чином: `<!-- текст коментаря -->`. Все, що знаходиться між `<!--` та `-->`, буде проігноровано браузером при відображенні сторінки, але залишиться видимим у вихідному коді при перегляді через інструменти розробника.

Коментарі можуть займати як один рядок, так і бути багаторядковими. Вони можуть містити будь-який текст, окрім послідовності символів `--`, яка може спричинити проблеми з парсингом коментаря в деяких браузерах.

Синтаксис коментаря:

```
<!-- Це коментар -->
```

HTML-коментарі — це не просто декорація, а важливий інструмент для організації та підтримки коду. Навіщо потрібні коментарі?

1. Документування: Вони дозволяють розробникам залишати пояснення до складних або неочевидних частин коду, пояснюючи, для чого потрібен той чи інший елемент або блок. Це особливо важливо при роботі в команді, де кожен може швидко зрозуміти логіку колеги.
2. Дебаггінг: Якщо потрібно тимчасово вимкнути частину коду для тестування або пошуку помилок, замість того, щоб видаляти її, можна просто помістити її в коментар. Це дозволяє легко повернути її назад, коли це буде потрібно.
3. Читабельність: У великих файлах коментарі можна використовувати як роздільники, позначаючи початок і кінець окремих секцій сторінки (наприклад, "Блок навігації", "Основний вміст", "Футер"). Це значно спрощує навігацію по коду.
4. Інформаційні нотатки: Розробники можуть додавати інформацію про авторство, дату останніх змін або нагадування про майбутні доопрацювання.

Важливо пам'ятати, що HTML-коментарі залишаються видимими у вихідному коді сторінки, тому вони не повинні містити конфіденційну інформацію, пароли або інші чутливі дані.

Приклад використання:

```

<!DOCTYPE html>
<html lang="uk">
<head>
  <meta charset="UTF-8">
  <title>Приклад коментаря</title>
</head>
<body>
  <!-- Це головний заголовок сторінки -->
  <h1>Ласкаво просимо на сайт</h1>

  <p>Цей текст відображається для користувачів.</p>

  <!--
    Нижче буде додано новий розділ у майбутньому:
    <section>
      <h2>Новини</h2>
    </section>
  -->
</body>
</html>

```

У цьому прикладі: коментар пояснює структуру коду; закоментований блок (<section>...</section>) не відображається у браузері, але залишається в коді для розробників.

У HTML5 коментарі — важливий інструмент для документації коду, але їх неправильне використання може зробити код захаращеним, менш читабельним і навіть небезпечним. Ось приклади "поганих" коментарів і рекомендації, як їх уникати.

### ✗ Погані приклади коментарів:

#### 1. Надмірне коментування очевидного

```

<!-- Початок тіла сторінки -->

```

```
<body>
  <!-- Це заголовок -->
  <h1>Вітаємо!</h1>
  <!-- Це абзац -->
  <p>Тут буде текст.</p>
</body>
```

Чому погано: коментарі дублюють те, що й так видно в коді, і не несуть додаткової користі.

## 2. Застарілі або хибні коментарі

```
<!-- Тут буде меню (видалено у попередньому коміті) -->
<!-- TODO: додати форму реєстрації (форма вже є нижче) -->
```

Чому погано: коментарі вводять в оману, якщо не оновлюються разом із кодом.

## 3. Розголошення конфіденційної інформації

```
<!-- Логін адміністратора: admin, пароль: 123456 -->
```

Чому погано: такі коментарі можуть призвести до витоку даних і є серйозною вразливістю.

## 4. Велика кількість закоментованого коду

```
<!--
<div class="old-version">
  <p>Це застаріла версія сторінки</p>
</div>
-->
```

Чому погано: код, що вже не використовується, захаращує структуру і ускладнює підтримку. Для цього існують системи контролю версій (Git).

 Рекомендовані практики коментування у HTML5:

Коментарі мають пояснювати "чому", а не "що".

Замість:

```
<!-- Кнопка -->
```

```
<button>Надіслати</button>
```

Краще:

```
<!-- Кнопка надсилання форми, викликає валідацію -->
<button type="submit">Надіслати</button>
```

Використовувати коментарі для логічних блоків.

```
<!-- Блок навігації -->
<nav>
  <ul>
    <li><a href="/">Головна</a></li>
    <li><a href="/about">Про нас</a></li>
  </ul>
</nav>
```

Оновлювати коментарі при зміні коду.

Якщо блок змінюється, коментар має залишатися актуальним.

Уникати розголошення даних і секретів.

Використовувати багаторядкові коментарі для великих пояснень тільки за потреби.

```
<!--
Цей розділ відповідає за адаптивність.
Для екранів менше 768px використовується інший шаблон.
-->
```

### 2.1.4 Структура HTML-документа

Кожен валідний HTML-документ має дотримуватися певної структури, яка забезпечує правильне відображення сторінки в різних браузерах та сумісність з веб-стандартами. Базова структура HTML-документа включає декілька обов'язкових елементів, кожен з яких виконує специфічну роль у формуванні веб-сторінки.

Стандартна структура починається з декларації типу документа `<!DOCTYPE html>`, яка інформує браузер про версію HTML, що

використовується. У HTML5 ця декларація значно спрощена порівняно з попередніми версіями та просто вказує на використання найновішого стандарту HTML.

Після декларації DOCTYPE слідує кореневий елемент `<html>`, який охоплює весь контент документа. Цей елемент зазвичай містить атрибут `lang`, що вказує основну мову документа, наприклад `<html lang="uk">` для українськомовних сторінок.

Всередині елемента `html` знаходяться два основні розділи: `<head>` та `<body>`. Елемент `head` містить метадані та службову інформацію, яка не відображається безпосередньо на сторінці, але впливає на її поведінку та відображення. Елемент `body` містить весь видимий контент сторінки.

Мінімальна структура HTML-документа виглядає наступним чином:

```
<!DOCTYPE html>
<html lang="uk">
<head>
  <meta charset="UTF-8">
  <title>Заголовок сторінки</title>
</head>
<body>
  <!-- Видимий контент сторінки -->
</body>
</html>
```

Ця структура забезпечує базову функціональність та сумісність з усіма сучасними браузерами, створюючи основу для додавання більш складного контенту та функціональності.

### 2.1.5 Елемент `<head>`

Елемент `head` є критично важливою частиною HTML-документа, яка містить метадані та службову інформацію про сторінку. Хоча контент цього елемента не відображається безпосередньо користувачеві, він суттєво впливає

на поведінку сторінки, її індексацію пошуковими системами, відображення в браузері та взаємодію з зовнішніми ресурсами.

Елемент `<title>` є обов'язковим дочірнім елементом `head` та визначає заголовок документа. Цей заголовок відображається у вкладці браузера, використовується як назва закладки при збереженні сторінки користувачем, та служить основним заголовком у результатах пошукових систем. Заголовок повинен бути унікальним для кожної сторінки сайту та точно описувати її зміст.

Метатеги `<meta>` надають браузеру додаткову інформацію про документ. Найважливішим є `<meta charset="UTF-8">`, який встановлює кодування символів та повинен розташовуватися якомога раніше в `head` для забезпечення правильного відображення тексту. Інші корисні метатеги включають `viewport` для адаптивного дизайну, `description` для опису сторінки в пошукових результатах, та `keywords` для ключових слів.

Елемент `<link>` використовується для підключення зовнішніх ресурсів, найчастіше CSS-стилів. Наприклад, `<link rel="stylesheet" href="styles.css">` підключає файл стилів до документа. Цей елемент також може використовуватися для підключення іконок сайту, шрифтів та інших ресурсів.

Елемент `<style>` дозволяє включати CSS-стилі безпосередньо в HTML-документ, хоча зазвичай рекомендується виносити стилі у зовнішні файли для кращої організації коду.

Елемент `<script>` може розташовуватися як в `head`, так і в `body`, та використовується для підключення JavaScript-коду або зовнішніх скриптів.

### 2.1.6 Елемент `<body>`

Елемент `body` містить весь видимий контент веб-сторінки та являє собою область, де розміщуються всі елементи, з якими взаємодіє користувач. Цей елемент служить контейнером для всього контенту, який буде відображатися в

вікні браузера, включаючи текст, зображення, посилання, форми, мультимедіа та інтерактивні елементи.

Структурна організація контенту в межах `body` є критично важливою для створення логічної та доступної веб-сторінки. HTML5 запровадив семантичні елементи, які допомагають краще структурувати контент та покращують доступність для користувачів з обмеженими можливостями та пошукових систем.

Семантичні елементи HTML5 включають `<header>` для шапки сайту або розділу, `<nav>` для навігаційних меню, `<main>` для основного контенту сторінки, `<article>` для автономних статей або записів, `<section>` для тематичних розділів, `<aside>` для додаткової інформації або бічних панелей, та `<footer>` для підвалу сайту або розділу.

Контент `body` може включати різноманітні елементи: заголовки різних рівнів (`h1-h6`) для структурування інформації, абзаци (`p`) для основного тексту, списки (`ul`, `ol`, `dl`) для організації елементів, таблиці (`table`) для табличних даних, форми (`form`) для збору користувацьких даних, зображення (`img`) для візуального контенту, та мультимедійні елементи (`video`, `audio`) для відео та аудіо контенту.

Правильна організація контенту в `body` також включає використання атрибутів доступності, таких як `alt` для зображень, `label` для полів форм, та ARIA-атрибутів для покращення доступності складних інтерфейсів.

### 2.1.7 Блокові елементи

Блокові елементи являють собою фундаментальний тип HTML-елементів, які створюють окремі блоки контенту та займають всю доступну ширину свого батьківського контейнера. Ці елементи автоматично починаються з нового рядка та створюють перенесення рядка після себе, що робить їх ідеальними для структурування основного макету сторінки.

Заголовки (h1-h6) є одними з найважливіших блокових елементів, які створюють ієрархічну структуру контенту. H1 зазвичай використовується для головного заголовка сторінки, h2 для основних розділів, h3 для підрозділів тощо. Правильне використання заголовків критично важливе для SEO та доступності сайту.

```
<h1>Основний заголовок сторінки</h1>
<h2>Розділ</h2>
<h3>Підрозділ</h3>
```

H1 — головний заголовок (використовується один раз на сторінці). H2–H6 — підзаголовки для логічної структури.

Абзаци (<p>) служать для організації текстового контенту в логічні блоки та є основним елементом для відображення звичайного тексту на веб-сторінках.

```
<p>Це перший абзац тексту, який відкриває тему
розділу.</p>
<p>Другий абзац надає додаткову інформацію.</p>
```

Елемент <div> є універсальним контейнером, який не має семантичного значення, але надзвичайно корисний для групування інших елементів та застосування стилів або скриптів до групи елементів.

```
<div class="content-block">
  <h2>Розділ статті</h2>
  <p>Тут розміщується основний текст.</p>
</div>
```

Списки включають впорядковані списки (<ol>), неупорядковані списки (<ul>), та списки визначень (<dl>), кожен з яких служить для організації інформації у структурованому форматі.

```
<h3>Неупорядкований список:</h3>
<ul>
  <li>Перший елемент</li>
  <li>Другий елемент</li>
```



```

</ul>

<h3>Впорядкований список:</h3>
<ol>
    <li>Крок 1</li>
    <li>Крок 2</li>
</ol>

<h3>Список визначень:</h3>
<dl>
    <dt>HTML</dt>
    <dd>Мова розмітки гіпертексту</dd>
    <dt>CSS</dt>
    <dd>Каскадні таблиці стилів</dd>
</dl>

```

Таблиці (<table>) використовуються для відображення табличних даних та складаються з рядків (<tr>), заголовків стовпців (<th>), та комірок (<td>).

```

<table border="1">
    <tr>
        <th>Ім'я</th>
        <th>Вік</th>
    </tr>
    <tr>
        <td>Анна</td>
        <td>25</td>
    </tr>
    <tr>
        <td>Олег</td>
        <td>30</td>
    </tr>

```

```
</table>
```

Семантичні блокові елементи HTML5 включають `<article>`, `<section>`, `<header>`, `<footer>`, `<nav>`, `<aside>`, та `<main>`, які надають семантичне значення різним частинам документа та покращують структуру сторінки.

```
<header>
```

```
    <h1>Назва сайту</h1>
```

```
</header>
```

```
<nav>
```

```
    <ul>
```

```
        <li><a href="#">Головна</a></li>
```

```
        <li><a href="#">Про нас</a></li>
```

```
    </ul>
```

```
</nav>
```

```
<main>
```

```
    <article>
```

```
        <h2>Стаття</h2>
```

```
        <p>Основний текст статті.</p>
```

```
    </article>
```

```
    <aside>
```

```
        <p>Корисні посилання.</p>
```

```
    </aside>
```

```
</main>
```

```
<footer>
```

```
    <p>© 2025 Усі права захищено</p>
```

```
</footer>
```

Блокові елементи можуть містити як інші блокові, так і рядкові елементи, але потрібно дотримуватися правил:

- Не можна вкладати заголовок (<h1>–<h6>) у <p>.
- Списки (<ul>, <ol>) не повинні містити безпосередньо інші блокові елементи, окрім <li>.
- Семантичні блоки повинні використовуватися згідно з їх призначенням (наприклад, <main> лише один на сторінці).

### 2.1.8 Рядкові елементи

Рядкові елементи займають тільки ту ширину, яка необхідна для їх контенту, та не створюють переносів рядків до або після себе. Ці елементи зазвичай використовуються для форматування частин тексту або додавання інтерактивних елементів всередині блокового контенту.

Посилання (<a>) є найважливішими рядковими елементами, які дозволяють створювати гіперпосилання на інші сторінки, файли або розділи в межах того ж документа. Елемент а може містити текст, зображення або інші рядкові елементи.

```
<p>Відвідайте <a href="https://example.com">наш
сайт</a> для отримання додаткової інформації.</p>
<a> створює гіперпосилання.
```

Може містити текст, зображення чи інші рядкові елементи:

```
<a href="home.html">
   Головна
сторінка
</a>
```

Елементи форматування тексту включають <strong> для важливого тексту (зазвичай відображається жирним шрифтом), <em> для акцентованого тексту (зазвичай курсивом), <mark> для виділеного тексту, <small> для дрібного тексту, <sub> та <sup> для підрядкових та надрядкових символів.

```
<p>Це <strong>важливий</strong> текст і це
<em>виділений</em> текст.</p>
```

```
<p>Тут є <mark>підсвічений</mark> фрагмент та
<small>дрібний</small> текст.</p>
```

```
<p>Н<sub>2</sub>О і х<sup>2</sup> – приклади
підрядкових та надрядкових символів.</p>
```

- `<strong>` — акцентує важливість (часто жирний шрифт).
- `<em>` — додає емоційний акцент (часто курсив).
- `<mark>` — виділення кольором.
- `<sub>` і `<sup>` — для хімічних формул, математичних виразів.

Елемент `<span>` є рядковим еквівалентом `div` та служить універсальним контейнером для застосування стилів або скриптів до частини тексту без зміни його структури.

```
<p>Це звичайний текст, а ось <span style="color:
red;">виділений</span> фрагмент.</p>
```

Елементи коду включають `<code>` для відображення коду в тексті, `<kbd>` для позначення клавіш клавіатури, `<samp>` для зразків виводу програми, та `<var>` для змінних.

```
<p>Щоб вивести текст, використовуйте команду
<code>console.log("Hello");</code></p>
```

```
<p>Натисніть <kbd>Ctrl</kbd> + <kbd>C</kbd>, щоб
скопювати текст.</p>
```

```
<p>Програма повернула: <samp>Process completed
successfully</samp></p>
```

```
<p>Значення змінної: <var>x</var> = 42</p>
```

Елементи часу та цитат включають `<time>` для позначення дат та часу, `<cite>` для посилань на джерела, `<q>` для коротких цитат, та `<abbr>` для аббревіатур.

```
<p>Зустріч відбудеться <time datetime="2025-09-
01T10:00">1 вересня 2025, 10:00</time>.</p>
```

```
<p>Як казав <cite>Ісаак Ньютон</cite>: <q>Якщо я бачив
далі, то лише стоячи на плечах гігантів.</q></p>
```

```
<p><abbr title="HyperText Markup Language">HTML</abbr>
- основа веб-розробки.</p>
```

Мультимедійні рядкові елементи включають `<img>` для зображень, `<audio>` та `<video>` для мультимедійного контенту (хоча ці елементи можуть поводитися як блокові залежно від контексту).

```
<p>Подивіться на цю картинку: </p>
```

```
<p>Прослухайте аудіофайл: <audio controls
src="song.mp3"></audio></p>
```

Важливо пам'ятати, що рядкові елементи не можуть містити блокові елементи, хоча можуть містити інші рядкові елементи. Це правило є фундаментальним для створення валідного HTML-коду.

Наприклад, це валідно:

```
<a href="#"><strong>Перейти</strong></a>
```

А це — ні:

```
<a href="#">
  <div>Перейти</div>
</a>
```

Таблиця 2.3 — Порівняльна таблиця: Блокові vs Рядкові елементи

| Характеристика              | Блокові елементи                                                                                                                                                                                                                                     | Рядкові елементи (inline)                                                                                                                                                                                                |
|-----------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Розміщення</b>           | Починаються з нового рядка, займають всю доступну ширину батьківського контейнера                                                                                                                                                                    | Розташовуються в одному рядку, займають лише стільки місця, скільки потрібно для їхнього вмісту                                                                                                                          |
| <b>Приклади тегів</b>       | <code>&lt;div&gt;</code> , <code>&lt;p&gt;</code> , <code>&lt;h1&gt;</code> – <code>&lt;h6&gt;</code> , <code>&lt;section&gt;</code> , <code>&lt;article&gt;</code> , <code>&lt;ul&gt;</code> , <code>&lt;ol&gt;</code> , <code>&lt;table&gt;</code> | <code>&lt;span&gt;</code> , <code>&lt;a&gt;</code> , <code>&lt;strong&gt;</code> , <code>&lt;em&gt;</code> , <code>&lt;img&gt;</code> , <code>&lt;label&gt;</code> , <code>&lt;code&gt;</code> , <code>&lt;br&gt;</code> |
| <b>Можливість вкладення</b> | Можуть містити як блокові, так і рядкові елементи                                                                                                                                                                                                    | Можуть містити лише інші рядкові елементи, але не блокові                                                                                                                                                                |

| Характеристика    | Блокові елементи                                                          | Рядкові елементи (inline)                                                                      |
|-------------------|---------------------------------------------------------------------------|------------------------------------------------------------------------------------------------|
| Вплив на розмітку | Створюють логічні та структурні блоки контенту, що впливають на макет     | Використовуються для виділення окремих слів, фрагментів тексту або невеликих елементів у рядку |
| CSS-властивості   | Легко налаштовуються за допомогою width, height, margin, padding          | width і height не діють напряму (для керування розміром зазвичай використовується line-height) |
| Застосування      | Основна структура веб-сторінки, великі контейнери для групування контенту | Локальні стилі та акценти в тексті або посиланнях                                              |

Приклад: Блоковий і рядковий елемент разом

```
<div>
  <h2>Заголовок</h2>
  <p>Це <span style="color: red;">рядковий</span>
елемент у середині абзацу.</p>
</div>
```

У цьому прикладі:

- `<div>` і `<p>` — блокові елементи;
- `<span>` — рядковий елемент, що змінює лише частину тексту.

### 2.1.9 HTML текст

Робота з текстом в HTML включає використання різноманітних елементів для створення структурованого, читабельного та семантично правильного контенту. HTML надає широкий набір інструментів для форматування тексту, створення заголовків, абзаців, списків та спеціальних текстових елементів.

Заголовки (h1-h6) створюють ієрархічну структуру документа та допомагають користувачам та пошуковим системам зрозуміти організацію контенту. Рекомендується використовувати тільки один h1 на сторінці для

головного заголовка та послідовно використовувати інші рівні заголовків для створення логічної структури.

Абзаци (<p>) є основним елементом для організації тексту та автоматично створюють відступи між блоками тексту. Кожен абзац повинен містити одну основну думку або тему.

Елементи семантичного форматування надають значення тексту, а не просто змінюють його зовнішній вигляд. Елемент <strong> позначає важливий текст, <em> вказує на акцент або наголос, <mark> виділяє текст як релевантний для поточного контексту.

Елементи типографського форматування включають <b> для жирного тексту без додаткового семантичного значення, <i> для курсиву, <u> для підкресленого тексту, <s> для закресленого тексту.

Спеціальні текстові елементи включають <blockquote> для довгих цитат, які займають окремий блок, <q> для коротких цитат в межах тексту, <cite> для посилань на джерела, <address> для контактної інформації.

Елементи для технічного контенту включають <code> для фрагментів коду, <pre> для преформатованого тексту, який зберігає пробіли та переноси рядків, <samp> для прикладів виводу програм, <kbd> для позначення клавіш.

Робота з символами в HTML включає використання HTML-сутностей для відображення спеціальних символів, таких як &lt; для <, &gt; для >, &amp; для &, &nbsp; для нерозривного пробілу.

### 2.1.10 HTML посилання

Посилання є основою Всесвітньої павутини та дозволяють користувачам переміщуватися між різними ресурсами в Інтернеті. HTML-посилання створюються за допомогою елемента <a> (anchor) з атрибутом href, який вказує призначення посилання.

Основні типи посилань включають абсолютні посилання, які містять повну URL-адресу включаючи протокол та домен (наприклад,

<https://example.com/page.html>), відносні посилання, які вказують шлях відносно поточного документа (наприклад, [page.html](#) або [../folder/page.html](#)), та якірні посилання, які ведуть до певної частини тієї ж сторінки (наприклад, [#section1](#)).

Атрибути посилань значно розширюють їхню функціональність. Атрибут `target` визначає, де відкриється посилання: `_self` для поточного вікна (за замовчуванням), `_blank` для нового вікна або вкладки, `_parent` для батьківського фрейму, `_top` для повного вікна браузера.

Атрибут `title` надає додаткову інформацію про посилання, яка відображається як спливаюча підказка при наведенні курсора миші.

Атрибут `download` дозволяє вказати, що посилання призначене для завантаження файлу, а не для навігації, та може запропонувати назву файлу для збереження.

Атрибути `rel` описують відносини між поточним документом та пов'язаним ресурсом. Важливі значення включають `nofollow` для посилань, які не повинні впливати на рейтинг в пошукових системах, `noopener` та `noreferrer` для безпеки при відкритті посилань у нових вкладках, та `alternate` для альтернативних версій документа.

Посилання на електронну пошту створюються за допомогою схеми `mailto:` (наприклад, `<a href="mailto:user@example.com">Написати листа</a>`), а посилання на телефонні номери - за допомогою схеми `tel:` (наприклад, `<a href="tel:+380123456789">Подзвонити</a>`).

Внутрішні посилання дозволяють швидко переміщуватися по довгому документу, використовуючи якірі - елементи з атрибутом `id`, на які можна посилатися через символ `#` в URL.

### 2.1.11 HTML зображення

Зображення є невід'ємною частиною сучасних веб-сторінок та значно покращують користувацький досвід, надаючи візуальний контекст та



допомагаючи передавати інформацію ефективніше за текст. HTML надає кілька способів роботи із зображеннями, кожен з яких має свої особливості та призначення.

Елемент `<img>` є основним способом вставки зображень в HTML-документи. Цей одинарний тег обов'язково повинен містити атрибут `src`, який вказує шлях до файлу зображення, та атрибут `alt`, який надає альтернативний опис зображення для користувачів з порушеннями зору та у випадках, коли зображення не може бути завантажене.

Атрибут `alt` є критично важливим для доступності веб-сайтів. Він повинен містити стислий, але інформативний опис зображення, який передає його суть та контекст. Для декоративних зображень, які не несуть інформаційного навантаження, атрибут `alt` може бути порожнім (`alt=""`), але повинен бути присутнім.

Атрибути `width` та `height` дозволяють встановити розміри зображення в пікселях. Хоча розміри можна також контролювати через CSS, встановлення цих атрибутів допомагає браузеру зарезервувати простір для зображення до його завантаження, що покращує продуктивність сторінки та зменшує зміщення макету.

Атрибут `title` надає додаткову інформацію про зображення, яка відображається як спливаюча підказка при наведенні курсора миші на зображення.

Адаптивні зображення стали стандартом в сучасній веб-розробці. Елемент `<picture>` дозволяє вказати кілька варіантів зображення для різних умов відображення, включаючи різні розміри екрану, щільність пікселів та формати зображень.

Атрибут `srcset` елемента `img` дозволяє вказати кілька варіантів одного зображення з різною роздільною здатністю, дозволяючи браузеру вибрати найбільш підходящий варіант залежно від характеристик пристрою користувача.

Формати зображень відіграють важливу роль у продуктивності веб-сайтів. Сучасні формати, такі як WebP та AVIF, забезпечують кращу компресію порівняно з традиційними JPEG та PNG, але підтримуються не всіма браузерами, що вимагає використання fallback-варіантів.

Lazy loading є сучасною технікою, яка дозволяє відкладати завантаження зображень до моменту, коли вони стають видимими користувачу. Це досягається за допомогою атрибута `loading="lazy"`.

### 2.1.12 HTML списки

HTML-списки є потужним інструментом для організації та структурування інформації на веб-сторінках. Вони не тільки покращують читабельність контенту, але й надають семантичне значення організованим даним, що важливо для доступності та SEO-оптимізації.

Невпорядковані списки (`<ul>`) використовуються для групування пов'язаних елементів, порядок яких не має значення. Кожен елемент списку позначається тегом `<li>` та зазвичай відображається з маркером (крапка, квадрат або інший символ) зліва від тексту.

Невпорядковані списки ідеально підходять для переліків особливостей продукту, навігаційних меню, списків посилань або будь-яких інших елементів, де послідовність не є критичною. CSS дозволяє змінювати стиль маркерів або повністю їх приховувати для створення власного дизайну.

Впорядковані списки (`<ol>`) призначені для елементів, де послідовність або ієрархія мають значення. За замовчуванням вони нумеруються арабськими цифрами (1, 2, 3...), але за допомогою атрибута `type` можна змінити стиль нумерації на римські цифри (I, II, III... або i, ii, iii...) чи літери (A, B, C... або a, b, c...).

Атрибут `start` дозволяє почати нумерацію з певного числа, що корисно для продовження списку після перерви в тексті. Атрибут `reversed` створює

нумерацію в зворотному порядку, що може бути корисно для рейтингів або зворотних відліків.

Списки визначень (<dl>) використовуються для створення глосаріїв, словників або будь-яких інших структур, де потрібно поєднати терміни з їх описами. Ці списки складаються з термінів (<dt>) та їх визначень (<dd>).

Списки визначень особливо корисні для FAQ-секцій, технічних специфікацій, біографічних даних або будь-якого контенту, де потрібно встановити чіткі відносини між поняттями та їх поясненнями.

Вкладені списки дозволяють створювати ієрархічні структури, розміщуючи один список всередині елемента <li> іншого списку. Це особливо корисно для створення структурованих меню, планів документів, організаційних схем або будь-яких інших даних з багаторівневою структурою.

При створенні вкладених списків важливо дотримуватися правильної структури HTML, де вкладений список розміщується всередині елемента <li>, а не безпосередньо всередині <ul> або <ol>. Це забезпечує валідність коду та правильне відображення в усіх браузерях.

Стилізація списків за допомогою CSS дозволяє значно розширити їх візуальні можливості. Властивість list-style-type контролює вигляд маркерів або нумерацію, list-style-position визначає розташування маркерів відносно тексту, а list-style-image дозволяє використовувати власні зображення як маркери.

Сучасні веб-сайти часто використовують списки не тільки для традиційного відображення переліків, але й як основу для навігаційних меню, галерей зображень, карток товарів та інших складних інтерфейсних елементів, повністю змінюючи їх зовнішній вигляд за допомогою CSS.

### 2.1.13 Практичні рекомендації та найкращі практики

При роботі з HTML важливо дотримуватися певних принципів та найкращих практик, які забезпечують створення якісних, доступних та підтримуваних веб-сторінок.

Семантична розмітка повинна бути пріоритетом при виборі HTML-елементів. Замість використання універсальних `<div>` та `<span>` для всього контенту, слід обирати елементи, які найкраще передають семантичне значення контенту. Наприклад, використовувати `<article>` для статей, `<nav>` для навігації, `<aside>` для додаткового контенту.

Валідність HTML-коду є критично важливою для забезпечення сумісності з різними браузерами та пристроями. Всі теги повинні бути правильно закриті, атрибути - правильно оформлені, а структура документа - логічно організована.

Доступність веб-контенту повинна враховуватися з самого початку розробки. Це включає використання правильних заголовків для створення структури документа, надання альтернативного тексту для зображень, забезпечення належного контрасту кольорів, та створення логічного порядку табуляції для навігації з клавіатури.

Оптимізація продуктивності починається вже на рівні HTML. Це включає мінімізацію кількості HTTP-запитів, оптимізацію зображень, використання `lazy loading` для зображень поза межами видимої області, та правильну організацію структури документа для швидшого парсингу браузером.

Адаптивний дизайн повинен закладатися вже на рівні HTML-структури. Правильне використання семантичних елементів, логічна організація контенту та продумана ієрархія елементів створюють міцну основу для адаптації сторінки до різних розмірів екрану за допомогою CSS.

## 2.2 Створення макета сторінки та верстка

Створення макета веб-сторінки є одним із найважливіших етапів розробки веб-сайту, який визначає не лише зовнішній вигляд, але й функціональність та зручність використання ресурсу. У цьому розділі ми детально розглянемо основні принципи та техніки верстки HTML сторінок, які дозволять вам створювати професійні та адаптивні веб-інтерфейси. Сучасна верстка потребує глибокого розуміння взаємодії HTML та CSS, а також знання різних методів позиціонування елементів на сторінці.

### 2.2.1 Блокова верстка

Блокова верстка є фундаментальним підходом до створення макетів веб-сторінок, який базується на використанні блокових елементів як основних структурних одиниць. Цей метод передбачає розподіл контенту сторінки на логічні блоки, кожен з яких виконує певну функцію та має визначені розміри й позицію на сторінці.

Основою блокової верстки є HTML-елементи типу `<div>`, `<section>`, `<article>`, `<header>`, `<footer>` та інші семантичні теги HTML5, які за своєю природою є блоковими елементами. Блокові елементи займають всю доступну ширину батьківського контейнера та розташовуються один під одним, утворюючи вертикальну структуру документа.

```
<!DOCTYPE html>
<html lang="uk">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <title>Блокова верстка</title>
  <style>
    .container {
```

```
        max-width: 1200px;
        margin: 0 auto;
        padding: 20px;
    }

    .header {
        background-color: #2c3e50;
        color: white;
        padding: 20px;
        margin-bottom: 20px;
    }

    .content {
        background-color: #ecf0f1;
        padding: 20px;
        margin-bottom: 20px;
        min-height: 400px;
    }

    .sidebar {
        background-color: #95a5a6;
        padding: 20px;
        margin-bottom: 20px;
        min-height: 300px;
    }

    .footer {
        background-color: #34495e;
        color: white;
        padding: 20px;
```

```

        text-align: center;
    }
</style>
</head>
<body>
    <div class="container">
        <div class="header">
            <h1>Заголовок сайту</h1>
            <p>Підзаголовок або короткий опис</p>
        </div>

        <div class="content">
            <h2>Основний контент</h2>
            <p>Тут розміщується основний вміст
сторінки...</p>
        </div>

        <div class="sidebar">
            <h3>Бічна панель</h3>
            <p>Додаткова інформація або навігація</p>
        </div>

        <div class="footer">
            <p>&copy; 2024 Мій веб-сайт. Всі права
захищені.</p>
        </div>
    </div>
</body>
</html>

```





```
.clearfix::after {
    content: "";
    display: table;
    clear: both;
}

.main-content {
    float: left;
    width: 65%;
    background-color: #3498db;
    color: white;
    padding: 20px;
    margin-right: 5%;
    box-sizing: border-box;
}

.sidebar {
    float: right;
    width: 30%;
    background-color: #e74c3c;
    color: white;
    padding: 20px;
    box-sizing: border-box;
}

.nested-left {
    float: left;
    width: 48%;
    background-color: #2980b9;
    padding: 15px;
```

```
        margin-right: 4%;
        margin-top: 20px;
    }

    .nested-right {
        float: right;
        width: 48%;
        background-color: #2980b9;
        padding: 15px;
        margin-top: 20px;
    }

    .full-width {
        clear: both;
        background-color: #27ae60;
        color: white;
        padding: 20px;
        margin-top: 20px;
    }
</style>
</head>
<body>
    <div class="container">
        <h1>Приклад вкладених плаваючих блоків</h1>

        <div class="clearfix">
            <div class="main-content">
                <h2>Основний контент</h2>
                <p>Це основна область контенту, яка
займає більшу частину ширини сторінки.</p>
```

```

<div class="clearfix">
    <div class="nested-left">
        <h4>Вкладений лівий блок</h4>
        <p>Цей блок плаває зліва
всередині основного контенту.</p>
    </div>

    <div class="nested-right">
        <h4>Вкладений правий блок</h4>
        <p>Цей блок плаває справа
всередині основного контенту.</p>
    </div>
</div>

<div class="full-width">
    <h4>Повноширинний вкладений
блок</h4>
    <p>Цей блок займає всю ширину
батьківського елемента після очищення float.</p>
</div>
</div>

<div class="sidebar">
    <h3>Бічна панель</h3>
    <p>Додаткова інформація або навігаційні
елементи розміщуються тут.</p>
    <ul>
        <li>Пункт меню 1</li>
        <li>Пункт меню 2</li>
    </ul>

```

```

        <li>Пункт меню 3</li>
    </ul>
</div>
</div>
</div>
</body>
</html>

```

Важливим аспектом роботи з плаваючими блоками є правильне очищення float за допомогою класу clearfix або властивості clear. Без цього батьківські елементи можуть "схлопнутися", оскільки плаваючі елементи виходять з нормального потоку документа.

### 2.2.3 Вирівнювання стовпців за висотою

Одним із класичних завдань верстки є створення стовпців однакової висоти, незалежно від кількості контенту в кожному з них. Ця задача особливо актуальна при створенні карткових макетів або багатостолбцевих дизайнів, де візуальна симетрія має важливе значення.

Існує декілька методів досягнення однакової висоти стовпців. Розглянемо найпоширеніші підходи:

Метод з використанням display: table :

```

<!DOCTYPE html>
<html lang="uk">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Вирівнювання стовпців за висотою</title>
    <style>
        .table-container {
            display: table;

```

```
        width: 100%;
        max-width: 1000px;
        margin: 0 auto;
        border-spacing: 20px;
    }

    .table-cell {
        display: table-cell;
        background-color: #ecf0f1;
        padding: 20px;
        border: 1px solid #bdc3c7;
        vertical-align: top;
    }

    .column-1 {
        background-color: #3498db;
        color: white;
    }

    .column-2 {
        background-color: #e74c3c;
        color: white;
    }

    .column-3 {
        background-color: #2ecc71;
        color: white;
    }
</style>
</head>
```

```

<body>
  <h1 style="text-align: center;">Стовпці однакової
висоти</h1>

  <div class="table-container">
    <div class="table-cell column-1">
      <h3>Стовпець 1</h3>
      <p>Це короткий текст у першому стовпці.</p>
    </div>

    <div class="table-cell column-2">
      <h3>Стовпець 2</h3>
      <p>Це більш довгий текст у другому стовпці,
який займає декілька рядків та демонструє, як стовпці
вирівнюються за висотою найвищого елемента.</p>
      <p>Додатковий параграф для збільшення
висоти.</p>
    </div>

    <div class="table-cell column-3">
      <h3>Стовпець 3</h3>
      <p>Середній за розміром текст у третьому
стовпці.</p>
      <p>Ще один параграф.</p>
    </div>
  </div>
</body>
</html>

```

Сучасний підхід з Flexbox:

```
.flex-container {
```

```

    display: flex;
    max-width: 1000px;
    margin: 0 auto;
    gap: 20px;
}

.flex-column {
    flex: 1;
    background-color: #ecf0f1;
    padding: 20px;
    border: 1px solid #bdc3c7;
}

.flex-column:nth-child(1) { background-color: #3498db;
color: white; }
.flex-column:nth-child(2) { background-color: #e74c3c;
color: white; }
.flex-column:nth-child(3) { background-color: #2ecc71;
color: white; }

```

Flexbox є найсучаснішим та найзручнішим методом для вирівнювання елементів за висотою, оскільки це відбувається автоматично без додаткових налаштувань.

#### 2.2.4 Властивість display

Властивість display є однією з найважливіших у CSS, оскільки вона визначає тип відображення елемента та його поведінку в документі. Розуміння різних значень цієї властивості є критично важливим для створення складних макетів та контролю над розташуванням елементів на сторінці.

Основні значення display:

```
<!DOCTYPE html>
```





```
.inline-block-element {
    display: inline-block;
    background-color: #2ecc71;
    color: white;
    padding: 10px;
    margin: 5px;
    width: 150px;
}

.none-element {
    display: none;
}

.flex-container {
    display: flex;
    background-color: #9b59b6;
    padding: 10px;
    gap: 10px;
}

.flex-item {
    background-color: #8e44ad;
    color: white;
    padding: 15px;
    flex: 1;
}

.grid-container {
    display: grid;
    grid-template-columns: repeat(3, 1fr);
```

```

        gap: 10px;
        background-color: #f39c12;
        padding: 10px;
    }

    .grid-item {
        background-color: #e67e22;
        color: white;
        padding: 15px;
        text-align: center;
    }
</style>
</head>
<body>
    <h1>Демонстрація властивості display</h1>

    <div class="demo-container">
        <h3>display: block</h3>
        <div class="block-element">Блоковий елемент
1</div>
        <div class="block-element">Блоковий елемент
2</div>
        <div class="block-element">Блоковий елемент
3</div>
    </div>

    <div class="demo-container">
        <h3>display: inline</h3>
        <span class="inline-element">Інлайн 1</span>
        <span class="inline-element">Інлайн 2</span>

```

```

    <span class="inline-element">Инлайн 3</span>
    <span class="inline-element">Инлайн 4</span>
</div>

```

```

<div class="demo-container">
    <h3>display: inline-block</h3>
    <div class="inline-block-element">Инлайн-блок
1</div>
    <div class="inline-block-element">Инлайн-блок
2</div>
    <div class="inline-block-element">Инлайн-блок
3</div>
</div>

```

```

<div class="demo-container">
    <h3>display: flex</h3>
    <div class="flex-container">
        <div class="flex-item">Flex элемент 1</div>
        <div class="flex-item">Flex элемент 2</div>
        <div class="flex-item">Flex элемент 3</div>
    </div>
</div>

```

```

<div class="demo-container">
    <h3>display: grid</h3>
    <div class="grid-container">
        <div class="grid-item">1</div>
        <div class="grid-item">2</div>
        <div class="grid-item">3</div>
        <div class="grid-item">4</div>
    </div>
</div>

```

```

        <div class="grid-item">5</div>
        <div class="grid-item">6</div>
    </div>
</div>
</body>
</html>

```

Кожне значення display має свої особливості:

- **block** - елемент займає всю доступну ширину та розташовується на новому рядку;
- **inline** - елемент займає лише необхідну ширину та розташовується в одному рядку з іншими inline елементами;
- **inline-block** - комбінує властивості inline та block, дозволяючи встановлювати ширину та висоту;
- **flex** - створює flex-контейнер для гнучкого розташування дочірніх елементів;
- **grid** - створює сітковий контейнер для двовимірного розташування елементів;
- **none** - повністю приховує елемент.

## 2.2.5 Створення навігаційної панелі

Навігаційна панель є одним із найважливіших елементів веб-сторінки, що забезпечує зручну навігацію по сайту. Сучасні навігаційні панелі повинні бути не лише функціональними, але й адаптивними, доступними та естетично привабливими.

Горизонтальна навігаційна панель:

```

<!DOCTYPE html>
<html lang="uk">
<head>
    <meta charset="UTF-8">

```

```
<meta name="viewport" content="width=device-width,
initial-scale=1.0">
<title>Навігаційна панель</title>
<style>
    * {
        margin: 0;
        padding: 0;
        box-sizing: border-box;
    }

    body {
        font-family: Arial, sans-serif;
    }

    .navbar {
        background-color: #2c3e50;
        padding: 0;
        position: relative;
    }

    .nav-container {
        max-width: 1200px;
        margin: 0 auto;
        display: flex;
        justify-content: space-between;
        align-items: center;
        padding: 0 20px;
    }

    .logo {
```

```
        color: white;
        font-size: 24px;
        font-weight: bold;
        text-decoration: none;
        padding: 15px 0;
    }

    .nav-menu {
        display: flex;
        list-style: none;
        margin: 0;
        padding: 0;
    }

    .nav-item {
        margin: 0;
    }

    .nav-link {
        display: block;
        color: white;
        text-decoration: none;
        padding: 20px 25px;
        transition: background-color 0.3s ease;
        position: relative;
    }

    .nav-link:hover {
        background-color: #34495e;
    }
```

```
.nav-link.active {
    background-color: #3498db;
}

/* Підменю */
.dropdown {
    position: relative;
}

.dropdown-menu {
    position: absolute;
    top: 100%;
    left: 0;
    background-color: #34495e;
    min-width: 200px;
    box-shadow: 0 8px 16px rgba(0,0,0,0.2);
    z-index: 1000;
    opacity: 0;
    visibility: hidden;
    transform: translateY(-10px);
    transition: all 0.3s ease;
    list-style: none;
    padding: 0;
    margin: 0;
}

.dropdown:hover .dropdown-menu {
    opacity: 1;
    visibility: visible;
```

```
        transform: translateY(0);
    }

.dropdown-menu .nav-link {
    padding: 15px 20px;
    border-bottom: 1px solid #2c3e50;
}

.dropdown-menu .nav-link:last-child {
    border-bottom: none;
}

.dropdown-menu .nav-link:hover {
    background-color: #2c3e50;
    padding-left: 30px;
}

/* Мобільне меню */
.hamburger {
    display: none;
    flex-direction: column;
    cursor: pointer;
    padding: 10px;
}

.hamburger span {
    width: 25px;
    height: 3px;
    background-color: white;
    margin: 3px 0;
```



```

        transition: 0.3s;
    }

    @media screen and (max-width: 768px) {
        .nav-menu {
            position: fixed;
            left: -100%;
            top: 70px;
            flex-direction: column;
            background-color: #2c3e50;
            width: 100%;
            text-align: center;
            transition: 0.3s;
            box-shadow: 0 10px 27px
rgba(0,0,0,0.05);
        }

        .nav-menu.active {
            left: 0;
        }

        .nav-item {
            margin: 0;
        }

        .nav-link {
            padding: 20px;
            display: block;
        }
    }

```

```

        .hamburger {
            display: flex;
        }

        .dropdown-menu {
            position: static;
            opacity: 1;
            visibility: visible;
            transform: none;
            box-shadow: none;
            background-color: #1a252f;
        }
    }
</style>
</head>
<body>
    <nav class="navbar">
        <div class="nav-container">
            <a href="#" class="logo">Мій Сайт</a>

            <ul class="nav-menu">
                <li class="nav-item">
                    <a href="#" class="nav-link
active">Головна</a>
                </li>
                <li class="nav-item dropdown">
                    <a href="#" class="nav-link">Про
нас</a>

                    <ul class="dropdown-menu">

```

```

        <li><a href="#" class="nav-
link">Історія компанії</a></li>
        <li><a href="#" class="nav-
link">Наша команда</a></li>
        <li><a href="#" class="nav-
link">Місія і цінності</a></li>
    </ul>
</li>
<li class="nav-item dropdown">
    <a href="#" class="nav-
link">Послуги</a>
    <ul class="dropdown-menu">
        <li><a href="#" class="nav-
link">Веб-розробка</a></li>
        <li><a href="#" class="nav-
link">Дизайн</a></li>
        <li><a href="#" class="nav-
link">Консультації</a></li>
        <li><a href="#" class="nav-
link">Підтримка</a></li>
    </ul>
</li>
<li class="nav-item">
    <a href="#" class="nav-
link">Портфоліо</a>
</li>
<li class="nav-item">
    <a href="#" class="nav-
link">Контакти</a>
</li>

```

```

        </ul>

        <div class="hamburger">
            <span></span>
            <span></span>
            <span></span>
        </div>
    </div>
</nav>

<script>
    // JavaScript для мобільного меню
    const hamburger =
document.querySelector('.hamburger');
    const navMenu = document.querySelector('.nav-
menu');

    hamburger.addEventListener('click', () => {
        hamburger.classList.toggle('active');
        navMenu.classList.toggle('active');
    });

    // Закриття меню при кліку на посилання
    document.querySelectorAll('.nav-
link').forEach(n => n.addEventListener('click', () => {
        hamburger.classList.remove('active');
        navMenu.classList.remove('active');
    }));
</script>
</body>

```

```
</html>
```

Цей приклад демонструє повнофункціональну навігаційну панель з підменю, адаптивним дизайном та JavaScript-функціональністю для мобільних пристроїв.

### 2.2.6 Вирівнювання плаваючих елементів

Вирівнювання плаваючих елементів є важливим аспектом створення складних макетів, особливо при роботі зі старими техніками верстки. Правильне розуміння механізмів вирівнювання допомагає створювати стабільні та передбачувані макети.

```
<!DOCTYPE html>
<html lang="uk">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <title>Вирівнювання плаваючих елементів</title>
  <style>
    .container {
      max-width: 1000px;
      margin: 0 auto;
      padding: 20px;
    }

    .section {
      margin-bottom: 40px;
      border: 2px solid #34495e;
      padding: 20px;
    }
```

```
/* Очищення float */
.clearfix::before,
.clearfix::after {
    content: "";
    display: table;
}

.clearfix::after {
    clear: both;
}

/* Базові стилі для плаваючих елементів */
.float-left {
    float: left;
    width: 30%;
    background-color: #3498db;
    color: white;
    padding: 15px;
    margin-right: 5%;
    margin-bottom: 20px;
}

.float-right {
    float: right;
    width: 30%;
    background-color: #e74c3c;
    color: white;
    padding: 15px;
    margin-left: 5%;
    margin-bottom: 20px;
}
```

```
}

.float-center {
    float: left;
    width: 30%;
    background-color: #2ecc71;
    color: white;
    padding: 15px;
    margin-right: 5%;
}

/* Вирівнювання по вертикалі */
.vertical-align {
    overflow: hidden; /* Створюємо новий
блоковий контекст форматування */
}

.tall-content {
    float: left;
    width: 48%;
    background-color: #9b59b6;
    color: white;
    padding: 20px;
    margin-right: 4%;
    min-height: 200px;
}

.short-content {
    float: right;
    width: 48%;
```

```

        background-color: #f39c12;
        color: white;
        padding: 20px;
    }

    /* Псевдо-вирівнювання по центру */
    .pseudo-center {
        text-align: center;
    }

    .centered-float {
        display: inline-block;
        background-color: #1abc9c;
        color: white;
        padding: 20px;
        margin: 10px;
        width: 200px;
    }

    /* Адаптивні плаваючі елементи */
    @media screen and (max-width: 768px) {
        .float-left,
        .float-right,
        .float-center {
            float: none;
            width: 100%;
            margin: 0 0 20px 0;
        }

        .tall-content,

```



```

        .short-content {
            float: none;
            width: 100%;
            margin: 0 0 20px 0;
        }
    }
</style>
</head>
<body>
    <div class="container">
        <h1>Вирівнювання плаваючих елементів</h1>

        <div class="section">
            <h2>Трьохколонковий макет</h2>
            <div class="clearfix">
                <div class="float-left">
                    <h4>Ліва колонка</h4>
                    <p>Контент лівої колонки з
плаваючим вирівнюванням зліва.</p>
                </div>

                <div class="float-center">
                    <h4>Центральна колонка</h4>
                    <p>Центральна колонка також
використовує float: left.</p>
                </div>

                <div class="float-right">
                    <h4>Права колонка</h4>
                </div>
            </div>
        </div>
    </div>
</body>
</html>

```

```

        <p>Контент правої колонки з
плаваючим вирівнюванням справа.</p>
    </div>
</div>
</div>

<div class="section">
    <h2>Вирівнювання по висоті</h2>
    <div class="vertical-align clearfix">
        <div class="tall-content">
            <h4>Високий блок</h4>
            <p>Цей блок має більше контенту та
займає більше місця по вертикалі. Lorem ipsum dolor sit
amet, consectetur adipiscing elit. Sed do eiusmod
tempor incididunt ut labore et dolore magna aliqua.</p>
            <p>Додатковий контент для
збільшення висоти блоку.</p>
        </div>

        <div class="short-content">
            <h4>Низький блок</h4>
            <p>Цей блок має менше контенту, але
завдяки очищенню float він вирівнюється коректно.</p>
        </div>
    </div>
</div>

<div class="section">
    <h2>Псевдо-центрування з inline-block</h2>
    <div class="pseudo-center">

```

```

        <div class="centered-float">
            <h4>Блок 1</h4>
            <p>Центрований блок</p>
        </div>
        <div class="centered-float">
            <h4>Блок 2</h4>
            <p>Центрований блок</p>
        </div>
    </div>
</div>
</body>
</html>

```

Ключові принципи вирівнювання плаваючих елементів включають правильне використання clearfix, розуміння того, як float впливає на потік документа, та застосування адаптивних технік для мобільних пристроїв.

### 2.2.7 Створення найпростішого макета

Створення базового макета веб-сторінки є фундаментальною навичкою, яка дозволяє швидко прототипувати та розробляти функціональні веб-інтерфейси. Найпростіший макет зазвичай включає заголовок, основний контент, бічну панель та підвал.

```

<!DOCTYPE html>
<html lang="uk">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Простий макет сторінки</title>
    <style>

```

```

* {
    margin: 0;
    padding: 0;
    box-sizing: border-box;
}

body {
    font-family: Arial, sans-serif;
    line-height: 1.6;
    color: #333;
    background-color: #f4f4f4;
}

.wrapper {
    max-width: 1200px;
    margin: 0 auto;
    background-color: white;
    box-shadow: 0 0 10px rgba(0,0,0,0.1);
    min-height: 100vh;
    display: flex;
    flex-direction: column;
}

/* Заголовок сайту */
.header {
    background: linear-gradient(135deg, #667eea
0%, #764ba2 100%);
    color: white;
    padding: 20px;
    text-align: center;

```

```
}

.header h1 {
    font-size: 2.5em;
    margin-bottom: 10px;
}

.header p {
    font-size: 1.2em;
    opacity: 0.9;
}

/* Навігаційне меню */
.navigation {
    background-color: #333;
    padding: 0;
}

.nav-list {
    list-style: none;
    display: flex;
    justify-content: center;
    flex-wrap: wrap;
}

.nav-item {
    margin: 0;
}

.nav-link {
```

```
        display: block;
        color: white;
        text-decoration: none;
        padding: 15px 20px;
        transition: background-color 0.3s ease;
    }

    .nav-link:hover,
    .nav-link.active {
        background-color: #555;
    }

    /* Основная область контента */
    .main-container {
        display: flex;
        flex: 1;
        gap: 20px;
        padding: 20px;
    }

    .main-content {
        flex: 2;
        background-color: white;
        padding: 20px;
        border-radius: 5px;
        box-shadow: 0 2px 5px rgba(0,0,0,0.1);
    }

    .main-content h2 {
        color: #333;
    }
```

```
        margin-bottom: 15px;
        border-bottom: 2px solid #667eea;
        padding-bottom: 10px;
    }

    .main-content p {
        margin-bottom: 15px;
        text-align: justify;
    }

    .article {
        background-color: #f8f9fa;
        padding: 20px;
        margin: 20px 0;
        border-left: 4px solid #667eea;
        border-radius: 0 5px 5px 0;
    }

    .article h3 {
        color: #555;
        margin-bottom: 10px;
    }

    /* Бічна панель */
    .sidebar {
        flex: 1;
        background-color: white;
        padding: 20px;
        border-radius: 5px;
        box-shadow: 0 2px 5px rgba(0,0,0,0.1);
    }
```

```
        height: fit-content;
    }

.sidebar h3 {
    color: #333;
    margin-bottom: 15px;
    padding-bottom: 10px;
    border-bottom: 2px solid #764ba2;
}

.sidebar-section {
    margin-bottom: 25px;
}

.sidebar ul {
    list-style: none;
}

.sidebar ul li {
    padding: 8px 0;
    border-bottom: 1px solid #eee;
}

.sidebar ul li:last-child {
    border-bottom: none;
}

.sidebar ul li a {
    color: #555;
    text-decoration: none;
```



```
        transition: color 0.3s ease;
    }

.sidebar ul li a:hover {
    color: #667eea;
}

.widget {
    background-color: #f8f9fa;
    padding: 15px;
    border-radius: 5px;
    margin-bottom: 20px;
}

.widget h4 {
    color: #333;
    margin-bottom: 10px;
}

/* Підвал */
.footer {
    background-color: #333;
    color: white;
    text-align: center;
    padding: 20px;
    margin-top: auto;
}

.footer-content {
    display: flex;
```

```
        justify-content: space-between;
        align-items: center;
        flex-wrap: wrap;
        gap: 20px;
    }

    .footer-links {
        list-style: none;
        display: flex;
        gap: 20px;
    }

    .footer-links a {
        color: #ccc;
        text-decoration: none;
        transition: color 0.3s ease;
    }

    .footer-links a:hover {
        color: white;
    }

    /* Адаптивний дизайн */
    @media screen and (max-width: 768px) {
        .main-container {
            flex-direction: column;
        }

        .nav-list {
            flex-direction: column;
        }
    }
}
```

```
}

.footer-content {
    flex-direction: column;
    text-align: center;
}

.header h1 {
    font-size: 2em;
}

}

@media screen and (max-width: 480px) {
    .wrapper {
        margin: 0;
        box-shadow: none;
    }

    .main-container {
        padding: 10px;
    }

    .main-content,
    .sidebar {
        padding: 15px;
    }

    .header {
        padding: 15px;
    }
}
```

```

    }
    </style>
</head>
<body>
    <div class="wrapper">
        <!-- Заголовок сайту -->
        <header class="header">
            <h1>Мій Веб-сайт</h1>
            <p>Приклад простого макета HTML
сторінки</p>
        </header>

        <!-- Навігаційне меню -->
        <nav class="navigation">
            <ul class="nav-list">
                <li class="nav-item"><a href="#"
class="nav-link active">Головна</a></li>
                <li class="nav-item"><a href="#"
class="nav-link">Про нас</a></li>
                <li class="nav-item"><a href="#"
class="nav-link">Послуги</a></li>
                <li class="nav-item"><a href="#"
class="nav-link">Портфоліо</a></li>
                <li class="nav-item"><a href="#"
class="nav-link">Блог</a></li>
                <li class="nav-item"><a href="#"
class="nav-link">Контакти</a></li>
            </ul>
        </nav>

```

```

<!-- Основна область контенту -->
<div class="main-container">
    <main class="main-content">
        <h2>Ласкаво просимо на наш сайт</h2>
        <p>Це приклад найпростішого макета веб-
сторінки, який демонструє основні принципи
структурування контенту. Макет включає всі необхідні
елементи: заголовок, навігацію, основний контент, бічну
панель та підвал.</p>

        <p>Даний макет використовує сучасні
CSS-технології, включаючи Flexbox для створення
адаптивного дизайну, який коректно відображається на
різних пристроях та розмірах екранів.</p>

        <div class="article">
            <h3>Переваги структурованого
макета</h3>

            <p>Правильно структурований макет
забезпечує хорошу користувацьку навігацію, зручність
читання контенту та професійний зовнішній вигляд.
Використання семантичних HTML5 елементів покращує
доступність та SEO-оптимізацію сайту.</p>
        </div>

        <div class="article">
            <h3>Адаптивний дизайн</h3>
            <p>Сучасні веб-сайти повинні
коректно відображатися на всіх пристроях. Цей макет
використовує медіа-запити для адаптації під різні

```

розміри екранів, забезпечуючи оптимальний користувацький досвід на мобільних телефонах, планшетах та настільних комп'ютерах.</p>

```

        </div>
    </main>

    <!-- Бічна панель -->
    <aside class="sidebar">
        <div class="sidebar-section">
            <h3>Навігація</h3>
            <ul>
                <li><a href="#">Головна
сторінка</a></li>
                <li><a href="#">Новини</a></li>
                <li><a href="#">Статті</a></li>
                <li><a
href="#">Галерея</a></li>
                <li><a
href="#">Завантаження</a></li>
            </ul>
        </div>

        <div class="widget">
            <h4>Останні новини</h4>
            <p>Тут можна розмістити короткі
анонси останніх новин або важливих подій.</p>
        </div>

        <div class="widget">
            <h4>Корисні посилання</h4>

```

```

        <ul>
            <li><a
href="#">Документація</a></li>
            <li><a
href="#">Тьюторіали</a></li>
            <li><a href="#">FAQ</a></li>
            <li><a
href="#">Підтримка</a></li>
        </ul>
    </div>
</aside>
</div>

<!-- Підвал сайту -->
<footer class="footer">
    <div class="footer-content">
        <p>&copy; 2024 Мій Веб-сайт. Всі права
захищені.</p>
        <ul class="footer-links">
            <li><a href="#">Політика
конфіденційності</a></li>
            <li><a href="#">Умови
користування</a></li>
            <li><a href="#">Карта
сайту</a></li>
        </ul>
    </div>
</footer>
</div>
</body>

```

```
</html>
```

Цей макет демонструє всі основні принципи створення структурованої веб-сторінки та може слугувати основою для більш складних проєктів.

### 2.2.8 Позиціювання

Позиціювання елементів є одним із найскладніших аспектів, який дозволяє точно контролювати розміщення елементів на сторінці. Розуміння різних типів позиціонування критично важливе для створення складних інтерактивних інтерфейсів.

Типи позиціонування:

```
<!DOCTYPE html>
<html lang="uk">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <title>Позиціювання елементів</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      margin: 0;
      padding: 20px;
      height: 200vh; /* Висота для демонстрації
скролінгу */
    }

    .demo-container {
      margin: 30px 0;
      padding: 20px;
      border: 2px solid #34495e;
```



```
        background-color: #ecf0f1;
        position: relative;
        height: 300px;
    }

    /* Static позиціювання (за замовчуванням) */
    .static-element {
        background-color: #3498db;
        color: white;
        padding: 15px;
        margin: 10px;
        width: 200px;
    }

    /* Relative позиціювання */
    .relative-element {
        position: relative;
        background-color: #e74c3c;
        color: white;
        padding: 15px;
        margin: 10px;
        width: 200px;
        top: 20px;
        left: 50px;
    }

    /* Absolute позиціювання */
    .absolute-element {
        position: absolute;
        background-color: #2ecc71;
```

```
        color: white;
        padding: 15px;
        width: 180px;
        top: 50px;
        right: 20px;
    }

    /* Fixed позиціювання */
    .fixed-element {
        position: fixed;
        background-color: #9b59b6;
        color: white;
        padding: 15px;
        width: 200px;
        top: 20px;
        right: 20px;
        z-index: 1000;
        border-radius: 5px;
        box-shadow: 0 4px 8px rgba(0,0,0,0.3);
    }

    /* Sticky позиціювання */
    .sticky-element {
        position: sticky;
        top: 10px;
        background-color: #f39c12;
        color: white;
        padding: 15px;
        margin: 10px 0;
        z-index: 100;
```

```
}

/* Z-index демонстрація */
.z-index-container {
    position: relative;
    height: 200px;
}

.z-low {
    position: absolute;
    background-color: #e67e22;
    color: white;
    padding: 20px;
    top: 20px;
    left: 20px;
    width: 150px;
    z-index: 1;
}

.z-medium {
    position: absolute;
    background-color: #1abc9c;
    color: white;
    padding: 20px;
    top: 40px;
    left: 40px;
    width: 150px;
    z-index: 2;
}
```

```
.z-high {
    position: absolute;
    background-color: #c0392b;
    color: white;
    padding: 20px;
    top: 60px;
    left: 60px;
    width: 150px;
    z-index: 3;
}

/* Центрування абсолютно позиціонованого
елемента */
.centered-absolute {
    position: absolute;
    background-color: #8e44ad;
    color: white;
    padding: 20px;
    width: 200px;
    height: 100px;
    top: 50%;
    left: 50%;
    transform: translate(-50%, -50%);
    border-radius: 10px;
    text-align: center;
    display: flex;
    align-items: center;
    justify-content: center;
}
```

```
/* Модальне вікно */  
.modal-overlay {  
    position: fixed;  
    top: 0;  
    left: 0;  
    width: 100%;  
    height: 100%;  
    background-color: rgba(0,0,0,0.5);  
    z-index: 10000;  
    display: none;  
    align-items: center;  
    justify-content: center;  
}  
  
.modal-content {  
    background-color: white;  
    padding: 30px;  
    border-radius: 10px;  
    box-shadow: 0 10px 30px rgba(0,0,0,0.3);  
    max-width: 500px;  
    width: 90%;  
    position: relative;  
}  
  
.modal-close {  
    position: absolute;  
    top: 10px;  
    right: 15px;  
    font-size: 24px;  
    cursor: pointer;
```

```
        color: #666;
    }

    .modal-close:hover {
        color: #333;
    }

    /* Кнопка для відкриття модального вікна */
    .open-modal-btn {
        background-color: #3498db;
        color: white;
        border: none;
        padding: 12px 24px;
        border-radius: 5px;
        cursor: pointer;
        font-size: 16px;
        transition: background-color 0.3s ease;
    }

    .open-modal-btn:hover {
        background-color: #2980b9;
    }

    /* Tooltip */
    .tooltip-container {
        position: relative;
        display: inline-block;
        cursor: pointer;
        background-color: #34495e;
        color: white;
```

```
padding: 10px 15px;
border-radius: 5px;
margin: 20px;
}

.tooltip {
  position: absolute;
  bottom: 125%;
  left: 50%;
  transform: translateX(-50%);
  background-color: #2c3e50;
  color: white;
  padding: 8px 12px;
  border-radius: 4px;
  white-space: nowrap;
  opacity: 0;
  visibility: hidden;
  transition: opacity 0.3s ease, visibility
0.3s ease;
  z-index: 1000;
}

.tooltip::after {
  content: "";
  position: absolute;
  top: 100%;
  left: 50%;
  transform: translateX(-50%);
  border: 5px solid transparent;
  border-top-color: #2c3e50;
```

```

    }

    .tooltip-container:hover .tooltip {
        opacity: 1;
        visibility: visible;
    }
</style>
</head>
<body>
    <h1>Демонстрація позиціювання елементів</h1>

    <!-- Fixed елемент (залишається на екрані при
скролінгу) -->
    <div class="fixed-element">
        <strong>Fixed елемент</strong><br>
        Залишається на місці при скролінгу
    </div>

    <!-- Sticky елемент -->
    <div class="sticky-element">
        <strong>Sticky навігація</strong> - прилипає до
верху при скролінгу
    </div>

    <div class="demo-container">
        <h3>Static та Relative позиціювання</h3>
        <div class="static-element">
            Static позиціювання (за замовчуванням)
        </div>
        <div class="relative-element">

```



```

        Relative позиціювання (зсув на 20px вниз,
        50px праворуч)
    </div>

```

```

    <!-- Absolute елемент позиціонується відносно
    цього контейнера -->

```

```

    <div class="absolute-element">
        Absolute позиціювання відносно
        батьківського елемента
    </div>
</div>

```

```

<div class="demo-container">
    <h3>Z-index та шарування елементів</h3>
    <div class="z-index-container">
        <div class="z-low">Z-index: 1</div>
        <div class="z-medium">Z-index: 2</div>
        <div class="z-high">Z-index: 3</div>
    </div>
</div>

```

```

<div class="demo-container">
    <h3>Центрування абсолютно позиціонованого
    елемента</h3>
    <div class="centered-absolute">
        Ідеально відцентрований елемент
    </div>
</div>

```

```

<div style="margin: 40px 0;">

```

```

<h3>Інтерактивні елементи</h3>

<!-- Кнопка для відкриття модального вікна -->
<button class="open-modal-btn"
onclick="openModal()">Відкрити модальне вікно</button>

<!-- Tooltip -->
<div class="tooltip-container">
    Наведи на мене
    <div class="tooltip">Це tooltip з
позиціюванням!</div>
</div>
</div>

<!-- Контент для демонстрації скролінгу -->
<div style="margin-top: 50px;">
    <h3>Додатковий контент для демонстрації
скролінгу</h3>
    <p>Прокрутіть сторінку, щоб побачити поведінку
fixed та sticky елементів.</p>
    <p>Fixed елемент завжди залишається у правому
верхньому куті екрана, незалежно від позиції
скролінгу.</p>
    <p>Sticky елемент поводить ся як relative до
моменту, коли досягає встановленої позиції (top: 10px),
після чого "прилипає" до цієї позиції.</p>

    <!-- Багато тексту для створення скролінгу -->

```

```

        <div style="height: 500px; background: linear-
gradient(45deg, #f0f0f0, #e0e0e0); padding: 20px;
margin: 20px 0;">
            <h4>Область для скролінгу</h4>
            <p>Цей блок створений для демонстрації
поведінки різних типів позиціювання при скролінгу
сторінки.</p>
        </div>
    </div>

    <!-- Модальне вікно -->
    <div class="modal-overlay" id="modalOverlay"
onclick="closeModal(event)">
        <div class="modal-content">
            <span class="modal-close"
onclick="closeModal()">&times;</span>
            <h3>Модальне вікно</h3>
            <p>Це приклад модального вікна, створеного
з використанням position: fixed. Модальне вікно
повністю покриває екран та центрується за допомогою
Flexbox.</p>
            <p>Клікніть поза межами цього вікна або на
хрестик, щоб закрити його.</p>
        </div>
    </div>

    <script>
        function openModal() {

```

```

document.getElementById('modalOverlay').style.display =
'flex';

        document.body.style.overflow = 'hidden'; //
Забороняємо скролінг фону
    }

    function closeModal(event) {
        if (!event || event.target ===
document.getElementById('modalOverlay') ||
event.target.classList.contains('modal-close')) {

document.getElementById('modalOverlay').style.display =
'none';

        document.body.style.overflow = 'auto';
// Дозволяємо скролінг фону
        }
    }
</script>
</body>
</html>

```

Властивість `position` керує тим, як елемент розташовується відносно потоку документа і відносно інших елементів. Значення найчастіше зустрічаються: `static`, `relative`, `absolute`, `fixed`, `sticky`. Далі — детально.

1) `position: static` — за замовчуванням.

Що робить: елемент розташовується в нормальному потоці документа. Не реагує на властивості `top/left/right/bottom`.

Поведінка: елементи йдуть один за одним, займають звичайну ширину (блокові — всю доступну, рядкові — тільки свій контент).

```
<p style="position: static;">Я звичайний абзац (static)
— не зсувається властивостями top/left.</p>
```

Коли використовувати: за замовчуванням майже для всіх елементів; не потрібно явно писати `position: static`.

2) `position: relative` — зсув від початкової позиції.

Що робить: елемент залишається в потоці, тобто місце під нього зберігається, але його візуальне положення можна зсунути `top/left/right/bottom`. Також `relative` часто використовується, щоб зробити елемент контейнером позиціонування для його абсолютно позиціонованих нащадків.

```
<div style="background:#eee; padding:10px;">
  <p style="position: relative; left: 20px; top:
10px;">
```

Я зсунувся праворуч на 20px і вниз на 10px, але місце в потоці залишилось.

```
</p>
</div>
```

У цьому прикладі сусідні елементи все одно "бачать" початкове місце для параграфа.

```
<div style="position: relative; width: 300px; height:
150px; background:#f8f8f8;">
  <div style="position: absolute; right: 10px; bottom:
10px; background:#c8c8c8;">
```

```
    Я прижаний до правого-нижнього кута батька
  </div>
</div>
```

Тут `relative` у батька створює базу для `position: absolute` у дитини.

Коли використовувати: коли треба локально зсунути елемент без порушення потоку або створити контекст позиціонування.

3) `position: absolute` — повне вилучення з потоку.

Що робить: елемент вилучається з нормального потоку (інші елементи поведуться так, ніби його немає) і позиціюється відносно найближчого батька, який має `position` відмінний від `static` (тобто `relative`, `absolute`, `fixed`, або `sticky`). Якщо такого предка немає — позиціюється відносно початкового контейнера (у більшості випадків — `html/body`).

```
<div style="position: relative; width: 400px; height: 200px; background:#fafafa;">
  
  <span style="position: absolute; top: 8px; right: 8px; background: red; color:#fff; padding:4px 8px;">
    Новинка
  </span>
</div>
```

`span` буде прижаний до правого верхнього кута блоку-родича.

Підсумок властивостей:

- не займає місця в потоці;
- координати (`top/left/...`) відносяться до контейнера позиціювання.

Підводні камені: якщо ви очікуєте, що `absolute` відноситься до сторінки (`viewport`), але існує проміжний батько з `position: relative`, координати вже від нього — це класична помилка при верстці.

4) `position: fixed` — прив'язка до `viewport`.

Що робить: елемент позиціюється відносно вікна перегляду (`viewport`) і зазвичай залишається на місці при скролі сторінки. `top: 0; left: 0` — приклеїть елемент до верхнього лівого кута вікна.

```
<header style="position: fixed; top: 0; left: 0; right: 0; height: 56px; background:#222; color:#fff;">
  Фіксоване меню
</header>
<main style="margin-top: 64px;">
```

Контент сторінки...

</main>

Тут `margin-top` потрібен, щоб основний контент не ховався під фіксованим `header`.

Особливі зауваження:

- Зазвичай фіксовані елементи знімаються з потоку, як `absolute`.
- **ВАЖЛИВО:** якщо якийсь предок має CSS-трансформацію (`transform: translateZ(0)`, `transform: rotate(...)`) або деякі інші властивості (`filter`, `perspective`, `will-change`), то в сучасних браузерах `position: fixed` може поводитися як `position: absolute` відносно цього предка (тобто не від `viewport`). Це часте джерело плутанини.
- На мобільних пристроях фіксовані елементи можуть поводитися по-різному через UI браузера (адресний рядок), тому тестуйте.

5) `position: sticky` — "прилипання" (комбінація `relative` + `fixed`).

Що робить: за замовчуванням елемент поводить себе як `relative` у потоці, але коли його позиція досягає заданого порога вікна (наприклад, `top: 0`), він починає поводитися як `fixed` і "прилипає" до цієї позиції; коли батьківський контейнер прокручується далі, елемент звільняється (не прилипає за межами батька).

Ключові умови:

- Потрібно задати `top/bottom/left/right` щоб `sticky` спрацював (наприклад, `top: 0`).
- Поведінка `sticky` відбувається відносно найближчого прокручуваного предка (найчастіше — `viewport`, якщо предки не мають `overflow`).
- Якщо предок має `overflow` відмінний від `visible`, то область, у межах якої прилипає елемент, — це цей предок.

<article>

```

<h2 style="position: sticky; top: 0; background:
white; border-bottom: 1px solid #ddd;">
    Заголовок секції (прилипає)
</h2>
<p>Довгий вміст...</p>
<!-- при прокрутці заголовок прилипає до верху
viewport доки не закінчиться article -->
</article>

```

Коли використовувати: для "липких" навігацій, заголовків таблиць, побічних меню, які повинні бути видимі під час прокрутки.

### Додаткові важливі теми

Як визначається контейнер позиціювання для absolute? Контейнером буде найближчий предок, у якого position не static. Тобто значення relative, absolute, fixed чи sticky у предка роблять його відправною точкою. Якщо такого немає — використовується корінь (зазвичай html/body).

z-index і stacking context (порядок шарів):

- z-index керує видимістю елементів по осі Z (хто спереду). Зазвичай z-index працює для елементів з position відмінним від static (також для деяких flex/grid елементів).
- Деякі CSS-властивості створюють stacking context (наприклад, позиціювання з z-index, opacity < 1, transform, filter, mix-blend-mode, will-change тощо). Елементи в різних stacking context не змішуються за z-index простим способом. Це джерело багатьох несподіваних проблем, коли z-index не працює як очікується.

Побутові шаблони (корисні приклади):

Центрування модалки

```

.modal {
    position: fixed;
    top: 50%;
    left: 50%;

```



```
transform: translate(-50%, -50%);
}
```

Бейджик у кутку з абсолютот:

батько `position: relative`, дитина `position: absolute; top: 8px; right: 8px`.

Sticky колонка у довгому контенті:

```
.sidebar {
  position: sticky;
  top: 1rem;
}
```

Сайдбар прилипає, але тільки поки не закінчиться батьківський контейнер.

Поради щодо продуктивності та анімацій:

- Для анімацій переважно змінювати `transform` та `opacity` — це GPU-оптимізовано і не викликає повного рефлоу. Зсув `top/left` часто призводить до перерахунку `layout` і менш продуктивний.
- Велика кількість елементів з `position: fixed/absolute` може ускладнити рендер, але сам по собі не є проблемою; важливо тестувати в реальних браузерях.

Типові помилки та як їх уникати:

- Очікувати, що `position: absolute` прив'язується до сторінки, але у вас є `position: relative` у проміжку — перевірте предків.
- `position: fixed` не прилипає у браузері мобільному як на десктопі через UI браузера — перевіряйте поведінку.
- `position: sticky` не працює, якщо не заданий поріг (`top / bottom`) або якщо контейнер-родич не має достатньої висоти.
- `z-index` «не працює» якщо елементи знаходяться у різних `stacking context` — у таких випадках треба подивитися на `transform`, `opacity`, `position` у предків.

Коротке резюме:

- `static` — нормальний потік; нічого не зсувається.
- `relative` — зберігає місце в потоці, можна зсунути; часто використовується як контейнер для `absolute`.
- `absolute` — вилучається з потоку і прив'язується до найпершого позиціонованого предка.
- `fixed` — прив'язується до `viewport` (але трансформовані предки можуть змінити поведінку).
- `sticky` — починає як `relative`, але «прилипає» до вказаної позиції, поки її контейнер не завершиться.

## 2.3 Основи CSS

Що таке CSS і навіщо він потрібен?

CSS (Cascading Style Sheets) - це мова стилів, яка працює разом з HTML для створення веб-сторінок. Щоб краще зрозуміти роль CSS, уявіть собі будівництво будинку:

HTML - це як каркас будинку, фундамент, стіни, кімнати. Він визначає структуру і зміст

CSS - це як дизайн інтер'єру, фарба на стінах, меблі, освітлення. Він визначає, як все це виглядає

Приклад без CSS vs з CSS:

Тільки HTML (без стилів):

```
<h1>Ласкаво просимо на мій сайт</h1>
<p>Це мій перший сайт. Тут буде цікава інформація.</p>
<a href="contact.html">Зв'язатися зі мною</a>
```

Результат: простий чорний текст на білому фоні, стандартний шрифт браузера, все виглядає дуже просто і непривабливо.

HTML + CSS:

```
<h1>Ласкаво просимо на мій сайт</h1>
<p>Це мій перший сайт. Тут буде цікава інформація.</p>
```

```
<a href="contact.html">Зв'язатися зі мною</a>
```

```
h1 {
    color: #2c3e50;
    font-family: 'Arial', sans-serif;
    text-align: center;
    background: linear-gradient(45deg, #3498db,
#2980b9);
    color: white;
    padding: 20px;
    border-radius: 10px;
}

p {
    font-size: 18px;
    line-height: 1.6;
    color: #34495e;
    max-width: 600px;
    margin: 20px auto;
}

a {
    background-color: #e74c3c;
    color: white;
    padding: 12px 24px;
    text-decoration: none;
    border-radius: 5px;
    display: inline-block;
    transition: background-color 0.3s;
}
```

```
a:hover {
    background-color: #c0392b;
}
```

Результат: красивий градієнтний заголовок, стилізований текст, привабливе посилання-кнопка з ефектом при наведенні.

Основні переваги CSS:

1. Розділення змісту і презентації

Що це означає?

Зміст (HTML) - це текст, зображення, посилання, структура сторінки

Презентація (CSS) - це кольори, шрифти, розміщення, анімації

Чому це важливо?

Погана практика (застаріла):

```
<font color="red" size="5">
    <center>
        <b>Важливий заголовок</b>
    </center>
</font>
<p bgcolor="yellow">Цей текст на жовтому фоні</p>
```

Хороша практика (сучасна):

```
<!-- HTML - тільки структура -->
<h1 class="important-title">Важливий заголовок</h1>
<p class="highlighted-text">Цей текст на жовтому
фоні</p>
```

```
/* CSS - тільки стилі */
.important-title {
    color: red;
    font-size: 2em;
    text-align: center;
```

```

    font-weight: bold;
}

.highlighted-text {
    background-color: yellow;
}

```

Переваги цього підходу:

- HTML стає чистішим і зрозумілішим;
- Легше змінювати дизайн, не торкаючись HTML;
- Кращий SEO (пошукові системи краще розуміють структуру).

2. Можливість застосування одного стилю до багатьох сторінок.

Як це працює? Уявіть, що у вас сайт з 100 сторінок і ви хочете змінити колір всіх заголовків.

Без CSS (жахливий сценарій):

```

<!-- Сторінка 1 -->
<h1 style="color: blue;">Заголовок сторінки 1</h1>

<!-- Сторінка 2 -->
<h1 style="color: blue;">Заголовок сторінки 2</h1>

<!-- ...і так 100 разів! -->

```

Щоб змінити колір, треба відредагувати 100 файлів!

З CSS (ідеальний сценарій):

```

<!-- Всі 100 сторінок посилаються на один CSS файл -->
<link rel="stylesheet" href="styles.css">
<h1>Заголовок сторінки</h1>

```

```

/* styles.css - один файл для всіх сторінок */
h1 {

```

```

    color: blue; /* Щоб змінити колір, редагуємо тільки
тут! */
}

```

### Реальний приклад:

```

/* Стили для всього сайту інтернет-магазину */
.product-card {
    border: 1px solid #ddd;
    padding: 20px;
    border-radius: 8px;
    background: white;
    box-shadow: 0 2px 4px rgba(0,0,0,0.1);
}

.price {
    color: #e74c3c;
    font-size: 24px;
    font-weight: bold;
}

.buy-button {
    background: #27ae60;
    color: white;
    padding: 12px 24px;
    border: none;
    border-radius: 5px;
    cursor: pointer;
}

```

Ці стилі можна використати на сторінці каталогу, сторінці товару, в корзині - скрізь!

### 3. Зменшення розміру коду

Приклад порівняння:

Без зовнішнього CSS (повторення стилів):

```
<!-- index.html -->
<style>
    .header { background: blue; color: white; padding:
20px; }
    .button { background: green; color: white; padding:
10px; }
</style>

<!-- about.html -->
<style>
    .header { background: blue; color: white; padding:
20px; }
    .button { background: green; color: white; padding:
10px; }
</style>

<!-- contact.html -->
<style>
    .header { background: blue; color: white; padding:
20px; }
    .button { background: green; color: white; padding:
10px; }
</style>
```

З зовнішнім CSS (стили завантажуються один раз):

```
<!-- Всі HTML файли -->
<link rel="stylesheet" href="styles.css">
```

```
/* styles.css - завантажується браузером один раз і
кешується */
.header { background: blue; color: white; padding:
20px; }
.button { background: green; color: white; padding:
10px; }
```

Результат: Швидше завантаження сайту, менше трафіку, краща продуктивність.

#### 4. Легкість підтримки та оновлення дизайну.

Сценарій: Замовник каже: "Хочу змінити дизайн сайту - зробити його в темних тонах"

#### З правильно організованим CSS:

```
/* Змінюємо тільки ці кілька рядків */
:root {
    --primary-color: #2c3e50;      /* було: #3498db */
    --background: #34495e;         /* було: #ffffff */
    --text-color: #ecf0f1;         /* було: #333333 */
}

body {
    background-color: var(--background);
    color: var(--text-color);
}

.button {
    background-color: var(--primary-color);
}
```

І весь сайт автоматично стає темним!

Без CSS або з поганим CSS: треба змінювати кожен елемент окремо, витратити дні на роботу, ризикувати зламати дизайн.



Підсумок: CSS - це не просто "прикрашалки" для сайту. Це потужний інструмент, який:

- робить веб-розробку більш організованою;
- економить час розробника;
- покращує користувацький досвід;
- дозволяє створювати професійні, сучасні сайти;
- спрощує підтримку великих проектів.

Без CSS сучасний інтернет був би схожий на документи Word 1995 року - функціональні, але абсолютно не привабливі.

### 2.3.1 Додавання CSS до веб-сторінки

Існує три основні способи підключення CSS до HTML-документа. Розглянемо кожен з них детально.

#### 1. Внутрішній CSS (Internal CSS)

Внутрішній CSS записується всередині тегу `<style>` в секції `<head>` HTML-документа.

Приклад:

```
<!DOCTYPE html>
<html>
<head>
  <style>
    body {
      font-family: Arial, sans-serif;
      background-color: #f0f0f0;
    }

    h1 {
      color: #333;
      text-align: center;
```

```

    }

    p {
        font-size: 16px;
        line-height: 1.5;
    }
</style>
</head>
<body>
    <h1>Заголовок сторінки</h1>
    <p>Це параграф тексту.</p>
</body>
</html>

```

Переваги внутрішнього CSS:

1. Простота використання для невеликих проектів

Що це означає? Для маленьких проектів внутрішній CSS дуже зручний, тому що не потрібно створювати додаткові файли.

Коли це корисно:

Приклад 1: Невелика лендінг сторінка

```

<!DOCTYPE html>
<html>
<head>
    <title>Моя візитка</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            max-width: 600px;
            margin: 0 auto;
            padding: 20px;

```

```
        background: linear-gradient(135deg, #667eea
0%, #764ba2 100%);
        color: white;
    }

    .profile-card {
        background: rgba(255, 255, 255, 0.1);
        padding: 30px;
        border-radius: 15px;
        text-align: center;
        backdrop-filter: blur(10px);
    }

    .contact-btn {
        background: #ff6b6b;
        color: white;
        padding: 12px 24px;
        border: none;
        border-radius: 25px;
        cursor: pointer;
        font-size: 16px;
    }
</style>
</head>
<body>
    <div class="profile-card">
        <h1>Іван Петренко</h1>
        <p>Веб-розробник</p>
        <button class="contact-btn">Зв'язатися</button>
    </div>
```

```
</body>  
</html>
```

Чому тут зручно використати внутрішній CSS:

- всього одна сторінка;
- не потрібно створювати окремий .css файл;
- всі стилі одразу видно в коді;
- легко поділитися з кимось (один файл замість двох).

Приклад 2: Швидкий прототип або тест

```
<!-- Тестуємо новий дизайн кнопки -->  
<!DOCTYPE html>  
<html>  
<head>  
  <style>  
    .test-button {  
      background: #3498db;  
      color: white;  
      border: none;  
      padding: 15px 30px;  
      border-radius: 8px;  
      cursor: pointer;  
      transition: all 0.3s ease;  
      font-size: 16px;  
    }  
  
    .test-button:hover {  
      background: #2980b9;  
      transform: translateY(-2px);  
      box-shadow: 0 5px 15px rgba(0,0,0,0.3);  
    }  
  </style>
```

```

</head>
<body>
    <button class="test-button">Тестова кнопка</button>
</body>
</html>

```

## 2. Всі стилі знаходяться в одному файлі

Переваги цього:

Легко знайти і змінити стиль:

```

<!DOCTYPE html>
<html>
<head>
    <style>
        /* Я одразу бачу всі стилі тут, в тому ж файлі
        */

        .header {
            background-color: #2c3e50;
            color: white;
            padding: 20px;
            text-align: center;
        }

        .main-content {
            max-width: 800px;
            margin: 0 auto;
            padding: 20px;
        }

        .footer {
            background-color: #34495e;
            color: white;

```

```

        text-align: center;
        padding: 10px;
    }
</style>
</head>
<body>
    <header class="header">
        <h1>Мій блог</h1>
    </header>

    <main class="main-content">
        <p>Контент сторінки...</p>
    </main>

    <footer class="footer">
        <p>&copy; 2025 Мій блог</p>
    </footer>
</body>
</html>

```

Чому це зручно:

- не потрібно перемикатися між файлами;
- одразу видно зв'язок між HTML і стилями;
- легко копіювати і передавати (один файл).

Недоліки:

1. Не можна використовувати для декількох сторінок

Проблема дублювання:

Уявіть, що у вас сайт з 5 сторінок і ви використовуєте внутрішній CSS:

index.html:

```

<!DOCTYPE html>
<html>

```

```

<head>
  <style>
    body { font-family: Arial; background: #f5f5f5;
}

    .header { background: #333; color: white;
padding: 20px; }
    .nav { background: #444; }
    .nav a { color: white; padding: 10px; }
  </style>
</head>
<body>
  <div class="header">Головна сторінка</div>
  <nav class="nav">
    <a href="about.html">Про нас</a>
    <a href="contact.html">Контакти</a>
  </nav>
</body>
</html>

```

#### about.html:

```

<!DOCTYPE html>
<html>
<head>
  <style>
    /* ТОЧНО ТІ Ж САМІ СТИЛІ! ДУБЛЮВАННЯ! */
    body { font-family: Arial; background: #f5f5f5;
}

    .header { background: #333; color: white;
padding: 20px; }
    .nav { background: #444; }
    .nav a { color: white; padding: 10px; }

```

```

    </style>
</head>
<body>
    <div class="header">Про нас</div>
    <nav class="nav">
        <a href="index.html">Головна</a>
        <a href="contact.html">Контакти</a>
    </nav>
</body>
</html>

```

**contact.html:**

```

<!DOCTYPE html>
<html>
<head>
    <style>
        /* ЗНОВУ ТИ Ж САМИ СТИЛІ! */
        body { font-family: Arial; background: #f5f5f5;
        }

        .header { background: #333; color: white;
padding: 20px; }
        .nav { background: #444; }
        .nav a { color: white; padding: 10px; }
    </style>
</head>
<body>
    <div class="header">Контакти</div>
    <nav class="nav">
        <a href="index.html">Головна</a>
        <a href="about.html">Про нас</a>
    </nav>

```



```
</body>
</html>
```

Проблеми цього підходу:

- Жахлива підтримка: Якщо треба змінити колір header з #333 на #2c3e50, треба редагувати 5 файлів!
- Помилки: Легко забути змінити в одному з файлів
- Витрата часу: Багато копії-пейсту.

Краще рішення з зовнішнім CSS:

```
<!-- Всі HTML файли -->
<link rel="stylesheet" href="styles.css">
```

```
/* styles.css - один файл для всіх сторінок */
body { font-family: Arial; background: #f5f5f5; }
.header { background: #333; color: white; padding:
20px; }
.nav { background: #444; }
.nav a { color: white; padding: 10px; }
```

## 2. Збільшує розмір HTML-файлу

Практичний приклад:

Файл з внутрішнім CSS (великий):

```
<!DOCTYPE html>
<html>
<head>
  <title>Інтернет-магазин</title>
  <style>
    /* 200+ рядків CSS коду */
    * { margin: 0; padding: 0; box-sizing: border-
box; }

    body {
```

```
        font-family: 'Roboto', -apple-system,
BlinkMacSystemFont, sans-serif;
        line-height: 1.6;
        color: #333;
        background-color: #f8f9fa;
    }

    .container {
        max-width: 1200px;
        margin: 0 auto;
        padding: 0 20px;
    }

    .header {
        background: linear-gradient(135deg, #667eea
0%, #764ba2 100%);
        color: white;
        padding: 1rem 0;
        box-shadow: 0 2px 10px rgba(0,0,0,0.1);
    }

    .nav {
        display: flex;
        justify-content: space-between;
        align-items: center;
    }

    .nav-links {
        display: flex;
        list-style: none;
```

```
        gap: 2rem;
    }

    .nav-links a {
        color: white;
        text-decoration: none;
        font-weight: 500;
        transition: opacity 0.3s ease;
    }

    .nav-links a:hover { opacity: 0.8; }

    .hero {
        background: linear-
gradient(rgba(0,0,0,0.4), rgba(0,0,0,0.4)),
            url('hero-bg.jpg')
center/cover;
        height: 60vh;
        display: flex;
        align-items: center;
        text-align: center;
        color: white;
    }

    .product-grid {
        display: grid;
        grid-template-columns: repeat(auto-fit,
minmax(300px, 1fr));
        gap: 2rem;
        padding: 2rem 0;
```

```

    }

    .product-card {
        background: white;
        border-radius: 12px;
        overflow: hidden;
        box-shadow: 0 4px 20px rgba(0,0,0,0.1);
        transition: transform 0.3s ease, box-shadow
0.3s ease;
    }

    .product-card:hover {
        transform: translateY(-5px);
        box-shadow: 0 8px 30px rgba(0,0,0,0.15);
    }

    /* ... ще 100+ рядків CSS ... */
</style>
</head>
<body>
    <!-- HTML контент сторінки -->
    <div class="container">
        <h1>Товари магазину</h1>
        <div class="product-grid">
            <!-- товари -->
        </div>
    </div>
</body>
</html>

```

Розмір файлу: ~15-20 KB

Той же файл з зовнішнім CSS:

```
<!DOCTYPE html>
<html>
<head>
  <title>Інтернет-магазин</title>
  <link rel="stylesheet" href="styles.css">
</head>
<body>
  <div class="container">
    <h1>Товари магазину</h1>
    <div class="product-grid">
      <!-- товари -->
    </div>
  </div>
</body>
</html>
```

Розмір HTML: ~2-3 KB

Розмір CSS: ~12-15 KB (завантажується окремо і кешується)

Переваги зовнішнього підходу:

- HTML завантажується швидше;
- CSS кешується браузером (не завантажується повторно);
- Інші сторінки сайту завантажуються ще швидше (CSS уже в кеші).

Коли використовувати внутрішній CSS?

Хороші випадки:

- Email-шаблони (багато поштових клієнтів не підтримують зовнішні CSS);
- Одноразові лендінг-сторінки;
- Швидкі прототипи;

- Демо-сторінки для показу клієнту;
- А/В тестування окремих елементів.

Погані випадки:

- Багатосторінкові сайти;
- Коли CSS більше 50-100 рядків;
- Корпоративні сайти;
- Інтернет-магазини;
- Блоги з багатьма статтями.

Золоте правило: Якщо ваш CSS більше 100 рядків або у вас більше однієї сторінки - використовуйте зовнішній CSS.

## 2. Зовнішній CSS (External CSS)

Найпопулярніший і рекомендований спосіб. CSS-код зберігається в окремому файлі з розширенням .css і підключається через тег <link>.

HTML-файл (index.html):

```
<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet" href="styles.css">
</head>
<body>
  <h1>Заголовок сторінки</h1>
  <p class="highlight">Це виділений параграф.</p>
  <p>Це звичайний параграф.</p>
</body>
</html>
```

CSS-файл (styles.css):

```
/* Основні стилі для body */
body {
```

```
    font-family: 'Segoe UI', Tahoma, Geneva, Verdana,  
sans-serif;  
    margin: 0;  
    padding: 20px;  
    background-color: #ffffff;  
}  
  
/* Стили для заголовків */  
h1 {  
    color: #2c3e50;  
    text-align: center;  
    border-bottom: 2px solid #3498db;  
    padding-bottom: 10px;  
}  
  
/* Стили для параграфів */  
p {  
    font-size: 18px;  
    line-height: 1.6;  
    margin-bottom: 15px;  
}  
  
/* Клас для виділених параграфів */  
.highlight {  
    background-color: #fff3cd;  
    border: 1px solid #ffeaa7;  
    padding: 10px;  
    border-radius: 5px;  
}
```

Переваги зовнішнього CSS:

## 1. Можна використовувати для багатьох HTML-сторінок

Як це працює на практиці: уявіть інтернет-магазин з такою структурою:

my-shop/

|   |              |                              |
|---|--------------|------------------------------|
|   |              |                              |
| — | index.html   | (головна сторінка)           |
| — | catalog.html | (каталог товарів)            |
| — | product.html | (сторінка товару)            |
| — | cart.html    | (к о ш и к)                  |
| — | contact.html | (к о н т а к т и)            |
| — | styles.css   | (один файл стилів для всіх!) |

Один CSS файл для всього сайту:

```
/* styles.css - загальні стилі для всього сайту */

/* Базові стилі */
* {
    margin: 0;
    padding: 0;
    box-sizing: border-box;
}

body {
    font-family: 'Roboto', Arial, sans-serif;
    line-height: 1.6;
    color: #333;
    background-color: #f8f9fa;
}

/* Шапка сайту - використовується на всіх сторінках */
.site-header {
```



```
    background: linear-gradient(135deg, #667eea 0%,
#764ba2 100%);
    color: white;
    padding: 1rem 0;
    box-shadow: 0 2px 10px rgba(0,0,0,0.1);
}

.logo {
    font-size: 24px;
    font-weight: bold;
    text-decoration: none;
    color: white;
}

/* Навігація - на всіх сторінках */
.main-nav {
    display: flex;
    justify-content: space-between;
    align-items: center;
    max-width: 1200px;
    margin: 0 auto;
    padding: 0 20px;
}

.nav-links {
    display: flex;
    list-style: none;
    gap: 2rem;
}
```

```
.nav-links a {
    color: white;
    text-decoration: none;
    font-weight: 500;
    transition: opacity 0.3s ease;
}

.nav-links a:hover { opacity: 0.8; }

/* Контейнер - використовується скрізь */
.container {
    max-width: 1200px;
    margin: 0 auto;
    padding: 0 20px;
}

/* Кнопки - єдиний стиль по всьому сайту */
.btn {
    display: inline-block;
    padding: 12px 24px;
    background-color: #3498db;
    color: white;
    text-decoration: none;
    border-radius: 5px;
    border: none;
    cursor: pointer;
    font-size: 16px;
    transition: all 0.3s ease;
}
```

```
.btn:hover {
    background-color: #2980b9;
    transform: translateY(-2px);
}

.btn-success { background-color: #27ae60; }
.btn-success:hover { background-color: #229954; }

.btn-danger { background-color: #e74c3c; }
.btn-danger:hover { background-color: #c0392b; }

/* Картки товарів */
.product-card {
    background: white;
    border-radius: 12px;
    overflow: hidden;
    box-shadow: 0 4px 20px rgba(0,0,0,0.1);
    transition: transform 0.3s ease;
}

.product-card:hover {
    transform: translateY(-5px);
}

.product-image {
    width: 100%;
    height: 200px;
    object-fit: cover;
}
```

```
.product-info {
    padding: 20px;
}

.product-title {
    font-size: 18px;
    font-weight: bold;
    margin-bottom: 10px;
    color: #2c3e50;
}

.product-price {
    font-size: 24px;
    font-weight: bold;
    color: #e74c3c;
    margin-bottom: 15px;
}

/* Підвал - на всіх сторінках */
.site-footer {
    background-color: #2c3e50;
    color: white;
    text-align: center;
    padding: 30px 0;
    margin-top: 50px;
}
```

Тепер кожна HTML-сторінка просто посилається на цей файл:

index.html:

```
<!DOCTYPE html>
<html lang="uk">
```

```

<head>
    <meta charset="UTF-8">
    <title>Інтернет-магазин "Техноград"</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <header class="site-header">
        <nav class="main-nav">
            <a href="index.html"
class="logo">Техноград</a>
            <ul class="nav-links">
                <li><a
href="catalog.html">Каталог</a></li>
                <li><a href="cart.html">Кошик</a></li>
                <li><a
href="contact.html">Контакти</a></li>
            </ul>
        </nav>
    </header>

    <main class="container">
        <h1>Ласкаво просимо до Технограду!</h1>
        <a href="catalog.html" class="btn btn-
success">Переглянути товари</a>
    </main>

    <footer class="site-footer">
        <p>&copy; 2025 Техноград. Всі права
захищені.</p>
    </footer>

```

```
</body>
```

```
</html>
```

**catalog.html:**

```
<!DOCTYPE html>
```

```
<html lang="uk">
```

```
<head>
```

```
  <meta charset="UTF-8">
```

```
  <title>Каталог - Техноград</title>
```

```
  <link rel="stylesheet" href="styles.css">
```

```
</head>
```

```
<body>
```

```
  <header class="site-header">
```

```
    <nav class="main-nav">
```

```
      <a href="index.html"
```

```
class="logo">Техноград</a>
```

```
      <ul class="nav-links">
```

```
        <li><a
```

```
href="catalog.html">Каталог</a></li>
```

```
        <li><a href="cart.html">Кошик</a></li>
```

```
        <li><a
```

```
href="contact.html">Контакти</a></li>
```

```
      </ul>
```

```
    </nav>
```

```
  </header>
```

```
  <main class="container">
```

```
    <h1>Каталог товарів</h1>
```

```
    <div class="product-card">
```

```

        
        <div class="product-info">
            <h3 class="product-title">Ноутбук Dell
XPS 13</h3>
            <div class="product-price">25,999
грн</div>
            <button class="btn">Додати в
кошик</button>
        </div>
    </div>
</main>

<footer class="site-footer">
    <p>&copy; 2025 Техноград. Всі права
захищені.</p>
</footer>
</body>
</html>

```

Магія зовнішнього CSS:

- Всі сторінки виглядають однаково і професійно;
- Щоб змінити дизайн всього сайту, треба редагувати тільки styles.css;
- Новий розробник може швидко зрозуміти структуру стилів.

## 2. Зменшує розмір HTML-файлів

Порівняння розмірів файлів:

З внутрішнім CSS (один великий файл):

```

<!DOCTYPE html>
<html>
<head>
    <style>

```

```

        /* 500 рядків CSS коду тут! */
        body { font-family: Arial; background: #f5f5f5;
    }

        .header { background: linear-gradient(135deg,
    #667eea, #764ba2); }

        .nav { display: flex; justify-content: space-
    between; }

        .nav-links { display: flex; list-style: none;
    gap: 2rem; }

        .product-card { background: white; border-
    radius: 12px; }

        .product-card:hover { transform: translateY(-
    5px); }

        .btn { padding: 12px 24px; background: #3498db;
    }

        .btn:hover { background: #2980b9; }

        .container { max-width: 1200px; margin: 0 auto;
    }

        .site-footer { background: #2c3e50; color:
    white; }

        /* ... ще 490 рядків ... */
    </style>
</head>
<body>
    <!-- HTML контент -->
    <h1>Заголовок</h1>
    <p>Параграф</p>
</body>
</html>

```

Розмір файлу: ~25 KB



### З зовнішнім CSS:

```
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <h1>Заголовок</h1>
    <p>Параграф</p>
</body>
</html>
```

Розмір HTML: ~1 KB

Розмір styles.css: ~20 KB

### Реальний приклад з великим проектом:

Сайт інтернет-магазину з внутрішнім CSS:

```
└── index.html (30 KB - HTML + CSS)
└── catalog.html (35 KB - HTML + CSS)
└── product.html (28 KB - HTML + CSS)
└── cart.html (25 KB - HTML + CSS)
└── contact.html (22 KB - HTML + CSS)
```

Загальний розмір: 140 KB

Той же сайт з зовнішнім CSS:

```
└── index.html (3 KB - тільки HTML)
└── catalog.html (5 KB - тільки HTML)
└── product.html (4 KB - тільки HTML)
└── cart.html (3 KB - тільки HTML)
└── contact.html (2 KB - тільки HTML)
```

└── styles.css (20 KB - весь CSS)

Загальний розмір: 37 KB

Економія: 140 KB - 37 KB = 103 KB (майже в 4 рази менше!)

### 3. Легкість підтримки

Сценарій: Замовник хоче змінити кольорову схему сайту з синьої на зелену.

З зовнішнім CSS (5 хвилин роботи):

```
/* styles.css - змінюємо тільки тут */
:root {
    --primary-color: #27ae60;          /* було: #3498db */
    --primary-hover: #229954;          /* було: #2980b9 */
    --accent-color: #2ecc71;           /* було: #5dade2 */
}

/* Всі кнопки автоматично стануть зеленими */
.btn {
    background-color: var(--primary-color);
}

.btn:hover {
    background-color: var(--primary-hover);
}

/* Всі акцентні елементи теж */
.nav-link.active {
    color: var(--accent-color);
}
```

Результат: Весь сайт (50+ сторінок) миттєво стає зеленим!

Без зовнішнього CSS (2 дні роботи):

```
<!-- Треба змінювати в КОЖНОМУ файлі окремо -->
```

```

<!-- index.html -->
<style>
    .btn { background-color: #27ae60; } /* змінили */
</style>

<!-- catalog.html -->
<style>
    .btn { background-color: #27ae60; } /* змінили */
</style>

<!-- product.html -->
<style>
    .btn { background-color: #27ae60; } /* змінили */
</style>

<!-- ... і так для всіх 50 файлів! -->

```

**Інші приклади легкої підтримки:**

**Адаптивність для мобільних:**

```

/* styles.css - додаємо медіа-запити один раз */
@media (max-width: 768px) {
    .nav-links {
        flex-direction: column;
        gap: 1rem;
    }

    .product-card {
        margin-bottom: 20px;
    }

    .container {

```

```
padding: 0 10px;
}
}
```

Всі сторінки сайту автоматично стають адаптивними.

Додавання нових компонентів:

```
/* Додаємо новий компонент модального вікна */
.modal {
  position: fixed;
  top: 0;
  left: 0;
  width: 100%;
  height: 100%;
  background: rgba(0, 0, 0, 0.5);
  display: flex;
  justify-content: center;
  align-items: center;
}

.modal-content {
  background: white;
  padding: 30px;
  border-radius: 12px;
  max-width: 500px;
  width: 90%;
}
```

Тепер цей компонент можна використати на будь-якій сторінці.

#### 4. Кешування браузером покращує швидкість завантаження

Як працює кешування:

Перший візит користувача на сайт:

1. Браузер завантажує index.html (3 KB)

2. Браузер бачить `<link rel="stylesheet" href="styles.css">`

3. Браузер завантажує styles.css (20 KB)

4. Браузер зберігає styles.css в кеш

Загальний трафік: 23 KB

**Користувач переходить на catalog.html:**

1. Браузер завантажує catalog.html (5 KB)

2. Браузер бачить `<link rel="stylesheet" href="styles.css">`

3. Браузер каже: "У мене уже є цей файл в кеші!"

4. Браузер НЕ завантажує styles.css повторно

Трафік: тільки 5 KB

**Користувач переходить на product.html:**

1. Браузер завантажує product.html (4 KB)

2. CSS береться з кешу (0 KB трафіку)

Трафік: тільки 4 KB

**Порівняння з внутрішнім CSS:**

**З внутрішнім CSS (жахливо):**

Візит на index.html: 30 KB

Візит на catalog.html: 35 KB (весь CSS завантажується знову!)

Візит на product.html: 28 KB (весь CSS завантажується знову!)

Загальний трафік: 93 KB

**З зовнішнім CSS (ідеально):**

Візит на index.html: 23 KB (HTML + CSS)

Візит на catalog.html: 5 KB (тільки HTML)

Візит на product.html: 4 KB (тільки HTML)

Загальний трафік: 32 KB

**Економія трафіку: 93 KB - 32 KB = 61 KB (майже в 3 рази менше).**

Практичний вплив на швидкість:

Повільний мобільний інтернет (3G):

- З внутрішнім CSS: кожна сторінка завантажується 2-3 секунди
- З зовнішнім CSS: перша сторінка 2 секунди, наступні 0.5 секунди

Результат: Користувачі швидше переглядають сайт, менше відмовляються, більше купують.

Додаткові переваги зовнішнього CSS:

Командна робота

```
/* styles.css можуть редагувати кілька розробників */
/* designer.css - дизайнер працює тут */
@import url('components.css');
@import url('layouts.css');

/* components.css - frontend розробник */
.btn, .card, .modal { ... }

/* layouts.css - інший розробник */
.grid, .flex, .container { ... }
```

Версіонування

```
<!-- Можна легко оновлювати стилі -->
<link rel="stylesheet" href="styles-v2.3.css">
```

CDN та оптимізація

```
<!-- Можна використати CDN для швидшого завантаження -->
<link rel="stylesheet"
href="https://cdn.example.com/styles.min.css">
```

Таким чином, зовнішній CSS - це стандарт професійної веб-розробки тому, що:

- Один файл стилів для всього сайту;
- Економить час розробника і трафік користувача;

- Легко підтримувати і масштабувати;
- Браузери кешують CSS, прискорюючи завантаження;
- Дозволяє командам працювати ефективно.

Правило: Якщо у вас більше однієї сторінки або CSS більше 50 рядків - завжди використовуйте зовнішній CSS.

### 3. Вбудований CSS (Inline CSS)

Стили прописуються безпосередньо в HTML-тегах через атрибут style.

Приклад:

```
<!DOCTYPE html>
<html>
<body>
  <h1 style="color: red; text-align:
center;">Червоний заголовок</h1>
  <p style="font-size: 20px; color: blue; font-
weight: bold;">
    Синій жирний текст
  </p>
  <div style="background-color: yellow; padding:
10px; border: 2px solid green;">
    Жовтий блок з зеленою рамкою
  </div>
</body>
</html>
```

Коли використовувати inline CSS:

- Для швидкого тестування стилів;
- Коли потрібно застосувати унікальний стиль лише до одного елемента;
- В email-шаблонах (деякі поштові клієнти не підтримують зовнішні стилі).

Недоліки:

- Важко підтримувати;
- Збільшує розмір HTML;
- Порушує принцип розділення змісту і презентації.

### 2.3.2 Селектори

Селектори - це "адреси" елементів на веб-сторінці, до яких ми хочемо застосувати стилі. Розглянемо основні типи селекторів.

#### 1. Селектор типу (Type Selector)

Вибирає всі елементи певного типу.

```
/* Застосовується до всіх параграфів */
p {
    color: #333;
    font-size: 16px;
}

/* Застосовується до всіх заголовків h2 */
h2 {
    color: #2980b9;
    margin-top: 20px;
}

/* Застосовується до всіх посилань */
a {
    color: #e74c3c;
    text-decoration: none;
}

/* Ефект при наведенні на посилання */
a:hover {
```



```
text-decoration: underline;
}
```

## 2. Селектор класу (Class Selector)

Вибирає елементи з певним класом. Клас позначається крапкою перед назвою.

HTML:

```
<p class="important">Важливий текст</p>
<div class="container">Контейнер</div>
<span class="highlight important">Виділений важливий
текст</span>
```

CSS:

```
/* Селектор класу important */
.important {
    font-weight: bold;
    color: #e74c3c;
}

/* Селектор класу container */
.container {
    max-width: 1200px;
    margin: 0 auto;
    padding: 20px;
    background-color: #f8f9fa;
}

/* Селектор класу highlight */
.highlight {
    background-color: yellow;
    padding: 5px;
}
```

### 3. Селектор ідентифікатора (ID Selector)

Вибирає елемент з певним ID. ID позначається символом # перед назвою.

Пам'ятайте: ID має бути унікальним на сторінці!

HTML:

```
<header id="main-header">Головний заголовок сайту</header>
<nav id="navigation">Навігація</nav>
<footer id="site-footer">Підвал сайту</footer>
```

CSS:

```
/* Селектор ID для заголовка */
#main-header {
    background-color: #34495e;
    color: white;
    padding: 20px;
    text-align: center;
}

/* Селектор ID для навігації */
#navigation {
    background-color: #2c3e50;
    padding: 10px;
}

/* Селектор ID для підвалу */
#site-footer {
    background-color: #95a5a6;
    color: white;
    text-align: center;
    padding: 15px;
    margin-top: 50px;
}
```

}

#### 4. Комбіновані селектори

##### Селектор нащадків (Descendant Selector)

Вибирає елементи, що знаходяться всередині інших елементів.

HTML:

```
<div class="article">
  <h2>Заголовок статті</h2>
  <p>Перший параграф статті</p>
  <div class="sidebar">
    <p>Параграф в бічній панелі</p>
  </div>
</div>
```

CSS:

```
/* Всі параграфи всередині елементів з класом article */
.article p {
  font-size: 18px;
  line-height: 1.6;
}

/* Заголовки h2 всередині елементів з класом article */
.article h2 {
  color: #2c3e50;
  border-bottom: 1px solid #bdc3c7;
}

/* Параграфи всередині sidebar, що знаходиться в
article */
.article .sidebar p {
  font-size: 14px;
```

```
color: #7f8c8d;
font-style: italic;
}
```

### Дочірній селектор (Child Selector)

Вибирає лише безпосередні дочірні елементи.

HTML:

```
<ul class="menu">
  <li>Пункт 1
    <ul>
      <li>Підпункт 1.1</li>
      <li>Підпункт 1.2</li>
    </ul>
  </li>
  <li>Пункт 2</li>
</ul>
```

CSS:

```
/* Лише безпосередні дочірні li елементи menu */
.menu > li {
  font-weight: bold;
  padding: 10px;
  border-bottom: 1px solid #ecf0f1;
}

/* Вкладені li не будуть мати цей стиль */
.menu li li {
  font-weight: normal;
  font-size: 14px;
  padding: 5px;
}
```

## 5. Псевдокласи та псевдоелементи

### Псевдокласи (:hover, :focus, :active)

```
/* Зміна стилю при наведенні миші */
button:hover {
    background-color: #3498db;
    transform: translateY(-2px);
    transition: all 0.3s ease;
}

/* Стиль для елементів у фокусі */
input:focus {
    border: 2px solid #2980b9;
    outline: none;
    box-shadow: 0 0 5px rgba(41, 128, 185, 0.5);
}

/* Стиль для натиснутого елемента */
button:active {
    transform: translateY(0);
}
```

### Псевдоелементи (::before, ::after, ::first-line)

```
/* Додавання контенту перед елементом */
.quote::before {
    content: "";
    font-size: 2em;
    color: #95a5a6;
}

.quote::after {
    content: "";
    font-size: 2em;
```

```

        color: #95a5a6;
    }

    /* Стилiзацiя першого рядка параграфа */
    p::first-line {
        font-weight: bold;
        font-size: 1.2em;
    }

```

### 2.3.3 Принцип каскадування і специфічність правила

Каскадування - це основний принцип CSS, який визначає, які стилі застосовуються до елемента, коли існує кілька правил для одного і того ж елемента.

Специфічність - це "вага" селектора, яка визначає, яке правило буде застосоване. Специфічність обчислюється за такими правилами:

- Inline стилі: 1000 балів;
- ID селектори: 100 балів;
- Класи, атрибути, псевдокласи: 10 балів;
- Елементи та псевдоелементи: 1 бал.

Приклад обчислення специфічності:

```

/* Специфічність: 0001 (1 бал) */
p {
    color: blue;
}

/* Специфічність: 0010 (10 балів) */
.text {
    color: red;
}

```

```

/* Специфічність: 0100 (100 балів) */
#content {
    color: green;
}

/* Специфічність: 0111 (111 балів) */
#content .text p {
    color: orange;
}

/* Специфічність: 1000 (1000 балів) */
/* style="color: purple;" в HTML */

```

HTML для демонстрації:

```

<div id="content">
    <p class="text" style="color: purple;">
        Який колір буде у цього тексту?
    </p>
</div>

```

Текст буде фіолетовим, тому що inline стиль має найвищу специфічність (1000 балів).

Коли специфічність однакова, діє принцип "останнє правило виграє":

```

p {
    color: red;
}

p {
    color: blue;  /* Цей стиль буде застосований */
}

```

Правило !important дає найвищий пріоритет стилю:

```
p {
    color: blue !important; /* Завжди буде
застосований */
}

#content p {
    color: red; /* Буде проігнорований */
}
```

Увага! Використовуйте `!important` дуже обережно, тільки в крайніх випадках, оскільки це ускладнює підтримку коду.

Практичний приклад каскадування

HTML:

```
<div class="container">
    <article id="main-article" class="post featured">
        <h1 class="title">Заголовок статті</h1>
        <p class="intro">Вступний параграф</p>
        <p>Звичайний параграф</p>
    </article>
</div>
```

CSS:

```
/* Базові стилі для всіх параграфів (специфічність: 1)
*/
p {
    color: #333;
    font-size: 16px;
    line-height: 1.5;
}

/* Стилi для параграфів з класом intro (специфічність:
10) */
```



```
.intro {
    color: #2980b9;
    font-size: 18px;
    font-weight: bold;
}

/* Стили для параграфів всередині статті
(специфічність: 101) */
#main-article p {
    margin-bottom: 20px;
}

/* Стили для intro параграфів у featured постах
(специфічність: 20) */
.featured .intro {
    background-color: #ecf0f1;
    padding: 15px;
    border-left: 4px solid #3498db;
}
```

Результат:

- Звичайний параграф: колір #333, розмір 16px, margin-bottom 20px
- Intro параграф: колір #2980b9, розмір 18px, жирний, з фоном і відступами, margin-bottom 20px

### 2.3.4 Практичні поради та кращі практики

```
/* Коментарі для структурування */
/*
```

```
=====
=====
```

Базові стилі

```

=====
===== */

/* Скидання стилів */
* {
    margin: 0;
    padding: 0;
    box-sizing: border-box;
}

/*

=====

=====

Типографія

=====

===== */

body {
    font-family: -apple-system, BlinkMacSystemFont,
'Segoe UI', Roboto, sans-serif;
    line-height: 1.6;
    color: #333;
}

/*

=====

=====

Компоненти

```

```

=====
===== */

.button {
    display: inline-block;
    padding: 12px 24px;
    background-color: #3498db;
    color: white;
    text-decoration: none;
    border-radius: 4px;
    transition: background-color 0.3s ease;
}

.button:hover {
    background-color: #2980b9;
}

```

Найчастіші помилки:

- Занадто висока специфічність селекторів
- Зловживання !important
- Відсутність коментарів у коді
- Неправильна організація файлів

## 2.4 Трансформації, переходи та анімації

Сучасний веб-дизайн неможливо уявити без інтерактивних елементів, плавних переходів та ефектних анімацій. CSS надає розробникам потужний інструментарій для створення динамічних веб-сторінок без використання JavaScript. У цьому розділі ми детально розглянемо основи CSS-

трансформацій, переходів та анімацій, які дозволяють оживити статичні елементи сторінки.

Трансформації дають змогу змінювати зовнішній вигляд елементів: повертати, масштабувати, переміщувати та нахилити їх. Переходи забезпечують плавні зміни властивостей протягом певного часу. А анімації відкривають необмежені можливості для створення складних багатоетапних ефектів.

### 2.4.1 CSS-трансформації

#### Обертання

Обертання елементів реалізується за допомогою функції `rotate()` властивості `transform`. Кут обертання може задаватися в градусах (`deg`), радіанах (`rad`) або обертах (`turn`).

```
.rotate-element {
    transform: rotate(45deg);
    transition: transform 0.3s ease;
}

.rotate-element:hover {
    transform: rotate(180deg);
}
```

Для тривимірного обертання використовуються функції `rotateX()`, `rotateY()` та `rotateZ()`:

```
.card {
    width: 200px;
    height: 300px;
    background: linear-gradient(135deg, #667eea 0%,
    #764ba2 100%);
    transition: transform 0.6s;
}
```

```
.card:hover {
    transform: rotateY(180deg);
}

.card-3d {
    transform: rotateX(30deg) rotateY(45deg);
}
```

### Масштабування

Масштабування змінює розмір елемента за допомогою функцій `scale()`, `scaleX()` та `scaleY()`. Значення 1 відповідає оригінальному розміру, менше 1 - зменшує елемент, більше 1 - збільшує.

```
.scale-button {
    padding: 12px 24px;
    background-color: #3498db;
    border: none;
    border-radius: 6px;
    color: white;
    cursor: pointer;
    transition: transform 0.2s ease;
}

.scale-button:hover {
    transform: scale(1.1);
}

.scale-button:active {
    transform: scale(0.95);
}
```

```
/* Масштабування лише по одній осі */
.scale-x {
    transform: scaleX(1.5);
}

.scale-y {
    transform: scaleY(0.8);
}
```

### Переміщення

Функція `translate()` дозволяє переміщувати елементи без впливу на потік документа. Це особливо корисно для створення ефектів та позиціонування.

```
.slide-menu {
    width: 250px;
    height: 100vh;
    background-color: #2c3e50;
    position: fixed;
    left: 0;
    top: 0;
    transform: translateX(-100%);
    transition: transform 0.3s ease;
}

.slide-menu.active {
    transform: translateX(0);
}

/* Переміщення по окремих осях */
.move-right {
    transform: translateX(50px);
}
```

```
.move-up {
    transform: translateY(-30px);
}

/* Центрування елемента */
.centered {
    position: absolute;
    top: 50%;
    left: 50%;
    transform: translate(-50%, -50%);
}
```

### Нахил

Функції `skew()`, `skewX()` та `skewY()` створюють ефект нахилу елемента, деформуючи його форму.

```
.skew-card {
    width: 200px;
    height: 150px;
    background-color: #e74c3c;
    margin: 50px;
    transition: transform 0.3s ease;
}

.skew-card:hover {
    transform: skewX(15deg);
}

.diamond {
    transform: rotate(45deg) skew(15deg, 15deg);
}
```

```
.parallelogram {
    transform: skewX(-20deg);
}
```

### **Комбінування перетворень**

Найбільша сила CSS-трансформацій розкривається при їх комбінуванні. Важливо пам'ятати, що порядок застосування трансформацій має значення.

```
.complex-transform {
    width: 100px;
    height: 100px;
    background-color: #9b59b6;
    margin: 100px auto;
    transition: transform 0.5s ease;
}

.complex-transform:hover {
    transform: translateX(50px) rotate(45deg)
scale(1.2);
}

/* Різний порядок дає різний результат */
.order-1 {
    transform: rotate(45deg) translateX(50px);
}

.order-2 {
    transform: translateX(50px) rotate(45deg);
}
```

### **Початкова точка трансформації**



Властивість `transform-origin` визначає точку, навколо якої відбуваються трансформації. За замовчуванням це центр елемента (50% 50%).

```
.origin-demo {
    width: 100px;
    height: 100px;
    background-color: #f39c12;
    margin: 50px;
    transition: transform 0.3s ease;
}

.origin-top-left {
    transform-origin: top left;
}

.origin-top-left:hover {
    transform: rotate(45deg);
}

.origin-bottom-right {
    transform-origin: bottom right;
}

.origin-custom {
    transform-origin: 75% 25%;
}
```

### 2.4.2 CSS-переходи

Переходи дозволяють створити плавні зміни значень CSS-властивостей протягом певного часу. Це робить інтерфейс більш привабливим та зручним для користувача.

## Основи переходів

Властивість `transition` включає чотири компоненти: властивість, тривалість, функцію часу та затримку.

### 1. Властивість (`transition-property`)

Це визначає, яка саме CSS-властивість буде анімуватися.

```
/* Анімуємо тільки колір фону */  
transition-property: background-color;  
  
/* Анімуємо кілька властивостей */  
transition-property: width, height, background-color;  
  
/* Анімуємо ВСІ властивості, що можуть анімуватися */  
transition-property: all;
```

Приклад:

```
.button {  
    background-color: blue;  
    width: 100px;  
    transition-property: background-color; /* тільки  
колір буде анімуватися */  
}  
  
.button:hover {  
    background-color: red;  
    width: 200px; /* ця зміна буде миттєвою, без  
анімації */  
}
```

### 2. Тривалість (`transition-duration`)

Це визначає, скільки часу триватиме перехід від початкового до кінцевого стану.

```
transition-duration: 0.3s; /* 0.3 секунди */
```

```
transition-duration: 300ms;    /* 300 мілісекунд (те ж
came) */
transition-duration: 2s;      /* 2 секунди */
```

#### Приклад:

```
.fast { transition-duration: 0.1s; }    /* швидкий
перехід */
.normal { transition-duration: 0.3s; }  /* звичайний
перехід */
.slow { transition-duration: 2s; }      /* повільний
перехід */
```

### 3. Функція часу (transition-timing-function)

Це визначає, як швидко відбувається перехід на різних етапах анімації.

```
transition-timing-function: linear;      /* рівномірно
*/
transition-timing-function: ease;        /* повільно-
швидко-повільно */
transition-timing-function: ease-in;     /* починає
повільно */
transition-timing-function: ease-out;    /* закінчує
повільно */
transition-timing-function: ease-in-out; /* повільно на
початку і в кінці */

/* Власна функція Безьє */
transition-timing-function: cubic-bezier(0.25, 0.1,
0.25, 1);
```

#### Візуальна різниця:

```
.linear {
  transition-timing-function: linear;
  /* ____/---- - рівномірна швидкість */
```

```

}

.ease-in {
    transition-timing-function: ease-in;
    /* ____/---- - починає повільно, прискорюється */
}

.ease-out {
    transition-timing-function: ease-out;
    /* ____/---- - починає швидко, сповільнюється */
}

```

#### 4. Затримка (transition-delay)

Це визначає, скільки часу чекати перед початком переходу.

```

transition-delay: 0s;      /* починається відразу */
transition-delay: 0.5s;    /* чекає півсекунди */
transition-delay: 1s;      /* чекає секунду */

```

Як записати все разом:

Короткий запис:

```

/* transition: property duration timing-function delay;
*/
.element {
    transition: background-color 0.3s ease 0.1s;
}

```

Повний запис:

```

.element {
    transition-property: background-color;
    transition-duration: 0.3s;
    transition-timing-function: ease;
    transition-delay: 0.1s;
}

```

Для кількох властивостей:

```
.element {
  transition:
    background-color 0.3s ease 0s,
    width 0.5s linear 0.1s,
    height 0.4s ease-out 0.2s;
}
```

Практичний приклад:

```
.card {
  width: 200px;
  height: 150px;
  background-color: lightblue;
  transform: scale(1);
  opacity: 1;

  /* Різні переходи для різних властивостей */
  transition:
    background-color 0.3s ease 0s,          /* колір
змінюється швидко */
    transform 0.5s ease-out 0.1s,          /*
масштаб змінюється з затримкою */
    opacity 0.2s linear 0.3s;              /*
прозорість - в останню чергу */
}

.card:hover {
  background-color: coral;                  /* змінюється за 0.3с
без затримки */
  transform: scale(1.1);                    /* змінюється за 0.5с
з затримкою 0.1с */
}
```

```

    opacity: 0.8;                                /* змінюється за 0.2с з
затримкою 0.3с */
}

```

Результат: При наведенні миші колір зміниться відразу, потім почне збільшуватися розмір, а в кінці зміниться прозорість.

Ця система дає вам повний контроль над тим, як і коли відбуваються анімації на вашому сайті.

### **Анімація набору властивостей**

Можна анімувати практично будь-які числові CSS-властивості та деякі інші типи значень.

#### **1. Числові властивості**

Це найпростіший випадок - браузер може плавно змінювати числа від одного значення до іншого.

```

.animatable-numbers {
    /* Розміри */
    width: 100px → 200px;
    height: 50px → 100px;
    padding: 10px → 20px;
    margin: 5px → 15px;
    border-width: 1px → 5px;

    /* Позиціонування */
    top: 0px → 50px;
    left: 0px → 100px;
    z-index: 1 → 10;

    /* Шрифт */
    font-size: 16px → 24px;
}

```

```

line-height: 1.2 → 1.8;
letter-spacing: 0px → 2px;
word-spacing: 0px → 5px;

/* Прозорість */
opacity: 0 → 1;

/* Трансформації */
transform: translateX(0px) → translateX(100px);
transform: rotate(0deg) → rotate(360deg);
transform: scale(1) → scale(1.5);
}

```

#### Практичний приклад:

```

.growing-box {
    width: 50px;
    height: 50px;
    font-size: 12px;
    padding: 5px;
    transition: all 0.5s ease;
}

.growing-box:hover {
    width: 100px;    /* 50px → 100px */
    height: 100px;   /* 50px → 100px */
    font-size: 18px; /* 12px → 18px */
    padding: 15px;   /* 5px → 15px */
}

```

## 2. Кольори

Браузер може плавно змінювати кольори, інтерполюючи значення червоного, зеленого та синього каналів.

```
.color-transitions {
  /* Кольори тексту та фону */
  color: black → red;
  background-color: white → blue;
  border-color: gray → green;

  /* Градієнти (обмежено) */
  background: linear-gradient(red, blue) → linear-
gradient(green, yellow);

  /* Тіні з кольорами */
  box-shadow: 0 0 5px black → 0 0 10px red;
  text-shadow: 1px 1px blue → 2px 2px red;
}
```

Приклад з різними форматами кольорів:

```
.color-demo {
  /* Можна змішувати різні формати */
  background-color: rgb(255, 0, 0);      /* червоний
*/
  border-color: #00ff00;                  /* зелений */
  color: hsl(240, 100%, 50%);            /* синій */

  transition: all 0.3s ease;
}

.color-demo:hover {
```



```

    background-color: rgba(0, 255, 255, 0.8); /*
блакитний з прозорістю */
    border-color: orange;                      /*
помаранчевий */
    color: purple;                             /*
фіолетовий */
}

```

### 3. Відсотки та відносні одиниці

```

.percentage-animation {
    width: 50% → 100%;
    left: 10% → 90%;
    font-size: 1em → 2em;
    line-height: 120% → 180%;

    /* Viewport одиниці */
    width: 50vw → 80vw;
    height: 30vh → 60vh;
}

```

### 4. Спеціальні властивості

```

.special-properties {
    /* Фільтри */
    filter: blur(0px) → blur(5px);
    filter: brightness(100%) → brightness(150%);
    filter: contrast(100%) → contrast(200%);
    filter: hue-rotate(0deg) → hue-rotate(180deg);

    /* Clip-path (частково) */
    clip-path: circle(0%) → circle(50%);
}

```

```

/* SVG властивості */
stroke-width: 1 → 5;
stroke-dasharray: 0 → 100;
}

```

Що НЕ можна анімувати:

### 1. Текстові значення

```

/* ❌ НЕ працює */
.non-animatable {
    font-family: Arial → Times; /* миттєва зміна */
    text-align: left → center; /* миттєва зміна */
    position: static → absolute; /* миттєва зміна */
    display: block → flex; /* миттєва зміна */
}

```

### 2. Логічні властивості

```

/* ❌ НЕ працює */
.boolean-properties {
    visibility: visible → hidden; /* миттєва зміна */
    overflow: visible → hidden; /* миттєва зміна */
    white-space: normal → nowrap; /* миттєва зміна */
}

```

### 3. Складні значення

```

/* ❌ НЕ працює */
.complex-values {

```

```

    background-image: url(image1.jpg) →
url(image2.jpg);
    content: "Hello" → "World";
    cursor: pointer → crosshair;
}

```

### Приклад: Карточка товару

```

.product-card {
    width: 250px;
    height: 300px;
    background-color: white;
    border: 2px solid #ddd;
    border-radius: 8px;
    box-shadow: 0 2px 8px rgba(0,0,0,0.1);
    transform: translateY(0);
    opacity: 1;

    /* Анімуємо все, що можна */
    transition: all 0.3s cubic-bezier(0.4, 0, 0.2, 1);
}

.product-card:hover {
    /* Числові значення */
    width: 280px;           /*  плавно */
    height: 320px;          /*  плавно */
    border-width: 3px;      /*  плавно */

    /* Кольори */
    background-color: #f8f9fa; /*  плавно */
    border-color: #007bff;      /*  плавно */
}

```

```

    /* Тіні та трансформації */
    box-shadow: 0 8px 25px rgba(0,123,255,0.2); /* 
плавно */
    transform: translateY(-10px); /*
 плавно */

    /* Фільтри */
    filter: brightness(105%); /*  плавно */
}

```

Поради для оптимізації:

#### 1. Найшвидші для анімації:

```

.optimized {
    /* Ці властивості анімуються на GPU */
    transform: translateX(100px); /* найкраще */
    opacity: 0.5; /* найкраще */
    filter: blur(5px); /* добре */
}

```

#### 2. Повільніші (викликають перемальовку):

```

.slower {
    /* Ці властивості викликають перерахунок макета */
    width: 200px; /* повільніше */
    height: 200px; /* повільніше */
    padding: 20px; /* повільніше */
    margin: 20px; /* повільніше */
    top: 100px; /* повільніше */
    left: 100px; /* повільніше */
}

```

**Анімація набору властивостей**

Можна анімувати практично будь-які числові CSS-властивості та деякі інші типи значень.

```
.property-animations {  
    width: 200px;  
    height: 80px;  
    background: linear-gradient(90deg, #667eea,  
#764ba2);  
    border: 2px solid transparent;  
    box-shadow: 0 4px 8px rgba(0,0,0,0.1);  
    font-size: 16px;  
    color: white;  
    text-align: center;  
    line-height: 76px;  
    transition: all 0.4s cubic-bezier(0.4, 0, 0.2, 1);  
}  
  
.property-animations:hover {  
    width: 250px;  
    height: 100px;  
    border-color: #667eea;  
    box-shadow: 0 8px 25px rgba(102, 126, 234, 0.4);  
    font-size: 18px;  
    line-height: 96px;  
    transform: translateY(-5px);  
}
```

Функції анімації (timing functions) - це математичні функції, які визначають, як змінюється швидкість анімації протягом часу. Це один з найважливіших інструментів для створення природних та привабливих переходів.

Що таке функції часу? Уявіть собі автомобіль, який їде від точки А до точки Б:

- Linear: їде з постійною швидкістю;
- Ease-in: починає повільно, потім розганяється;
- Ease-out: починає швидко, потім гальмує;
- Ease-in-out: розганяється, потім гальмує.

#### 1. Linear - рівномірний рух

```
.linear-example {  
    transition: transform 2s linear;  
}  
  
.linear-example:hover {  
    transform: translateX(200px);  
}
```

Характеристика: Швидкість залишається постійною протягом всієї анімації.

Коли використовувати: Рідко, бо виглядає механічно. Підходить для обертання або технічних індикаторів.

#### 2. Ease (за замовчуванням)

```
.ease-example {  
    transition: transform 2s ease;  
    /* або просто transition: transform 2s; */  
}
```

Характеристика: Починає повільно, прискорюється в середині, сповільнюється в кінці.

Коли використовувати: Універсальний варіант для більшості переходів.

#### 3. Ease-in - розгін

```
.ease-in-example {  
    transition: transform 2s ease-in;  
}
```

Характеристика: Починає дуже повільно, потім поступово прискорюється.

Коли використовувати: Коли елемент "з'являється" або "вилітає" зі сторінки.

#### 4. Ease-out - гальмування

```
.ease-out-example {
  transition: transform 2s ease-out;
}
```

Характеристика: Починає швидко, потім поступово сповільнюється.

Коли використовувати: Коли елемент "приземляється" або "входить" на сторінку.

#### 5. Ease-in-out - розгін та гальмування

```
.ease-in-out-example {
  transition: transform 2s ease-in-out;
}
```

Характеристика: Повільний старт, прискорення, потім повільне завершення.

Коли використовувати: Для плавних, природних переходів між станами.

Кубічні криві Безьє (Cubic Bezier) - найпотужніший інструмент для створення власних функцій часу:

```
.custom-bezier {
  /* cubic-bezier(x1, y1, x2, y2) */
  transition: transform 1s cubic-bezier(0.68, -0.55,
0.265, 1.55);
}
```

Параметри:

- x1, y1: перша контрольна точка;
- x2, y2: друга контрольна точка;
- x завжди від 0 до 1 (час);
- y може бути будь-яким (швидкість).

### Популярні власні криві:

```
/* Пружний ефект */
.bouncy {
    transition: transform 0.6s cubic-bezier(0.68, -
0.55, 0.265, 1.55);
}

/* Швидкий старт */
.snappy {
    transition: transform 0.3s cubic-bezier(0.4, 0,
0.2, 1);
}

/* Драматичний */
.dramatic {
    transition: transform 1s cubic-bezier(0.25, 0.46,
0.45, 0.94);
}

/* Назад і вперед */
.back {
    transition: transform 0.5s cubic-bezier(0.175,
0.885, 0.32, 1.275);
}
```

### Приклад: Прогрес-бар з етапами

```
.progress-container {
    width: 300px;
    height: 8px;
    background: #e0e0e0;
    border-radius: 4px;
```



```
        overflow: hidden;
    }

    .progress-bar {
        height: 100%;
        width: 0%;
        background: linear-gradient(90deg, #4CAF50,
#45a049);
        border-radius: 4px;
    }

    /* Різні етапи з різними функціями */
    .progress-loading {
        transition: width 1s ease-in; /* розгін */
    }

    .progress-processing {
        transition: width 2s linear; /* рівномірно */
    }

    .progress-completing {
        transition: width 0.8s ease-out; /* гальмування */
    }

    .progress-success {
        transition: width 0.3s cubic-bezier(0.68, -0.55,
0.265, 1.55); /* пружний ефект */
    }
```

Коли використовувати які функції

### 1. Linear

- Обертання (спінери)
  - Технічні індикатори
  - Зміна кольорів у градієнтах
2. Ease (стандартна)
- Загальні hover-ефекти
  - Зміна розмірів
  - Базові переходи
3. Ease-in
- Елементи, що зникають
  - Початок складних анімацій
  - "Розгін" перед основною дією
4. Ease-out
- Елементи, що з'являються
  - Завершення анімацій
  - "Приземлення" елементів
5. Ease-in-out
- Переміщення між станами
  - Зміна позицій
  - Плавні трансформації
6. Cubic-bezier (власні)
- Унікальні ефекти
  - Фірмові анімації
  - Складні взаємодії

Поради для вибору функцій:

1. Для кнопок: ease-out або cubic-bezier(0.4, 0, 0.2, 1)
2. Для модальних вікон: cubic-bezier(0.34, 1.56, 0.64, 1) (пружний)
3. Для навігації: ease-in-out
4. Для індикаторів завантаження: linear
5. Для появи елементів: ease-out

6. Для зникнення елементів: ease-in

### Затримка переходу

Властивість transition-delay дозволяє затримати початок переходу:

```
.delayed-transitions {
  width: 50px;
  height: 50px;
  background-color: #3498db;
  margin: 5px;
  display: inline-block;
  transition: transform 0.5s ease;
}

.delay-0 { transition-delay: 0s; }
.delay-1 { transition-delay: 0.1s; }
.delay-2 { transition-delay: 0.2s; }
.delay-3 { transition-delay: 0.3s; }
.delay-4 { transition-delay: 0.4s; }

.container:hover .delayed-transitions {
  transform: translateY(-30px) rotate(360deg);
}
```

### Властивість transition

Повний синтаксис властивості transition:

```
.complete-syntax {
  /* transition: property duration timing-function
  delay; */
  transition: transform 0.3s cubic-bezier(0.4, 0,
  0.2, 1) 0.1s;

  /* Або окремо кожна властивість */
```

```

    transition-property: transform, opacity,
background-color;
    transition-duration: 0.3s, 0.5s, 0.2s;
    transition-timing-function: ease, ease-out, linear;
    transition-delay: 0s, 0.1s, 0.2s;
}

/* Перехід для всіх властивостей */
.all-properties {
    transition: all 0.3s ease;
}

```

### 2.4.3 CSS-анімації

Анімації надають більше контролю над процесом, дозволяючи створювати складні багатоетапні ефекти за допомогою ключових кадрів.

#### Основи анімації

Анімація створюється за допомогою правила `@keyframes` та властивості `animation`.

Основна відмінність Transitions vs Animations:

Transitions (Переходи):

- Працюють між двома станами (від А до В)
- Потребують тригера (hover, click, focus)
- Автоматично обчислюють проміжні значення
- Простіші у використанні

```

/* Перехід - тільки 2 стани */
.button {
    background: blue;
    transition: background 0.3s;
}

```

```
.button:hover {
    background: red; /* тільки кінцевий стан */
}
```

### Animations (Анімації)

- Можуть мати багато станів ( $A \rightarrow B \rightarrow C \rightarrow D \dots$ )
- Можуть працювати автоматично без тригерів
- Повний контроль над кожним етапом
- Складніші, але значно потужніші

```
/* Анімація - багато станів */
@keyframes rainbow {
    0% { background: red; }
    25% { background: orange; }
    50% { background: yellow; }
    75% { background: green; }
    100% { background: blue; }
}

.element {
    animation: rainbow 2s infinite;
}
```

### Структура CSS-анімації:

#### 1. @keyframes - визначення анімації

```
@keyframes ім'я-анімації {
    /* Ключові кадри */
    0% { /* початковий стан */ }
    50% { /* середній стан */ }
    100% { /* кінцевий стан */ }
}
```

#### 2. animation - застосування анімації

```
.element {
```

```

    animation: ім'я-анімації тривалість функція-часу
затримка повтори напрямок заповнення стан;
}

```

#### Способи задання ключових кадрів:

```

/* 1. Відсотками (найчастіше) */
@keyframes slideIn {
    0% { transform: translateX(-100%); }
    50% { transform: translateX(10px); }
    100% { transform: translateX(0); }
}

/* 2. Ключовими словами */
@keyframes fadeInOut {
    from { opacity: 0; }          /* те ж саме що 0% */
    to { opacity: 1; }           /* те ж саме що 100% */
}

/* 3. Комбінований підхід */
@keyframes complexMove {
    from {
        transform: translateX(0) rotate(0deg);
        background: red;
    }
    25% {
        transform: translateX(100px) rotate(90deg);
        background: orange;
    }
    50% {
        transform: translateX(100px) translateY(100px)
rotate(180deg);

```

```

        background: yellow;
    }
    75% {
        transform: translateX(0) translateY(100px)
rotate(270deg);
        background: green;
    }
    to {
        transform: translateX(0) translateY(0)
rotate(360deg);
        background: blue;
    }
}

```

#### Приклад: Проста анімація пульсації

```

@keyframes pulse {
    0% {
        transform: scale(1);
        opacity: 1;
    }
    50% {
        transform: scale(1.1);      /* збільшуємо на 10%
*/
        opacity: 0.7;              /* робимо
напівпрозорим */
    }
    100% {
        transform: scale(1);      /* повертаємо до
початкового розміру */
        opacity: 1;              /* повертаємо повну
непрозорість */
    }
}

```

```

    }
}

.pulsing-button {
    background: #e74c3c;
    border: none;
    padding: 15px 30px;
    border-radius: 50px;
    color: white;

    /* Застосовуємо анімацію */
    animation: pulse 2s infinite;
    /*
        ↑      ↑      ↑
        назва час повторення */
}

```

#### Ефективні властивості для анімації:

```

@keyframes optimized {
    0% {
        transform: translateX(0) scale(1); /* GPU
прискорення */
        opacity: 0; /* GPU
прискорення */
    }
    100% {
        transform: translateX(100px) scale(1.2);
        opacity: 1;
    }
}

```

#### Повільні властивості:

```

@keyframes slow {

```



```

    0% {
        width: 100px;          /* викликає перерахунок
макета */
        height: 100px;        /* викликає перерахунок
макета */
        left: 0;              /* викликає перерахунок
макета */
    }
    100% {
        width: 200px;
        height: 200px;
        left: 100px;
    }
}

```

### Множина ключових кадрів

Ключові кадри можуть визначати будь-яку кількість проміжних станів:

```

@keyframes complex-movement {
    0% {
        transform: translateX(0) translateY(0)
rotate(0deg);
        background-color: #3498db;
        border-radius: 0;
    }
    25% {
        transform: translateX(100px) translateY(0)
rotate(90deg);
        background-color: #2ecc71;
        border-radius: 25%;
    }
    50% {

```

```

        transform: translateX(100px) translateY(100px)
rotate(180deg);
        background-color: #f39c12;
        border-radius: 50%;
    }
    75% {
        transform: translateX(0) translateY(100px)
rotate(270deg);
        background-color: #9b59b6;
        border-radius: 25%;
    }
    100% {
        transform: translateX(0) translateY(0)
rotate(360deg);
        background-color: #3498db;
        border-radius: 0;
    }
}

.complex-animation {
    width: 60px;
    height: 60px;
    animation: complex-movement 4s ease-in-out
infinite;
}

```

### **Завершення анімації**

Властивість `animation-fill-mode` визначає стан елемента до та після анімації:

```

@keyframes slide-in {
    from {

```

```

        transform: translateX(-100%);
        opacity: 0;
    }
    to {
        transform: translateX(0);
        opacity: 1;
    }
}

.slide-element {
    animation: slide-in 1s ease-out;
    animation-fill-mode: both; /* застосовує стилі з 0%
до початку та зберігає стилі з 100% після завершення */
}

.fill-none { animation-fill-mode: none; }
.fill-forwards { animation-fill-mode: forwards; }
.fill-backwards { animation-fill-mode: backwards; }
.fill-both { animation-fill-mode: both; }

```

### **Затримка анімації**

Подібно до переходів, анімації можуть мати затримку:

```

@keyframes fade-in {
    from { opacity: 0; transform: translateY(20px); }
    to { opacity: 1; transform: translateY(0); }
}

.staggered-animation {
    opacity: 0;
    animation: fade-in 0.6s ease forwards;
}

```

```
.item-1 { animation-delay: 0.1s; }
.item-2 { animation-delay: 0.2s; }
.item-3 { animation-delay: 0.3s; }
.item-4 { animation-delay: 0.4s; }
```

### **Властивість animation**

Повний синтаксис властивості animation:

```
.full-animation {
    /* animation: name duration timing-function delay
iteration-count direction fill-mode play-state; */
    animation: bounce 2s ease-in-out 0.5s infinite
alternate both running;

    /* Або окремо */
    animation-name: bounce;
    animation-duration: 2s;
    animation-timing-function: ease-in-out;
    animation-delay: 0.5s;
    animation-iteration-count: infinite;
    animation-direction: alternate;
    animation-fill-mode: both;
    animation-play-state: running;
}

@keyframes bounce {
    0%, 100% { transform: translateY(0); }
    50% { transform: translateY(-50px); }
}
```

## 2.5 Практичний приклад: Створення банера з анімацією

Розглянемо створення інтерактивного банера з використанням всіх вивчених технік:

```
/* HTML структура:
<div class="banner">
  <div class="banner-content">
    <h1 class="banner-title">Інноваційні
рішення</h1>
    <p class="banner-subtitle">Для вашого
бізнесу</p>
    <button class="banner-btn">Дізнатися
більше</button>
  </div>
  <div class="banner-decoration">
    <div class="floating-element"></div>
    <div class="floating-element"></div>
    <div class="floating-element"></div>
  </div>
</div>
*/

.banner {
  position: relative;
  height: 400px;
  background: linear-gradient(135deg, #667eea 0%,
#764ba2 100%);
  display: flex;
  align-items: center;
  justify-content: center;
```

```
    overflow: hidden;
    color: white;
}

.banner-content {
    text-align: center;
    z-index: 2;
    animation: slideInFromBottom 1s ease-out;
}

.banner-title {
    font-size: 3rem;
    margin: 0 0 1rem 0;
    animation: fadeInScale 0.8s ease-out 0.2s both;
}

.banner-subtitle {
    font-size: 1.2rem;
    margin: 0 0 2rem 0;
    opacity: 0;
    animation: fadeIn 0.6s ease-out 0.6s forwards;
}

.banner-btn {
    padding: 15px 30px;
    font-size: 1.1rem;
    background: rgba(255, 255, 255, 0.2);
    border: 2px solid rgba(255, 255, 255, 0.5);
    color: white;
    border-radius: 30px;
```

```
    cursor: pointer;
    transition: all 0.3s cubic-bezier(0.4, 0, 0.2, 1);
    opacity: 0;
    animation: slideInFromRight 0.8s ease-out 0.8s
forwards;
}

.banner-btn:hover {
    background: rgba(255, 255, 255, 0.3);
    border-color: rgba(255, 255, 255, 0.8);
    transform: translateY(-3px);
    box-shadow: 0 10px 20px rgba(0, 0, 0, 0.2);
}

.banner-btn:active {
    transform: translateY(-1px);
}

.banner-decoration {
    position: absolute;
    top: 0;
    left: 0;
    width: 100%;
    height: 100%;
    pointer-events: none;
}

.floating-element {
    position: absolute;
    width: 60px;
```

```
    height: 60px;
    background: rgba(255, 255, 255, 0.1);
    border-radius: 50%;
    animation: floating 6s ease-in-out infinite;
}

.floating-element:nth-child(1) {
    top: 20%;
    left: 10%;
    animation-delay: 0s;
    animation-duration: 8s;
}

.floating-element:nth-child(2) {
    top: 60%;
    right: 15%;
    width: 40px;
    height: 40px;
    animation-delay: 2s;
    animation-duration: 6s;
}

.floating-element:nth-child(3) {
    bottom: 20%;
    left: 20%;
    width: 80px;
    height: 80px;
    animation-delay: 4s;
    animation-duration: 10s;
}
```



```
/* Ключові кадри для анімацій */  
@keyframes slideInFromBottom {  
    from {  
        transform: translateY(50px);  
        opacity: 0;  
    }  
    to {  
        transform: translateY(0);  
        opacity: 1;  
    }  
}  
  
@keyframes fadeInScale {  
    from {  
        transform: scale(0.8);  
        opacity: 0;  
    }  
    to {  
        transform: scale(1);  
        opacity: 1;  
    }  
}  
  
@keyframes fadeIn {  
    from { opacity: 0; }  
    to { opacity: 1; }  
}  
  
@keyframes slideInFromRight {
```

```

    from {
        transform: translateX(30px);
        opacity: 0;
    }
    to {
        transform: translateX(0);
        opacity: 1;
    }
}

@keyframes floating {
    0%, 100% {
        transform: translateY(0) rotate(0deg);
    }
    25% {
        transform: translateY(-20px) rotate(90deg);
    }
    50% {
        transform: translateY(-40px) rotate(180deg);
    }
    75% {
        transform: translateY(-20px) rotate(270deg);
    }
}

```

Цей практичний приклад демонструє створення повноцінного інтерактивного банера, який об'єднує всі техніки CSS-анімацій та переходів, які ми вивчили протягом лекції. Банер являє собою складну композицію з декількох шарів анімованих елементів, кожен з яких має свою унікальну роль у загальному враженні.

Основою банера служить контейнер із фіксованою висотою 400 пікселів та красивим градієнтним фоном, який плавно переходить від синьо-фіолетового до темно-фіолетового кольору під кутом 135 градусів. Використання flexbox для вирівнювання дозволяє ідеально центрувати весь контент як по горизонталі, так і по вертикалі, створюючи збалансовану композицію. Властивість `overflow: hidden` забезпечує те, що всі анімовані елементи залишаються в межах банера, не виходячи за його границі.

Контентна частина банера включає заголовок, підзаголовок та кнопку, які з'являються послідовно з різними анімаційними ефектами. Спочатку весь контентний блок піднімається знизу за допомогою анімації `slideDownFromBottom`, яка триває одну секунду та використовує функцію `ease-out` для природного сповільнення в кінці руху. Ця базова анімація створює відчуття того, що контент "вирастає" з нижньої частини банера.

Головний заголовок отримує додатковий ефект масштабування через анімацію `fadeInScale`, яка починається через 0.2 секунди після початку загальної анімації контенту. Заголовок спочатку з'являється у зменшеному розмірі (80% від оригінального) та з нульовою прозорістю, а потім поступово збільшується до повного розміру та набуває повної непрозорості. Використання `animation-fill-mode: both` гарантує, що початкові стилі анімації застосовуються ще до її початку, а кінцеві стилі зберігаються після завершення.

Підзаголовок використовує простішу, але не менш ефектну анімацію появи через зміну прозорості. Він початково має нульову прозорість і з'являється через 0.6 секунд після початку загальної анімації, створюючи ефект послідовного розкриття інформації. Властивість `animation-fill-mode: forwards` забезпечує збереження кінцевого стану анімації, тобто повної непрозорості.

Кнопка банера демонструє комбінацію анімації появи та інтерактивних переходів. Вона з'являється останньою, через 0.8 секунд, використовуючи анімацію `slideDownFromRight`, яка імітує рух зліва направо. Після появи кнопка отримує інтерактивну поведінку через CSS-переходи. При наведенні миші кнопка плавно змінює свій фон на більш яскравий, збільшує контрастність

рамки, піднімається на три пікселі вгору та отримує тінь, що створює ефект левітації. При натисканні кнопка злегка опускається, імітуючи фізичну взаємодію.

Декоративні елементи банера створюють атмосферу динамічності та життя. Три кола різного розміру розташовані в різних частинах банера та мають напівпрозорий білий колір, що робить їх ледь помітними, але достатньо видимими для створення глибини композиції. Кожен елемент має унікальні параметри анімації: різну тривалість, затримку початку та розмір.

Анімація *floating* створює складний рух, який поєднує вертикальне переміщення та обертання. На початку та в кінці циклу елементи знаходяться в початковому положенні без зміщення та обертання. На 25% циклу вони піднімаються на 20 пікселів вгору та повертаються на 90 градусів. В середині циклу (50%) досягають максимального підняття в 40 пікселів та обертання в 180 градусів. На 75% циклу повертаються на проміжну висоту з обертанням 270 градусів, після чого плавно повертаються до початкового стану.

Використання різної тривалості анімації для кожного декоративного елемента (8, 6 та 10 секунд) створює ефект хаотичного, але гармонійного руху, який нагадує рух частинок у рідині або повітрі. Затримки в 0, 2 та 4 секунди гарантують, що елементи ніколи не рухаються синхронно, що додає природності анімації.

Весь банер побудований за принципом шарової архітектури, де декоративні елементи знаходяться на задньому плані завдяки *z-index: auto*, а контентна частина піднята на передній план через *z-index: 2*. Властивість *pointer-events: none* для декоративного шару гарантує, що користувач може взаємодіяти з контентом, не зачіпаючи фонові елементи.

Цей приклад ілюструє важливі принципи створення професійних веб-анімацій: послідовність появи елементів створює наратив, різноманітність анімаційних ефектів підтримує увагу користувача, а правильний таймінг забезпечує природне сприйняття. Комбінація одноразових анімацій появи з

безкінечними фоновими анімаціями створює баланс між динамічністю та стабільністю інтерфейсу.

## Висновки

У даному розділі було розглянуто фундаментальні принципи мови гіпертекстової розмітки HTML, яка є базисом сучасної веб-розробки. HTML формує каркас веб-сторінки та забезпечує її структурну організацію, визначаючи як окремі елементи, так і логічні блоки контенту.

Важливим результатом опрацювання матеріалу є розуміння різниці між одинарними та парними тегамі, їхнього призначення та правил вкладеності. Окрема увага приділена семантичним елементам HTML5, які не лише покращують структуру документа, а й забезпечують його доступність та коректну індексацію пошуковими системами.

Було проаналізовано базову структуру HTML-документа, включаючи обов'язкові елементи `<html>`, `<head>` і `<body>`, а також особливості використання метаданих, блокових та рядкових елементів. Значна частина матеріалу присвячена роботі з текстом, посиланнями, зображеннями та списками, які є основними інструментами побудови змістовних і зручних у користуванні веб-сторінок.

Окремо підкреслено роль коментарів у підтримці зрозумілості та документованості коду, а також наведено приклади «правильного» та «некорисного» коментування.

Розглянуто практичні рекомендації щодо написання валідного та семантично правильного коду, що відповідає сучасним стандартам і забезпечує кращу продуктивність, доступність і адаптивність веб-ресурсів.

Таким чином, засвоєння основ HTML створює надійний фундамент для подальшого опанування каскадних таблиць стилів (CSS), мов програмування клієнтської та серверної частини, а також для розробки повноцінних інтерактивних веб-застосунків.

## Питання для самоперевірки

- 1) Що таке HTML і яку роль він відіграє у створенні веб-сторінок?
- 2) Чим HTML відрізняється від мов програмування?
- 3) Яке призначення декларації `<!DOCTYPE html>`?
- 4) Які основні обов'язкові елементи має стандартний HTML-документ?
- 5) У чому різниця між парними та одинарними тегами? Наведіть приклади.
- 6) Для чого використовуються атрибути тегів? Які атрибути є обов'язковими для елемента `<img>`?
- 7) Яку роль виконує елемент `<head>` у структурі документа?
- 8) Які дані розташовуються у контейнері `<body>`?
- 9) Чим відрізняються блокові елементи від рядкових? Наведіть приклади.
- 10) Які семантичні елементи HTML5 ви знаєте? Для чого вони призначені?
- 11) Як правильно організовувати структуру заголовків (`<h1>`–`<h6>`)?
- 12) Які типи списків існують в HTML і для чого вони застосовуються?
- 13) Чим відрізняються абсолютні та відносні посилання?
- 14) Що таке якірні посилання і як вони створюються?
- 15) Яке значення має атрибут `alt` у зображеннях і чому він важливий для доступності?
- 16) У чому полягає різниця між семантичним та несемантичним використанням елементів `<div>` і `<span>`?
- 17) Як створити форму в HTML і які основні елементи вона може містити?
- 18) Для чого використовуються HTML-коментарі? Які приклади «поганих» коментарів ви можете навести?
- 19) Які сучасні практики роботи із зображеннями дозволяють покращити продуктивність веб-сайту?
- 20) Які основні принципи валідності та семантичності HTML-коду?

## **РОЗДІЛ 3. ВЕБПРОГРАМУВАННЯ НА JAVASCRIPT: МОВА, ДОКУМЕНТ, ПОДІЇ**

### **3.1 Мова програмування JavaScript**

JavaScript є однією з найпоширеніших і найважливіших мов програмування сучасного веб-розробництва. Ця мова, створена в 1995 році Бренданом Айком в компанії Netscape, спочатку призначалася для надання інтерактивності веб-сторінкам, проте сьогодні вона використовується не лише в браузерях, а й на серверах, в мобільних додатках, настільних програмах та навіть в розробці ігор.

#### **3.1.1 Вступ до JavaScript**

JavaScript є динамічною, слабо типізованою мовою програмування, яка підтримує об'єктно-орієнтоване, функціональне та процедурне програмування. Головною особливістю JavaScript є його здатність виконуватися безпосередньо в веб-браузері, що дозволяє створювати інтерактивні веб-сторінки без необхідності перезавантаження сторінки.

Основні переваги JavaScript включають:

Простота у вивченні та використанні – JavaScript має відносно просту синтаксичну структуру, схожу на С-подібні мови, що робить його доступним для початківців у програмуванні.

Інтерпретована природа – код JavaScript не потребує попередньої компіляції і виконується безпосередньо браузером, що значно прискорює процес розробки та тестування.

Кросплатформність – JavaScript працює в усіх сучасних браузерах незалежно від операційної системи, забезпечуючи універсальність розроблених додатків.

Велика екосистема – навколо JavaScript сформувалася потужна екосистема бібліотек, фреймворків та інструментів, які значно спрощують розробку складних додатків.

JavaScript може використовуватися для вирішення широкого спектру завдань: від простої валідації форм до створення складних односторінкових додатків (SPA), веб-ігор, чат-ботів та систем управління контентом.

### 3.1.2 Інструменти розробника

Для ефективної роботи з JavaScript сучасні браузерери надають потужний набір інструментів розробника, які дозволяють налагоджувати, профілювати та оптимізувати код. Ці інструменти є невід'ємною частиною процесу розробки і значно спрощують пошук та виправлення помилок.

Консоль розробника є основним інструментом для інтерактивної роботи з JavaScript. Вона дозволяє виконувати команди JavaScript в реальному часі, переглядати результати виконання коду та отримувати інформацію про помилки. Для відкриття консолі в більшості браузерів використовується комбінація клавіш F12 або Ctrl+Shift+I (Cmd+Option+I на Mac).

У консолі можна виконувати будь-які JavaScript-команди:

```
console.log("Привіт, світ!");  
console.error("Це повідомлення про помилку");  
console.warn("Це попередження");  
console.info("Інформаційне повідомлення");
```

Вкладка Elements/Елементи дозволяє переглядати та редагувати HTML-структуру сторінки в реальному часі, що особливо корисно при роботі з DOM-маніпуляціями.

Вкладка Sources/Джерела надає можливість переглядати вихідний код JavaScript-файлів, встановлювати точки зупинки (breakpoints) для налагодження та покроково виконувати код.



Вкладка Network/Мережа показує всі мережеві запити, що виконуються сторінкою, включаючи завантаження JavaScript-файлів, що допомагає оптимізувати швидкість завантаження.

Для початківців особливо важливо навчитися користуватися консоллю, оскільки вона дозволяє експериментувати з кодом та миттєво бачити результати.

### 3.1.3 Тег "script"

HTML-тег `<script>` є основним способом включення JavaScript-коду в веб-сторінку. Цей тег повідомляє браузеру, що всередині нього знаходиться виконуваний код JavaScript, який повинен бути оброблений відповідним чином.

Базове використання тегу `script` виглядає наступним чином:

```
<!DOCTYPE html>
<html>
<head>
  <title>Приклад JavaScript</title>
</head>
<body>
  <h1>Моя веб-сторінка</h1>

  <script>
    alert("Ця сторінка використовує JavaScript!");
    console.log("Сторінка завантажена успішно");
  </script>
</body>
</html>
```

Щоб зрозуміти роль тега `<script>`, варто уявити, що вебсторінка завантажується поступово, зчитуючи код згори донизу. Коли браузер зустрічає цей тег, він зупиняє обробку HTML і починає виконувати JavaScript, що

міститься всередині або підключений зовнішнім файлом. Це означає, що місце розташування скрипта впливає на швидкість і послідовність завантаження сторінки.

Існує два основні способи використання `<script>`. Перший — це коли код пишуть безпосередньо між відкривальним і закривальним тегом. Це зручно для невеликих фрагментів коду або коли потрібно швидко перевірити якусь функцію. Другий — це підключення зовнішнього файлу за допомогою атрибута `src`. Такий підхід набагато зручніший у більшості випадків, бо дозволяє відокремити JavaScript від розмітки, робить сторінку більш акуратною та полегшує супровід коду.

Часто можна зустріти поради розміщувати `<script>` наприкінці документа, безпосередньо перед закривальним тегом `</body>`. Це робиться для того, щоб спершу завантажився і відобразився HTML-вміст, а вже після цього підвантажувався і виконувався JavaScript. Такий підхід дозволяє користувачеві швидше побачити сторінку, навіть якщо скрипти ще обробляються.

Є й сучасніші варіанти: використання атрибутів `defer` або `async`. Вони допомагають завантажувати скрипти паралельно з розміткою, не блокуючи побудову сторінки. `defer` забезпечує виконання коду лише після повного завантаження документа, а `async` дозволяє виконати його одразу після завантаження, не чекаючи решти коду.

Атрибут `defer` вказує браузеру завантажувати скрипт паралельно з HTML, але виконувати його лише після повного завантаження DOM:

```
<script defer src="script.js"></script>
```

Атрибут `async` дозволяє завантажувати та виконувати скрипт асинхронно, не блокуючи рендеринг сторінки:

```
<script async src="analytics.js"></script>
```

### 3.1.4 Зовнішні скрипти

Хоча вбудований JavaScript зручний для невеликих фрагментів коду, для більших проектів рекомендується виносити код в окремі файли. Це покращує організацію коду, полегшує його повторне використання та спрощує процес розробки та підтримки.

Щоб зрозуміти сенс підключення зовнішніх скриптів, уявіть собі, що ви створюєте сайт не з однією сторінкою, а з цілою структурою — головна, сторінка з контактами, новини, каталог. На кожній із них повинна працювати одна й та сама логіка: наприклад, перевірка заповнення форм, меню, яке розкривається при кліку, або невелика анімація. Якби ви кожного разу копіювали один і той самий код у різні HTML-файли, це не лише займало б зайвий час, але й створювало б додатковий ризик помилок. Варто змінити щось у цьому коді — і вам довелося б оновлювати кожен файл окремо.

Створення зовнішнього файлу передбачає збереження JavaScript-коду в файлі з розширенням .js. Наприклад, створимо файл app.js:

```
// app.js

function привітатиКористувача() {
    const ім_я = prompt("Як вас звати?");
    if (ім_я) {
        alert(`Привіт, ${ім_я}! Ласкаво просимо на наш сайт.`);
    }
}

function змінитиКолірТексту() {
    document.body.style.color = 'darkgreen';
}

// Виконати функції після завантаження сторінки
```

```
привітатиКористувача();
змінитиКолірТексту();
```

Ви створюєте один файл, наприклад `app.js`, де міститься вся необхідна програма, і потім підключаєте його там, де потрібно. Це схоже на те, як у будинку прокладають центральну систему опалення: достатньо підключити труби в усіх кімнатах до одного котла, замість того, щоб ставити окремий обігрівач у кожній.

Підключення зовнішнього скрипту виконується за допомогою тега `<script>` з атрибутом `src`. У HTML-коді це виглядає так:

```
<script src="app.js"></script>
```

Коли браузер бачить цей запис, він не шукає код усередині тегів, а завантажує зазначений файл і виконує його. Якщо той самий файл підключено на кількох сторінках, браузер завантажить його лише одного разу, а потім збереже у своїй пам'яті (кеші) і використовуватиме повторно. Це пришвидшує роботу сайту, адже зменшується кількість запитів до сервера.

Крім того, використання зовнішніх скриптів покращує організацію проекту. Ваш HTML-код залишається чистішим і легшим для читання, а JavaScript-файл можна редагувати окремо, без ризику зачепити верстку. Якщо потрібно щось змінити, ви відкриваєте лише `app.js`, вносите правки і зберігаєте — зміни відразу стають видимими на всіх сторінках, де цей файл підключений.

У великих проєктах це не просто зручно, а й необхідно. Уявіть, що ваш сайт зростає і замість одного скрипту з'являються десятки: окремо для обробки форм, для роботи з анімаціями, для підключення сторонніх бібліотек. Кожен з них можна винести в окремий файл, а потім підключити лише там, де він справді потрібен. Це робить сайт швидшим і допомагає підтримувати його в порядку.

Шляхи до файлів можуть бути як відносними, так і абсолютними:

```
<!-- Відносний шлях -->
<script src="js/main.js"></script>
<script src="../../scripts/utils.js"></script>
```

```
<!-- Абсолютний шлях -->
<script src="/assets/js/application.js"></script>

<!-- Зовнішні ресурси -->
<script
src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.6.
0/jquery.min.js"></script>
```

### 3.1.5 Структура коду

Структура коду є фундаментальною основою, яка визначає якість, читабельність та підтримуваність будь-якого JavaScript-проекту. Коли розробники дотримуються загальноприйнятих стандартів та конвенцій у структуруванні свого коду, це створює єдине середовище, в якому кожен учасник команди може легко орієнтуватися та розуміти логіку програми, навіть якщо її писав хтось інший.

Використання крапки з комою після кожної інструкції є однією з найважливіших практик у JavaScript-програмуванні, оскільки хоча мова й має механізм автоматичної вставки крапок з комою (Automatic Semicolon Insertion), цей механізм може працювати не завжди передбачувано та іноді призводити до неочікуваних помилок. Явне вказування крапки з комою робить код більш надійним та зменшує ймовірність виникнення синтаксичних проблем, особливо при мініфікації коду або при роботі з різними інструментами збірки.

Відступи та загальне форматування коду відіграють критично важливу роль у створенні візуально структурованого та логічно організованого коду, який легко сприймається людським оком. Послідовне використання відступів, чи то два чи чотири пробіли (головне, щоб це було однаково по всьому проекту), допомагає чітко відображати ієрархію та вкладеність блоків коду, що робить структуру програми інтуїтивно зрозумілою навіть при швидкому перегляді.

```
function обчислитиПлощуТрикутника(основа, висота) {
  if (основа > 0 && висота > 0) {
    let площа = (основа * висота) / 2;
    return площа;
  } else {
    console.error("Основа та висота повинні бути
додатними числами");
    return null;
  }
}

let результат = обчислитиПлощуТрикутника(10, 8);
if (результат !== null) {
  console.log(`Площа трикутника: ${результат}
квадратних одиниць`);
}
```

Коментарі служать мостом між кодом та його розумінням, надаючи контекст та пояснення для складних алгоритмів, бізнес-логіки або нетривіальних рішень. JavaScript пропонує два типи коментарів: однорядкові, які починаються з подвійного слеша та продовжуються до кінця рядка, та багаторядкові, які укладені між символами `/*` та `*/` і можуть охоплювати декілька рядків тексту, що особливо корисно для детальної документації функцій або складних фрагментів коду.

```
// Це однорядковий коментар
let швидкість = 60; // км/год

/*
Це багаторядковий коментар,
який може охоплювати
декілька рядків тексту
```

```

*/
function розрахуватиЧас(відстань, швидкість) {
    /**
     * Функція розраховує час подорожі
     * @param {number} відстань - відстань у кілометрах
     * @param {number} швидкість - швидкість у км/год
     * @returns {number} час у годинах
     */
    return відстань / швидкість;
}

```

Іменування змінних та функцій є мистецтвом, яке вимагає балансу між стислістю та описовістю, адже добре обрані назви можуть замінити цілі абзаци коментарів, роблячи код самодокументованим. У JavaScript традиційно використовується стиль camelCase, де перше слово пишеться з маленької літери, а кожне наступне слово починається з великої літери, що створює візуально приємний та зрозумілий код, який легко читається та запам'ятовується.

```

// Добре іменування
let кількістьСтудентів = 30;
let середнійБал = 4.2;

function розрахуватиФінальнуОцінку(поточнийБал,
екзаменаційнийБал) {
    return (поточнийБал * 0.6) + (екзаменаційнийБал *
0.4);
}

// Уникайте неописових назв
let x = 30; // незрозуміло, що означає x

```

```
let func1 = function(a, b) { return a + b; }; // назва  
не описує призначення
```

Правильна структуризація коду також передбачає логічне групування пов'язаних між собою фрагментів, розділення різних функціональних блоків порожніми рядками, та організацію коду таким чином, щоб логіка програми розкривалася поступово, від загального до конкретного. Це створює своєрідну історію, яку розповідає код, де кожна функція, змінна та коментар відіграють свою роль у загальній картині програми, роблячи процес розробки більш ефективним та приємним для всіх учасників проекту.

Правильна структуризація коду є мистецтвом створення логічно організованої програми, яка читається як добре написана книга з чітким сюжетом та послідовним розвитком ідей. Цей підхід передбачає створення візуальної ієрархії та функціональної архітектури, де кожен елемент має своє призначення та місце в загальній структурі програми.

Логічне групування пов'язаних фрагментів означає об'єднання коду, який виконує схожі або взаємопов'язані завдання, в окремі блоки або секції. Наприклад, всі функції валідації даних слід розмістити поруч одна з одною, функції математичних обчислень - в окремій групі, а функції виведення результатів - в своїй секції. Це дозволяє розробнику швидко знаходити потрібний код та розуміти його призначення без необхідності шукати по всьому файлу.

Розділення функціональних блоків порожніми рядками створює візуальні "паузи" в коді, які допомагають очам відпочити та підготуватися до сприйняття нової інформації. Ці пробіли працюють як розділові знаки в тексті, показуючи де закінчується одна ідея і починається інша. Особливо важливо відділяти різні функції, блоки коментарів, константи та змінні один від одного.

Організація від загального до конкретного передбачає структурування коду таким чином, щоб спочатку йшли найбільш загальні елементи (константи, конфігурація, базові структури даних), потім основні функції високого рівня, а далі - допоміжні функції та деталі реалізації. Це нагадує принцип написання



наукових статей або технічної документації, де спочатку подається загальний огляд проблеми, а потім поступово заглиблюються в деталі.

У наведеному прикладі системи управління студентами можна побачити, як ці принципи працюють на практиці. Код починається з констант та конфігурації, які встановлюють основні правила системи. Далі йдуть структури даних, які визначають, як буде зберігатися інформація. Потім розташовані основні функції управління студентами - найважливіша функціональність програми. Наступна секція містить функції обчислення та аналізу - більш спеціалізовані операції. Завершується код секцією статистики та демонстрацією роботи.

Кожна функція супроводжується детальними коментарями, які пояснюють її призначення, параметри та результат роботи. Коментарі-заголовки з символами = створюють візуальні розділи, які допомагають швидко орієнтуватися в структурі коду. Назви змінних та функцій обрані таким чином, щоб максимально точно описувати їхнє призначення, що робить код самодокументованим.

Така організація коду створює своєрідну історію про систему управління студентами, де кожен розділ розповідає свою частину: спочатку встановлюються правила гри (константи), потім визначаються учасники (структури даних), далі описуються основні дії (функції управління), потім аналіз результатів (функції обчислень) і нарешті - звітність та демонстрація можливостей системи. Це робить код не лише функціональним, але й зрозумілим для будь-кого, хто працюватиме з ним у майбутньому.

Приклад:

```
// =====
// СИСТЕМА УПРАВЛІННЯ СТУДЕНТАМИ
// =====

// ----- КОНСТАНТИ ТА КОНФІГУРАЦІЯ -----
const МІНІМАЛЬНИЙ_БАЛ = 2;
```

```

const МАКСИМАЛЬНИЙ_БАЛ = 5;
const ПРОХІДНИЙ_БАЛ = 3;

// ----- СТРУКТУРИ ДАНИХ -----
// Масив студентів для зберігання інформації
let студенти = [];

// Об'єкт статистики для зберігання загальних
показників
let статистика = {
    загальнаКількість: 0,
    середнійБал: 0,
    кількістьВідмінників: 0
};

// =====
// ОСНОВНІ ФУНКЦІЇ УПРАВЛІННЯ СТУДЕНТАМИ
// =====

/**
 * Створює новий об'єкт студента з валідацією даних
 * @param {string} ім'я - Ім'я студента
 * @param {number} вік - Вік студента
 * @param {Array} оцінки - Масив оцінок студента
 * @returns {Object|null} Об'єкт студента або null при
помилці
 */
function створитиСтудента(ім'я, вік, оцінки = []) {
    // Валідація вхідних даних

```

```

        if (!ім'я || typeof ім'я !== 'string' || ім'я.trim()
=== '') {
            console.error('Помилка: некоректне ім\'я
студента');
            return null;
        }

        if (!вік || вік < 16 || вік > 100) {
            console.error('Помилка: некоректний вік
студента');
            return null;
        }

        // Створення та повернення об'єкта студента
        return {
            ім'я: ім'я.trim(),
            вік: вік,
            оцінки: оцінки,
            середнійБал: розрахуватиСереднійБал(оцінки),
            статус:
визначитиСтатус(розрахуватиСереднійБал(оцінки))
        };
    }

/**
 * Додає студента до загального списку
 * @param {Object} студент - Об'єкт студента
 * @returns {boolean} true при успішному додаванні
 */
function додатиСтудента(студент) {

```

```

        if (!студент) {
            console.error('Помилка: неможливо додати
неіснуючого студента');
            return false;
        }

        студенти.push(студент);
        оновитиСтатистику();

        console.log(`Студента ${студент.імя} успішно додано
до системи`);
        return true;
    }

// =====
// ФУНКЦІЇ ОБЧИСЛЕННЯ ТА АНАЛІЗУ
// =====

/**
 * Розраховує середній бал студента
 * @param {Array} оцінки - Масив оцінок
 * @returns {number} Середній бал (округлений до двох
знаків)
 */
function розрахуватиСереднійБал(оцінки) {
    if (!оцінки || оцінки.length === 0) {
        return 0;
    }

```

```

// Валідація оцінок
const валідніОцінки = оцінки.filter(оцінка =>
  typeof оцінка === 'number' &&
  оцінка >= МІНІМАЛЬНИЙ_БАЛ &&
  оцінка <= МАКСИМАЛЬНИЙ_БАЛ
);

if (валідніОцінки.length === 0) {
  return 0;
}

const сума = валідніОцінки.reduce((акумулятор,
оцінка) => акумулятор + оцінка, 0);
return Math.round((сума / валідніОцінки.length) *
100) / 100;
}

/**
 * Визначає академічний статус студента на основі
середнього балу
 * @param {number} середнійБал - Середній бал студента
 * @returns {string} Статус студента
 */
function визначитиСтатус(середнійБал) {
  if (середнійБал >= 4.5) {
    return 'Відмінник';
  } else if (середнійБал >= 3.5) {
    return 'Хорошист';
  } else if (середнійБал >= ПРОХІДНИЙ_БАЛ) {
    return 'Задовільно';
  }
}

```

```

    } else {
        return 'Потребує покращення';
    }
}

// =====
// ФУНКЦІЇ СТАТИСТИКИ ТА ЗВІТНОСТІ
// =====

/**
 * Оновлює загальну статистику всіх студентів
 */
function оновитиСтатистику() {
    статистика.загальнаКількість = студенти.length;

    if (студенти.length === 0) {
        статистика.середнійБал = 0;
        статистика.кількістьВідмінників = 0;
        return;
    }

    // Розрахунок середнього балу по всіх студентам
    const сумаБалів = студенти.reduce((сума, студент)
=> сума + студент.середнійБал, 0);
    статистика.середнійБал = Math.round((сумаБалів /
студенти.length) * 100) / 100;

    // Підрахунок кількості відмінників
    статистика.кількістьВідмінників = студенти.filter(

```

```

        студент => студент.статус === 'Відмінник'
    ).length;
}

/**
 * Виводить детальну статистику групи
 */
function вивестиСтатистику() {
    console.log('\n=== СТАТИСТИКА ГРУПИ ===');
    console.log(`Загальна кількість студентів:
${статистика.загальнаКількість}`);
    console.log(`Середній бал групи:
${статистика.середнійБал}`);
    console.log(`Кількість відмінників:
${статистика.кількістьВідмінників}`);

    if (статистика.загальнаКількість > 0) {
        const відсотокВідмінників = Math.round(
            (статистика.кількістьВідмінників /
статистика.загальнаКількість) * 100
        );
        console.log(`Відсоток відмінників:
${відсотокВідмінників}%`);
    }
}

// =====
//  ДЕМОНСТРАЦІЯ РОБОТИ СИСТЕМИ
//  =====

```

```
// Створення тестових студентів
const студент1 = створитиСтудента('Марія Петренко', 20,
[5, 4, 5, 5, 4]);
const студент2 = створитиСтудента('Олександр Іваненко',
19, [3, 3, 4, 3, 4]);
const студент3 = створитиСтудента('Анна Козлова', 21,
[5, 5, 5, 4, 5]);

// Додавання студентів до системи
if (студент1) додатиСтудента(студент1);
if (студент2) додатиСтудента(студент2);
if (студент3) додатиСтудента(студент3);

// Виведення результатів
console.log('\n=== СПИСОК СТУДЕНТІВ ===');
студенти.forEach((студент, індекс) => {
    console.log(`${індекс + 1}. ${студент.імя}
(${студент.вік} років)`);
    console.log(`    Середній бал:
${студент.середнійБал}`);
    console.log(`    Статус: ${студент.статус}`);
});

// Виведення загальної статистики
вивестиСтатистику();
```



### 3.1.6 Змінні

Змінні в JavaScript є контейнерами для зберігання даних, які можуть змінюватися під час виконання програми. JavaScript надає три основні способи оголошення змінних: `var`, `let` та `const`, кожен з яких має свої особливості та області застосування.

Оголошення змінних за допомогою `let`. Ключове слово `let` є сучасним способом оголошення змінних, введеним в стандарті ES6. Змінні, оголошені за допомогою `let`, мають блочну область видимості:

```
let ім_яКористувача = "Марія";
let вікКористувача = 22;
let активнийСтудент = true;

console.log("Користувач:", ім_яКористувача);
console.log("Вік:", вікКористувача);

// Змінні можна перевизначати
вікКористувача = 23;
console.log("Новий вік:", вікКористувача);

// Блочна область видимості
if (активнийСтудент) {
    let спеціальність = "Комп'ютерні науки";
    console.log("Спеціальність:", спеціальність);
}

// console.log(спеціальність); // Помилка: змінна
// недоступна поза блоком
```

Константи з `const`. Ключове слово `const` використовується для оголошення констант - змінних, значення яких не може змінюватися після ініціалізації:

```

const УНІВЕРСИТЕТ = "Київський політехнічний інститут";
const ПОТОЧНИЙ_РІК = 2024;
const PI = 3.14159;

console.log(`Навчаюся в ${УНІВЕРСИТЕТ} у
${ПОТОЧНИЙ_РІК} році`);

// ПОТОЧНИЙ_РІК = 2025; // Помилка: неможливо
перевизначити константу

// Для об'єктів та масивів const захищає від
перевизначення, але не від зміни вмісту
const студент = {
    ім_я: "Олександр",
    курс: 3
};

студент.курс = 4; // Це дозволено
console.log(студент); // { ім_я: "Олександр", курс: 4 }

// студент = {}; // Помилка: неможливо перевизначити
об'єкт

```

**Застарілий var.** Хоча var все ще підтримується, його використання не рекомендується в сучасному JavaScript через проблеми з областю видимості:

```

// var має функціональну область видимості
function приклад() {
    if (true) {
        var старийСтиль = "Доступний поза блоком";
        let новийСтиль = "Доступний тільки в блоці";
    }
}

```

```

    console.log(старийСтиль); // Працює
    // console.log(новийСтиль); // Помилка
}

```

Правила іменування змінних передбачають використання описових назв, які починаються з літери, знака підкреслення або долара:

```

// Правильні назви
let ім_я = "Ігор";
let _приватнаЗмінна = "секретно";
let $елементДОМ = document.getElementById("myElement");
let кількістьВідвідувань = 0;

// Неправильні назви
// let 2змінна = "помилка"; // Не може починатися з
цифри
// let моя-змінна = "помилка"; // Дефіс недозволений
// let function = "помилка"; // Зарезервоване слово

```

### 3.1.7 Типи даних

JavaScript є динамічно типізованою мовою, що означає, що тип змінної визначається автоматично під час виконання програми на основі присвоєного їй значення. Розуміння типів даних є критично важливим для ефективного програмування на JavaScript.

Примітивні типи даних представляють найпростіші значення в JavaScript:

Number (Число) представляє як цілі числа, так і числа з плаваючою комою:

```

let ціле_число = 42;
let дробове_число = 3.14159;
let від_ємне_число = -273.15;

```

```
let велике_число = 1.7976931348623157e+308;

console.log(typeof ціле_число); // "number"
console.log(typeof дробове_число); // "number"

// Спеціальні числові значення
let нескінченність = Infinity;
let не_число = NaN; // Not a Number
console.log(10 / 0); // Infinity
console.log("текст" / 2); // NaN
```

**String (Рядок)** використовується для зберігання тексту:

```
let одинарні_лапки = 'Привіт, світ!';
let подвійні_лапки = "JavaScript - це круто!";
let зворотні_лапки = `Шаблонний рядок з
${одинарні_лапки}`;

console.log(зворотні_лапки); // "Шаблонний рядок з
Привіт, світ!"

// Багаторядкові рядки
let вірш = `Червона рута,
червона рута,
чому ж ти в'янеш,
чому не цвітеш?`;

// Екрановані символи
let спеціальні_символи = "Рядок з \"лапками\" та \n
новим рядком";
```

**Boolean (Логічний тип)** має лише два значення:

```
let студент_активний = true;
```

```
let іспит_складений = false;

// Логічні операції
let може_здавати_іспит = студент_активний &&
!іспит_складений;
console.log("Може здавати іспит:", може_здавати_іспит);
// true
```

Undefined та Null представляють відсутність значення:

```
let не_ініціалізована; // undefined
let порожня = null; // null

console.log(не_ініціалізована); // undefined
console.log(typeof не_ініціалізована); // "undefined"
console.log(typeof порожня); // "object" (особливість
JavaScript)
```

Складні типи даних включають об'єкти та масиви:

```
// Масив
let предмети = ["Математика", "Фізика",
"Програмування", "Англійська"];
console.log(предмети[0]); // "Математика"
console.log(предмети.length); // 4

// Об'єкт
let студент = {
  ім_я: "Анна",
  вік: 20,
  курс: 2,
  оцінки: [5, 4, 5, 3, 4]
};
```

```
console.log(студент.ім_я); // "Анна"  
console.log(студент["курс"]); // 2
```

Складні типи даних у JavaScript представляють фундаментально інший підхід до організації та зберігання інформації порівняно з простими типами даних. Якщо прості типи (числа, рядки, булеві значення) зберігають лише одне значення, то складні типи дозволяють створювати структуровані колекції даних, які можуть містити безліч взаємопов'язаних елементів різних типів. Об'єкти є основною структурою даних у JavaScript, яка дозволяє групувати пов'язані між собою дані та функціональність в єдину логічну одиницю. Об'єкт складається з пар "ключ-значення", де ключ (також називається властивістю або полем) є унікальним ідентифікатором, а значення може бути будь-яким типом даних - від простого числа до іншого об'єкта або функції. Це робить об'єкти надзвичайно гнучким інструментом для моделювання реальних сутностей та концепцій у програмному коді. Об'єкти можна створювати різними способами: за допомогою літералу об'єкта (фігурні дужки), конструктора `Object()`, або через функції-конструктори. Найпоширенішим є використання літералу об'єкта, оскільки це найбільш читабельний та зрозумілий синтаксис. Доступ до властивостей об'єкта здійснюється через крапкову нотацію або квадратні дужки, причому квадратні дужки дозволяють використовувати динамічні імена властивостей. Масиви представляють упорядковану колекцію елементів, де кожен елемент має свій унікальний числовий індекс, починаючи з нуля. Масиви в JavaScript є динамічними структурами даних, що означає можливість додавання, видалення та модифікації елементів після створення масиву. Важливою особливістю масивів у JavaScript є те, що вони можуть містити елементи різних типів в одному масиві, що надає додаткову гнучкість у роботі з даними. Масиви мають безліч вбудованих методів для маніпулювання даними: `push()` та `pop()` для додавання та видалення елементів з кінця масиву, `unshift()` та `shift()` для роботи з початком масиву, `slice()` для створення копій частини масиву, `splice()` для вставлення та видалення елементів у будь-якому місці, а також

функціональні методи як `map()`, `filter()`, `reduce()`, які дозволяють обробляти дані в функціональному стилі. Складні типи даних передаються за посиланням, що означає, що коли ви присвоюєте об'єкт або масив змінній, ця змінна зберігає не саме значення, а посилання на місце в пам'яті, де зберігається цей об'єкт або масив. Це має важливі наслідки при копіюванні та порівнянні складних типів даних, оскільки операції виконуються з посиланнями, а не з фактичними значеннями. Наприклад, коли ви створюєте об'єкт, що описує студента з властивостями імені, віку, та масиву оцінок, ви фактично створюєте складну структуру даних, яка може зберігати різнотипну інформацію в логічно організованому вигляді. Цей об'єкт може також містити методи (функції як властивості), які дозволяють виконувати операції над даними цього об'єкта, створюючи таким чином повноцінну модель реальної сутності з її характеристиками та поведінкою. Розуміння роботи зі складними типами даних є критично важливим для ефективного програмування на JavaScript, оскільки вони є основою для створення більшості реальних додатків, від простих калькуляторів до складних веб-застосунків з багатьма взаємопов'язаними компонентами та структурами даних.

Перевірка типів здійснюється за допомогою оператора `typeof`:

```
console.log(typeof 42); // "number"
console.log(typeof "привіт"); // "string"
console.log(typeof true); // "boolean"
console.log(typeof undefined); // "undefined"
console.log(typeof null); // "object" (особливість мови)
console.log(typeof []); // "object"
console.log(typeof {}); // "object"
console.log(typeof function(){}); // "function"
```

### 3.1.8 Взаємодія: alert, prompt, confirm

JavaScript надає три базові функції для взаємодії з користувачем через браузер: `alert`, `prompt` та `confirm`. Ці функції створюють модальні діалогові вікна, які дозволяють відображати повідомлення та отримувати прості відповіді від користувача.

Функція `alert` є найпростішим способом показати повідомлення користувачу. Коли браузер виконує цю команду, він зупиняє виконання всього коду та показує маленьке вікно з вашим текстом і кнопкою "ОК". Користувач не може нічого робити на сторінці, доки не натисне цю кнопку.

```
alert("Привіт!");
```

Ця функція корисна для швидких сповіщень, але її варто використовувати обережно, оскільки вона може роздратувати користувачів, якщо з'являється занадто часто. У реальних проектах `alert` часто замінюють на більш елегантні рішення.

Функція `prompt` дозволяє запитати у користувача якусь інформацію. Вона показує діалогове вікно з текстом запитання, полем для введення тексту та двома кнопками: "ОК" та "Скасувати". Якщо користувач натискає "ОК", функція повертає введений текст, а якщо "Скасувати" - повертає `null`.

```
let ім_я = prompt("Як вас звати?");  
console.log("Користувач ввів:", ім_я);
```

Важливо розуміти, що `prompt` завжди повертає рядок, навіть якщо користувач введе число. Тому для числових значень потрібно робити перетворення:

```
let вік = prompt("Скільки вам років?");  
let вік_число = Number(вік);  
  
if (isNaN(вік_число)) {  
    alert("Це не число!");  
}
```



Функція `prompt` може мати друге значення - текст за замовчуванням, який уже буде в полі введення:

```
let колір = prompt("Ваш улюблений колір?", "синій");
```

Функція `confirm` запитує у користувача "так" чи "ні". Вона показує повідомлення з двома кнопками: "ОК" (що означає "так") та "Скасувати" (що означає "ні"). Функція повертає `true`, якщо натиснуто "ОК", та `false`, якщо "Скасувати".

```
let згоден = confirm("Чи погоджуєтесь ви з умовами?");

if (згоден) {
    alert("Дякуємо!");
} else {
    alert("Шкода...");
}
```

Практичне використання разом цих функцій дозволяє створювати прості інтерактивні додатки:

```
function простаГра() {
    let грати = confirm("Хочете зіграти в гру?");

    if (грати) {
        let число = Math.floor(Math.random() * 10) + 1;
        let здогадка = prompt("Вгадайте число від 1 до 10:");

        if (Number(здогадка) === число) {
            alert("Вітаємо! Ви вгадали!");
        } else {
            alert(`На жаль, було ${число}`);
        }
    }
}
```

```
}
```

Обмеження цих функцій полягають у тому, що вони виглядають застарілими, не можуть бути стилізованими та блокують роботу всієї сторінки. У сучасному веб-розробленні їх замінюють на спеціальні модальні вікна, створені за допомогою HTML, CSS та JavaScript, які мають привабливіший вигляд та більше функцій.

Простий приклад сучасної альтернативи:

```
function показатиПовідомлення(текст) {
    const вікно = document.createElement('div');
    вікно.innerHTML = `
        <div style="background: rgba(0,0,0,0.5);
position: fixed; top: 0; left: 0; width: 100%; height:
100%;">

            <div style="background: white; margin: 20%
auto; padding: 20px; width: 300px; text-align:
center;">

                ${текст}
                <br><br>
                <button
onclick="this.parentElement.parentElement.remove()">OK<
/button>

            </div>
        </div>
    `;
    document.body.appendChild(вікно);
}
```

Ці базові функції взаємодії є відмінним початком для вивчення JavaScript, оскільки вони дозволяють швидко побачити результати своєї роботи та створити перші інтерактивні елементи.

### 3.1.9 Математика

Математичні операції у JavaScript є невід'ємною частиною програмування, оскільки вони дозволяють виконувати обчислення, створювати алгоритми та розв'язувати практичні завдання. JavaScript надає як базові арифметичні операції, так і потужний об'єкт `Math` з безліччю спеціалізованих функцій.

Основні арифметичні операції працюють інтуїтивно та покривають всі необхідні математичні дії. Додавання виконується за допомогою знака плюс, який також використовується для конкатенації рядків, тому важливо розуміти контекст використання. Коли ми працюємо з числами, JavaScript автоматично виконує математичне додавання, але якщо один з операндів є рядком, то відбувається з'єднання рядків.

```
let a = 10;
let b = 5;
console.log(a + b); // 15 (математичне додавання)
console.log("10" + "5"); // "105" (конкатенація рядків)
console.log(a + "5"); // "105" (число перетворюється в рядок)
```

Операція віднімання завжди намагається перетворити операнди в числа, що може призвести до неочікуваних результатів при роботі з рядками або іншими типами даних.

```
console.log(a - b); // 5
console.log("15" - "3"); // 12 (рядки автоматично перетворюються в числа)
console.log("hello" - 5); // NaN (Not a Number)
```

Операція модуля або остачі від ділення є особливо корисною для циклічних обчислень, перевірки парності чисел або створення повторюваних патернів. Наприклад, при створенні календаря можна використовувати модуль для визначення дня тижня.

```

let день = 15;
let день_тижня = день % 7; // Визначення дня тижня
console.log("Остача:", день_тижня); // 1

// Перевірка парності
let число = 42;
if (число % 2 === 0) {
    console.log("Парне число");
} else {
    console.log("Непарне число");
}

```

Операції інкременту та декременту мають дві форми: префіксну та постфіксну, які відрізняються моментом зміни значення. Префіксна форма спочатку змінює значення, а потім повертає нове, тоді як постфіксна спочатку повертає поточне значення, а потім змінює його.

```

let x = 5;
console.log(++x); // 6 (спочатку збільшили, потім
показали)
console.log(x++); // 6 (спочатку показали, потім
збільшили)
console.log(x); // 7 (тепер x дорівнює 7)

```

Об'єкт `Math` містить корисні константи, які часто використовуються в математичних обчисленнях. Число  $\pi$  (пі) незамінне при роботі з колами та тригонометрією, число  $e$  використовується в експоненціальних функціях, а різні логарифмічні константи допомагають в складних обчисленнях.

```

let радіус = 10;
let довжина_кола = 2 * Math.PI * радіус;
console.log(`Довжина кола: ${довжина_кола.toFixed(2)}
см`);

```

Функції округлення мають різне призначення залежно від потреб. `Math.ceil` завжди округлює вгору до найближчого цілого, що корисно, наприклад, при розрахунку кількості сторінок або контейнерів. `Math.floor` округлює вниз і часто використовується для генерації випадкових цілих чисел. `Math.round` виконує математичне округлення до найближчого цілого, а `Math.trunc` просто відкидає дробову частину.

```
let ціна = 99.67;
console.log("До сплати:", Math.ceil(ціна)); // 100
(завжди округлюємо вгору для розрахунків)

let кількість_товару = 15.8;
console.log("Цілих одиниць:",
Math.floor(кількість_товару)); // 15
```

Тригонометричні функції особливо важливі при створенні анімацій, ігор та графічних ефектів. JavaScript використовує радіани замість градусів, тому часто потрібно перетворювати кути.

```
function градуси_в_радіани(градуси) {
    return градуси * Math.PI / 180;
}

let кут = 45;
let синус = Math.sin(градуси_в_радіани(кут));
console.log(`Синус ${кут}°: ${синус.toFixed(3)}`); //
0.707
```

Генерація випадкових чисел базується на функції `Math.random()`, яка повертає випадкове дробове число від 0 до 1. Для отримання цілих чисел у певному діапазоні потрібно використовувати додаткові обчислення з функціями округлення.

```
// Випадкове число від 1 до 100
let випадкове = Math.floor(Math.random() * 100) + 1;
```

```

console.log("Випадкове число:", випадкове);

// Вибір випадкового елемента
let міста = ["Київ", "Львів", "Одеса", "Харків"];
let випадкове_місто = міста[Math.floor(Math.random() *
міста.length)];
console.log("Випадкове місто:", випадкове_місто);

```

Практичне застосування математичних функцій часто передбачає комбінування кількох операцій для розв'язання реальних завдань. Наприклад, при розрахунку фінансових показників, створенні ігрових механік або аналізі даних.

```

// Розрахунок складних відсотків
function складні_відсотки(початкова_сума, річна_ставка,
років) {
    return початкова_сума * Math.pow(1 + річна_ставка /
100, років);
}

let результат = складні_відсотки(10000, 5, 10);
console.log(`Через 10 років: ${результат.toFixed(2)}
грн`);

// Обчислення відстані для GPS координат (спрощена
формула)
function відстань_gps(lat1, lon1, lat2, lon2) {
    let dx = lat2 - lat1;
    let dy = lon2 - lon1;
    return Math.sqrt(dx * dx + dy * dy) * 111; //
Приблизно км
}

```

Математичний арсенал JavaScript дозволяє створювати складні обчислювальні програми, від простих калькуляторів до систем аналізу даних та наукових додатків.

### 3.1.10 Функції

Функції є одним з найважливіших концептів в JavaScript, оскільки вони дозволяють організувати код у логічні блоки, які можна повторно використовувати. Функція - це блок коду, який виконує конкретне завдання і може бути викликаний багато разів з різними параметрами.

Функції в JavaScript являють собою фундаментальні будівельні блоки мови програмування, які дозволяють розробникам структурувати код таким чином, щоб він був читабельним, зрозумілим та багаторазово використовуваним. Коли ми говоримо про функції, важливо розуміти, що це не просто механізм для групування інструкцій, а потужний інструмент, який дозволяє створювати абстракції та модульну архітектуру програми.

#### Оголошення та виклик функцій

Стандартне оголошення функції в JavaScript використовує ключове слово `function`, після якого вказується ім'я функції та список параметрів у круглих дужках. Тіло функції розміщується у фігурних дужках і містить код, який буде виконано при виклику функції. Наприклад, функція привітатиКористувача приймає два параметри - ім'я та прізвище користувача, і формує персоналізоване привітання:

```
function привітатиКористувача(ім_я, прізвище) {  
    console.log(`Доброго дня, ${ім_я} ${прізвище}!`);  
    return `Привіт від ${ім_я}`;  
}
```

Виклик функції здійснюється через вказання її імені з передачею аргументів у круглих дужках, що дозволяє отримати результат виконання та використати його в подальшій роботі програми.

## Параметри за замовчуванням

Сучасний JavaScript підтримує задання значень за замовчуванням для параметрів функції, що робить її більш гнучкою у використанні. Функція `створитиПрофіль` демонструє, як можна встановити типові значення для віку та міста, що дозволяє викликати функцію з неповним набором аргументів:

```
function створитиПрофіль(ім_я, вік = 18, місто =
"Київ") {
    return {
        повне_ім_я: ім_я,
        років: вік,
        місце_проживання: місто,
        створено: new Date().toLocaleDateString()
    };
}
```

Такий підхід значно спрощує використання функцій у ситуаціях, коли не всі параметри є обов'язковими для виконання основного завдання функції.

## Область видимості змінних

Область видимості (scope) змінних є критично важливим концептом для розуміння поведінки JavaScript програм. Змінні, оголошені поза функціями, мають глобальну область видимості і доступні в будь-якій частині програми, тоді як змінні, створені всередині функції, мають локальну область видимості і доступні лише в межах цієї функції та її вкладених функцій:

```
let глобальна_змінна = "Доступна скрізь";

function зовнішняФункція() {
    let зовнішня_змінна = "Доступна в зовнішній та
вкладених";

    function внутрішняФункція() {
        let внутрішня_змінна = "Доступна тільки тут";
```



```

        console.log(глобальна_змінна); // Працює
        console.log(зовнішня_змінна); // Працює
    }
}

```

Цей механізм дозволяє створювати ізольовані блоки коду, які не впливають на інші частини програми та забезпечують інкапсуляцію даних.

### Повернення значень та раннє завершення

Оператор `return` не лише дозволяє функції повертати результат обчислень, але й негайно завершує виконання функції, що робить можливим створення функцій з умовною логікою та раннім завершенням. Функція `перевіритиВік` демонструє, як використання множинних операторів `return` створює чітку та зрозумілу логіку категоризації:

```

function категоризуватиВік(вік) {
    if (вік < 0) return "Некоректні дані";
    if (вік < 18) return "Неповнолітній";
    if (вік < 65) return "Працездатний";
    return "Пенсійний вік";
}

```

Такий підхід робить код більш читабельним та дозволяє уникнути складних вкладених умов.

### Рекурсія як програмістський прийом

Рекурсія представляє собою елегантний спосіб розв'язання задач, які можна розкласти на подібні підзадачі меншого розміру. Рекурсивна функція обов'язково повинна мати базовий випадок - умову, за якої функція припиняє викликати саму себе, інакше виникне нескінченна рекурсія та переповнення стеку.

Класичним прикладом рекурсії є обчислення факторіала числа, де факторіал  $n$  дорівнює  $n$  помножити на факторіал  $(n-1)$ :

```

function факторіал(n) {
    if (n <= 1) return 1; // Базовий випадок
}

```

```
return n * факторіал(n - 1); // Рекурсивний виклик
}
```

Більш практичним прикладом може бути функція для пошуку значення у вкладеній структурі даних, наприклад, у багаторівневому об'єкті JSON:

```
function пошукВОб_єкті(об_єкт, шуканий_ключ) {
    for (let ключ in об_єкт) {
        if (ключ === шуканий_ключ) {
            return об_єкт[ключ];
        }

        if (typeof об_єкт[ключ] === 'object' &&
об_єкт[ключ] !== null) {
            let знайдене = пошукВОб_єкті(об_єкт[ключ],
шуканий_ключ);
            if (знайдене !== undefined) return
знайдене;
        }
    }
    return undefined;
}
```

Ця функція ілюструє, як рекурсія може бути використана для обходу складних структур даних без необхідності знати їхню глибину наперед.

Розуміння функцій та їхніх властивостей є основоположним для написання ефективного та підтримуваного JavaScript коду, оскільки вони забезпечують основу для організації логіки програми та створення повторно використовуваних компонентів.

### 3.1.11 Функціональні вирази

Функціональні вирази представляють собою альтернативний спосіб створення функцій у JavaScript, який принципово відрізняється від звичайного

оголошення функцій своєю поведінкою під час виконання програми. Основна ідея полягає в тому, що функція створюється як значення, яке присвоюється змінній, а не як окрема конструкція мови.

```
// Звичайне оголошення функції
function привітання(ім_я) {
    return `Привіт, ${ім_я}!`;
}

// Функціональний вираз
let привітання = function(ім_я) {
    return `Привіт, ${ім_я}!`;
};
```

Найпоширенішим типом є анонімний функціональний вираз, де функція не має власного імені та ідентифікується виключно через змінну, до якої вона присвоєна:

```
let обчислити = function(x, y) {
    return x * y + 10;
};
console.log(обчислити(5, 3)); // 25
```

Іменовані функціональні вирази дозволяють надати функції ім'я, яке можна використовувати для рекурсивних викликів або кращої читабельності коду:

```
let факторіал = function обчислити_факторіал(n) {
    if (n <= 1) return 1;
    return n * обчислити_факторіал(n - 1); //
рекурсивний виклик
};
console.log(факторіал(5)); // 120
```

Найважливішою відмінністю між оголошенням функцій та функціональними виразами є поведінка підняття. Звичайні функції

"піднімаються" на початок їх області видимості, що означає можливість їх виклику до фактичного оголошення в коді:

```
// Це працює завдяки hoisting
console.log(піднята_функція()); // "Я доступна!"

function піднята_функція() {
    return "Я доступна!";
}

// А це призведе до помилки
console.log(не_піднята()); // TypeError: не_піднята is
not a function

var не_піднята = function() {
    return "Мене створили пізніше";
};
```

Функціональні вирази дозволяють створювати функції умовно, що відкриває можливості для динамічної поведінки програми залежно від умов виконання:

```
let режим_розробки = true;

let логувати = режим_розробки
  ? function(повідомлення) {
      console.log(`[DEBUG]: ${повідомлення}`);
    }
  : function(повідомлення) {
      // В продакшні не логуємо
    };

```

```
логувати("Програма запущена"); // "[DEBUG]: Програма запущена"
```

Одним з найпотужніших застосувань функціональних виразів є їх використання як параметрів для інших функцій, що створює основу для функціонального програмування:

```
function виконатиОперацію(масив, операція) {
    let результат = [];
    for (let елемент of масив) {
        результат.push(операція(елемент));
    }
    return результат;
}

// Створюємо різні операції
let квадрат = function(x) { return x * x; };
let куб = function(x) { return x * x * x; };

let числа = [2, 3, 4];
console.log(виконатиОперацію(числа, квадрат)); // [4, 9, 16]
console.log(виконатиОперацію(числа, куб)); // [8, 27, 64]
```

Замикання представляє собою одну з найскладніших та найпотужніших концепцій JavaScript, яка дозволяє внутрішнім функціям зберігати доступ до змінних зовнішньої функції навіть після завершення її виконання:

```
function створитиМножник(множник) {
    // Ця змінна "запам'ятається" внутрішньою функцією
    return function(число) {
        return число * множник;
    };
};
```

```

}

let подвоїти = створитиМножник(2);
let потроїти = створитиМножник(3);

console.log(подвоїти(5)); // 10
console.log(потроїти(5)); // 15

```

Практичний приклад замикання: система конфігурації:

```

function створитиКонфігуратор(початкова_конфігурація) {
    let конфігурація = { ...початкова_конфігурація };

    return {
        отримати: function(ключ) {
            return конфігурація[ключ];
        },
        встановити: function(ключ, значення) {
            конфігурація[ключ] = значення;
        },
        отримати_всі: function() {
            return { ...конфігурація };
        }
    };
}

let налаштування = створитиКонфігуратор({
    мова: 'українська',
    тема: 'світла'
});

```

```
console.log(налаштування.отримати('мова')); //
'українська'
налаштування.встановити('тема', 'темна');
console.log(налаштування.отримати_всі()); // {мова:
'українська', тема: 'темна'}
```

Негайно викликувані функціональні вирази (IIFE - Immediately Invoked Function Expression) створюють ізольований простір імен та дозволяють реалізувати модульний патерн:

```
let модуль_користувача = (function() {
    // Приватні змінні та функції
    let користувачі = [];

    function валідувати_дані(дані) {
        return дані.ім_я && дані.вік > 0;
    }

    // Публічний інтерфейс модуля
    return {
        додати: function(дані) {
            if (валідувати_дані(дані)) {
                користувачі.push(дані);
                return true;
            }
            return false;
        },

        отримати_кількість: function() {
            return користувачі.length;
        },
    };
});
```

```

        отримати_всіх: function() {
            return [...користувачі]; // повертаємо
копію
        }
    };
})();

// Використання модуля
модуль_користувача.додати({ ім_я: "Олексій", вік: 25
});
console.log(модуль_користувача.отримати_кількість());
// 1

```

Функціональні вирази є основою для створення функцій вищого порядку - функцій, які приймають інші функції як параметри або повертають функції як результат:

```

function створити_фільтр(умова) {
    return function(масив) {
        let результат = [];
        for (let елемент of масив) {
            if (умова(елемент)) {
                результат.push(елемент);
            }
        }
        return результат;
    };
}

// Створюємо різні фільтри
let парні = створити_фільтр(function(x) { return x % 2
=== 0; });

```



```
let великі = створити_фільтр(function(x) { return x >
10; });
```

```
let числа = [1, 2, 3, 12, 15, 18, 20];
console.log(парні(числа)); // [2, 12, 18, 20]
console.log(великі(числа)); // [12, 15, 18, 20]
```

### Практичні застосування - система обробки подій:

```
let система_подій = (function() {
    let обробники = {};

    return {
        підписатися: function(тип_події, обробник) {
            if (!обробники[тип_події]) {
                обробники[тип_події] = [];
            }
            обробники[тип_події].push(обробник);
        },

        викликати_подію: function(тип_події, дані) {
            if (обробники[тип_події]) {
                обробники[тип_події].forEach(function(обробник) {
                    обробник(дані);
                });
            }
        }
    };
})();

// Використання
```

```

система_подій.підписатися('login', function(користувач)
{
    console.log(`Користувач ${користувач} увійшов до
системи`);
});

система_подій.викликати_подію('login', 'Марія'); //
"Користувач Марія увійшов до системи"

```

Функціональні вирази надають програмісту більшу гнучкість у структурування коду, дозволяючи створювати функції тільки тоді, коли вони справді потрібні, що може покращити продуктивність програми. Вони також сприяють створенню більш модульного та організованого коду, особливо при роботі з асинхронними операціями та обробкою подій.

Розуміння функціональних виразів є критично важливим для освоєння сучасного JavaScript, оскільки вони лежать в основі багатьох популярних патернів програмування, включаючи стрілочні функції, промиси та `async/await` синтаксис.

### 3.1.12 Стрілкові функції в JavaScript

Стрілкові функції представляють собою революційне нововведення в JavaScript, яке не просто змінило синтаксис мови, а кардинально вплинуло на стиль написання сучасного коду. Ця концепція виникла з практичної потреби розробників у більш виразному та лаконічному способі створення функцій, особливо в контексті функціонального програмування.

Основна філософія стрілкових функцій полягає у спрощенні синтаксису для найпоширеніших сценаріїв використання функцій. Замість громіздкого запису з ключовим словом `function`, програмісти отримали можливість використовувати символ `=>`, який нагадує стрілку та інтуїтивно показує напрямок передачі даних від параметрів до результату.

Особливо цінними стрілкові функції стають при роботі з функціональними методами масивів, такими як `map`, `filter`, `reduce`, де традиційний синтаксис часто створював надлишок коду, що ускладнював розуміння основної логіки. Наприклад, замість написання `array.map(function(x) { return x * 2; })` ми можемо просто написати `array.map(x => x * 2)`, що значно покращує читабельність.

Концепція неявного повернення значення робить стрілкові функції особливо привабливими для створення простих трансформацій та обчислень. Коли тіло функції складається з одного виразу, результат цього виразу автоматично повертається без явного використання `return`, що економить місце та робить код більш виразним.

Проте важливо розуміти, що стрілкові функції не є простою заміною традиційних функцій - вони мають власні особливості поведінки, особливо щодо контексту `this`, що робить їх непридатними для певних сценаріїв використання, зокрема як методи об'єктів або конструктори.

При вивченні стрілкових функцій студентам варто усвідомити, що це не просто синтаксичний цукор, а потужний інструмент, який відкриває нові можливості для написання елегантного та функціонального коду, особливо в контексті сучасних JavaScript фреймворків та бібліотек.

Стрілкові функції з'явилися в JavaScript з випуском стандарту ECMAScript 6 (ES2015) як відповідь на потребу розробників у більш лаконічному та зручному синтаксисі для створення функцій. До їх появи програмісти часто стикалися з надмірною багатослівністю коду, особливо при роботі з функціями зворотного виклику та методами масивів, що ускладнювало читання та підтримку програм.

```
// До ES6 - багатослівно та незручно
let числа = [1, 2, 3, 4, 5];
let подвоєні = числа.map(function(число) {
    return число * 2;
});
```

```
// З ES6 - лаконічно та зрозуміло
let подвоєні = числа.map(число => число * 2);
```

### Синтаксичні варіації

Стрілкові функції пропонують декілька варіантів синтаксису залежно від кількості параметрів та складності логіки функції, що дозволяє писати код максимально виразно та компактно.

Коли функція приймає рівно один параметр, дужки навколо нього можна опустити, що робить код ще більш лаконічним:

```
let зробити_великими = текст => текст.toUpperCase();
let отримати_довжину = рядок => рядок.length;
let є_парним = число => число % 2 === 0;

console.log(зробити_великими("україна")); // "УКРАЇНА"
console.log(є_парним(8)); // true
```

При наявності декількох параметрів або їх відсутності обов'язково використовуються круглі дужки:

```
let обчислити_площу = (довжина, ширина) => довжина *
    ширина;
let привітати_всіх = () => "Привіт усім присутнім!";
let створити_повідомлення = (автор, текст, час) =>
    `${час}: ${автор} написав "${текст}"`;

console.log(обчислити_площу(10, 5)); // 50
console.log(створити_повідомлення("Олена", "Добрий
ранок!", "08:30"));
```

### Неявне та явне повернення значень

Однією з найцікавіших особливостей стрілкових функцій є можливість неявного повернення значення, коли результат обчислення автоматично повертається без використання ключового слова `return`:

```
// неявне повернення - без фігурних дужок
let подвоїти = x => x * 2;
let привітання = ім_я => `Вітаю, ${ім_я}!`;

// Явне повернення - з фігурними дужками та return
let складна_функція = (a, b) => {
    let проміжний_результат = a * b;
    let фінальний_результат = проміжний_результат + 10;
    return фінальний_результат;
};

console.log(подвоїти(7)); // 14
console.log(складна_функція(3, 4)); // 22
```

При поверненні літерала об'єкта потрібно обгорнути його в круглі дужки, щоб JavaScript не сприйняв фігурні дужки як початок блоку коду:

```
// Неправильно - буде помилка
// let створити_особу = (ім_я, вік) => {ім_я: ім_я,
вік: вік};

// Правильно - об'єкт обгорнутий в дужки
let створити_особу = (ім_я, вік) => ({ім_я, вік,
активний: true});
let створити_координати = (x, y) => ({x, y, час:
Date.now()});

console.log(створити_особу("Марія", 22));
// {ім_я: "Марія", вік: 22, активний: true}
```

### Застосування з методами масивів

Стрілкові функції особливо елегантно виглядають при роботі з функціональними методами масивів, роблячи код більш читабельним та виразним:

```
let продукти = [
  {назва: "хліб", ціна: 25, категорія: "випічка"},
  {назва: "молоко", ціна: 30, категорія: "молочні"},
  {назва: "сир", ціна: 120, категорія: "молочні"},
  {назва: "батон", ціна: 20, категорія: "випічка"}
];

// Фільтрування дешевих товарів
let дешеві = продукти.filter(товар => товар.ціна < 50);

// Отримання тільки назв молочних продуктів
let молочні_назви = продукти
  .filter(товар => товар.категорія === "молочні")
  .map(товар => товар.назва);

// Розрахунок загальної вартості випічки
let вартість_випічки = продукти
  .filter(товар => товар.категорія === "випічка")
  .reduce((сума, товар) => сума + товар.ціна, 0);

console.log("Молочні продукти:", молочні_назви); //
["молоко", "сир"]
console.log("Вартість випічки:", вартість_випічки); //
45
```

### Комбінування з деструктуризацією

Стрілкові функції прекрасно поєднуються з деструктуризацією параметрів, що дозволяє писати надзвичайно виразний та зрозумілий код:

```

let студенти = [
    {ім_я: "Олександр", група: "ІТ-301", середній_бал:
4.5},
    {ім_я: "Катерина", група: "ІТ-302", середній_бал:
4.8},
    {ім_я: "Дмитро", група: "ІТ-301", середній_бал:
4.2}
];

// Деструктуризація в параметрах стрілкової функції
let отримати_інформацію = ({ім_я, група, середній_бал})
=>
    `${ім_я} (${група}) - середній бал:
${середній_бал}`;

// Фільтрування відмінників з групи ІТ-301
let відмінники_301 = студенти
    .filter(({група, середній_бал}) => група === "ІТ-
301" && середній_бал >= 4.5)
    .map(({ім_я}) => ім_я);

console.log(відмінники_301); // ["Олександр"]

```

### Створення функцій вищого порядку

Стрілкові функції надзвичайно зручні для створення функцій, які повертають інші функції, що є основою функціонального програмування:

```

// Фабрика валідаторів
let створити_валідатор = (мін, макс) => значення =>
    значення >= мін && значення <= макс;

// Фабрика форматтерів

```

```

let створити_форматтер = префікс => текст =>
  `${префікс}: ${текст}`;

// Використання
let валідатор_віку = створити_валідатор(18, 65);
let форматтер_помилки = створити_форматтер("ПОМИЛКА");
let форматтер_успіху = створити_форматтер("УСПІХ");

console.log(валідатор_віку(25)); // true
console.log(валідатор_віку(70)); // false
console.log(форматтер_помилки("Неправильний вік"));
// "ПОМИЛКА: Неправильний вік"

```

### Асинхронне програмування та стрілкові функції

В контексті асинхронного програмування стрілкові функції значно спрощують роботу з промисами та callback-функціями:

```

// Імітація асинхронної операції
let завантажити_дані = () =>
  new Promise(resolve =>
    setTimeout(() => resolve("Дані завантажені"),
      2000));

// Обробка промисів з стрілковими функціями
завантажити_дані()
  .then(результат => console.log(результат))
  .then(() => console.log("Обробка завершена"))
  .catch(помилка => console.log(`Помилка:
  ${помилка}`));

// Створення ланцюжка обробки даних
let обробити_користувачів = користувачі =>

```



```

    Promise.resolve(користувачі)
      .then(дані => дані.filter(користувач =>
користувач.активний))
      .then(активні => активні.map(користувач =>
користувач.ім_я))
      .then(імена => імена.sort());

```

### Практичні патерни використання

Створення утилітних функцій:

Стрілкові функції ідеально підходять для створення коротких утилітних функцій:

```

// Математичні утиліти
let абсолютне_значення = x => x < 0 ? -x : x;
let є_додатним = x => x > 0;
let обмежити = (значення, мін, макс) =>
Math.min(Math.max(значення, мін), макс);

// Рядкові утиліти
let скоротити = (текст, довжина) =>
  текст.length > довжина ? текст.slice(0, довжина) +
"...": текст;
let капіталізувати = слово =>
  слово.charAt(0).toUpperCase() +
слово.slice(1).toLowerCase();

console.log(скоротити("Дуже довгий текст", 10)); //
"Дуже довги..."
console.log(капіталізувати("jAVAScript")); //
"Javascript"

```

Функціональні ланцюжки обробки даних:

```

let замовлення = [

```

```

    {продукт: "Ноутбук", ціна: 25000, кількість: 1,
статус: "доставлено"},
    {продукт: "Миша", ціна: 500, ціна: 2, кількість:
статус: "в обробці"},
    {продукт: "Клавіатура", ціна: 1500, кількість: 1,
статус: "доставлено"}
];

// Розрахунок загальної вартості доставлених товарів
let загальна_вартість = замовлення
    .filter(({статус}) => статус === "доставлено")
    .map(({ціна, кількість}) => ціна * кількість)
    .reduce((сума, вартість) => сума + вартість, 0);

console.log(`Загальна вартість: ${загальна_вартість}
грн`);

```

Незважаючи на свою зручність, стрілкові функції мають певні обмеження, які важливо розуміти для правильного їх використання. Вони не мають власного контексту `this`, не можуть використовуватися як конструктори та не мають доступу до об'єкта `arguments`.

```

// Стрілкові функції успадковують this з зовнішнього
контексту
let об_єкт = {
    ім_я: "Тестовий об'єкт",

    традиційна_функція: function() {
        return this.ім_я; // this вказує на об'єкт
    },

    стрілкова_функція: () => {

```

```

        return this.ім_я; // this вказує на глобальний
об'єкт
    }
};

console.log(об_єкт.традиційна_функція()); // "Тестовий
об'єкт"
console.log(об_єкт.стрілкова_функція()); // undefined

```

Стрілкові функції найкраще підходять для коротких функцій-утилітарів, callback-функцій, роботи з масивами та функціонального програмування. Вони роблять код більш читабельним та зменшують кількість синтаксичного шуму, дозволяючи розробникам зосередитися на логіці програми замість деталей синтаксису.

Водночас традиційні функції залишаються кращим вибором для методів об'єктів, конструкторів, складних функцій з багатьма рядками коду та випадків, коли потрібен власний контекст `this`.

### 3.1.13 Об'єкти

Об'єкт у JavaScript - це складна структура даних, яка представляє собою колекцію пар "ключ-значення", де кожен ключ є рядком (або символом), а значення може бути будь-яким типом даних: числом, рядком, масивом, функцією або навіть іншим об'єктом. Це дозволяє нам групувати пов'язані між собою дані та функціональність в одну логічну одиницю, що робить код більш організованим та легким для розуміння.

Найпростіший спосіб створити об'єкт - використовувати літеральний синтаксис з фігурними дужками:

```

let студент = {
    ім_я: "Олена",
    вік: 20,
    курс: 3
}

```

```
};
```

У цьому прикладі ми створили об'єкт студент, який містить три властивості: ім'я, вік та курс, кожна з яких має своє значення.

### Способи доступу до властивостей об'єкта

Існує два основних способи отримати доступ до властивостей об'єкта: через точкову нотацію та через квадратні дужки. Точкова нотація є найпоширенішою та найзручнішою для статичних властивостей:

```
console.log(студент.ім_я); // "Олена"
console.log(студент.вік); // 20
```

Квадратні дужки корисні, коли ім'я властивості зберігається у змінній або містить спеціальні символи:

```
let властивість = "курс";
console.log(студент[властивість]); // 3
```

### Методи об'єктів та ключове слово this

Методи - це функції, які є властивостями об'єкта, і вони дозволяють об'єктам виконувати дії з їх власними даними. Ключове слово this всередині методу посилається на сам об'єкт, що дає можливість методам працювати з властивостями цього об'єкта:

```
let рахунок = {
  баланс: 1000,

  зняти(сума) {
    if (this.баланс >= сума) {
      this.баланс -= сума;
      console.log(`Знято ${сума} грн. Залишок:
${this.баланс} грн`);
    } else {
      console.log("Недостатньо коштів");
    }
  },
}
```

```

    поповнити(сума) {
        this.баланс += сума;
        console.log(`Поповнено на ${сума} грн. Баланс:
${this.баланс} грн`);
    }
};

рахунок.зняти(300);    // "Знято 300 грн. Залишок: 700
грн"
рахунок.поповнити(500); // "Поповнено на 500 грн.
Баланс: 1200 грн"

```

### Перебір властивостей об'єкта

JavaScript надає кілька способів перебрати всі властивості об'єкта, що особливо корисно при роботі з динамічними даними. Цикл `for...in` перебирає всі перелічувані властивості об'єкта:

```

let оцінки = {
    математика: 5,
    фізика: 4,
    хімія: 5,
    історія: 3
};

for (let предмет in оцінки) {
    console.log(`${предмет}: ${оцінки[предмет]}`);
}

```

Для більш сучасного підходу використовуються методи `Object.keys()`, `Object.values()` та `Object.entries()`, які повертають масиви та дозволяють використовувати всі методи масивів:

```
// Object.keys() - отримує масив ключів
```

```

let предмети = Object.keys(оцінки);
console.log(предмети); // ["математика", "фізика",
"хімія", "історія"]

// Object.values() - отримує масив значень
let бали = Object.values(оцінки);
let середній_бал = бали.reduce((сума, бал) => сума +
бал, 0) / бали.length;
console.log(`Середній бал: ${середній_бал}`);

```

### Копіювання та об'єднання об'єктів

При роботі з об'єктами часто виникає потреба створювати копії або об'єднувати кілька об'єктів в один. Оператор розпакування (spread operator) ... дозволяє легко створювати поверхневі копії об'єктів:

```

let базові_налаштування = {
  мова: "українська",
  тема: "світла"
};

let користувацькі_налаштування = {
  ...базові_налаштування,
  тема: "темна", // Перекриває базове значення
  розмір_шрифту: 14
};

console.log(користувацькі_налаштування);
// { мова: "українська", тема: "темна", розмір_шрифту:
14 }

```

### Деструктуризація об'єктів

Деструктуризація - це потужна можливість JavaScript, яка дозволяє витягувати значення з об'єктів у окремі змінні одним рядком коду, що робить код більш читабельним та лаконічним:

```
let викладач = {
    ім_я: "Іван",
    прізвище: "Петренко",
    кафедра: "Інформатики",
    стаж: 15
};

// Витягуємо потрібні властивості у змінні
let { ім_я, кафедра, стаж } = викладач;
console.log(`${ім_я} працює на кафедрі ${кафедра}
${стаж} років`);

// Деструктуризація з перейменуванням
let { ім_я: firstName, прізвище: lastName } = викладач;
```

### Практичне застосування

Об'єкти широко використовуються для моделювання реальних сутностей у програмах. Наприклад, ми можемо створити систему управління курсами університету:

```
let курс = {
    назва: "Технології Інтернет",
    код: "ТІ-301",
    кредити: 5,
    студенти: [],

    зареєструватиСтудента(ім_я, група) {
        let студент = {
            ім_я: ім_я,
```

```

        група: група,
        оцінки: [],
        дата_реєстрації: new Date()
    };
    this.студенти.push(студент);
    console.log(`Студент ${ім_я} зареєстрований на
курс`);
    },

    отриматиКількістьСтудентів() {
        return this.студенти.length;
    }
};

курс.zareestruvatiStudenta("Марія", "КН-31");
курс.zareestruvatiStudenta("Андрій", "КН-32");
console.log(`На курсі зареєстровано
${курс.отриматиКількістьСтудентів()} студентів`);

```

Об'єкти в JavaScript надзвичайно гнучкі та дозволяють створювати складні структури даних, які відображають реальний світ у вашому коді. Вони є основою для розуміння більш складних концепцій, таких як класи, прототипи та архітектурні патерни у веб-розробці.

## 3.2 Браузер: документ

Сучасні веб-браузери представляють собою складні програмні комплекси, які забезпечують виконання JavaScript коду та відображення веб-сторінок. Для розробників JavaScript найважливішими є можливості взаємодії з документом через спеціальні програмні інтерфейси. У цьому розділі ми детально розглянемо, як JavaScript може керувати структурою документа,



змінювати його вміст, стилі та поведінку через Document Object Model (DOM) та Browser Object Model (BOM).

### **3.2.1 Браузерне середовище та специфікації**

Браузерне середовище виконання JavaScript кардинально відрізняється від інших середовищ, таких як серверне середовище Node.js або мобільні додатки. Ця відмінність полягає не лише в доступних API, але й у самій філософії роботи з кодом та користувачем. У браузері JavaScript виконується в унікальному контексті, де головною метою є забезпечення інтерактивної взаємодії з користувачем через веб-сторінку, яка відображається на екрані.

Браузерне середовище надає JavaScript доступ до потужних можливостей візуалізації та взаємодії, які недоступні в інших середовищах. Код може безпосередньо впливати на те, що бачить користувач, змінювати кольори, розміри, положення елементів, створювати анімації та реагувати на найрізноманітніші дії користувача - від простих кліків миші до складних жестів на сенсорних екранах.

Важливою особливістю браузерного середовища є його багатозадачність та відзивність. На відміну від традиційних програм, які можуть "зависнути" під час виконання складних обчислень, браузерні додатки повинні постійно залишатися готовими до взаємодії з користувачем, навіть коли виконуються фонові операції.

Пісочниця безпеки (Sandbox) являє собою одну з найважливіших особливостей браузерного середовища, яка забезпечує захист користувача від потенційно шкідливого коду. Браузер створює ізольоване середовище виконання, де JavaScript код не може отримати прямий доступ до файлової системи комп'ютера, встановлених програм або системних ресурсів. Ця ізоляція означає, що веб-сторінка не може читати файли з жорсткого диска користувача без його явної згоди, не може встановлювати програми або змінювати системні налаштування.

```
// Це НЕ працює в браузері - немає доступу до файлової
системи
// const fs = require('fs'); // Помилка!

// Але можна отримати доступ до файлів через File API з
дозволу користувача
document.querySelector('#file-
input').addEventListener('change', function(event) {
    const file = event.target.files[0];
    if (file) {
        const reader = new FileReader();
        reader.onload = function(e) {
            console.log('Вміст файлу:',
e.target.result);
        };
        reader.readAsText(file);
    }
});
```

Водночас пісочниця не є абсолютною - браузери надають контрольовані способи доступу до певних ресурсів через спеціальні API, такі як File API, Geolocation API, Camera API, але завжди з дозволу користувача та з відповідними обмеженнями.

Подієвий цикл (Event Loop) є серцем браузерного середовища, який забезпечує асинхронну природу JavaScript виконання. На відміну від синхронних мов програмування, де код виконується послідовно рядок за рядком, JavaScript у браузері використовує асинхронну модель, яка дозволяє виконувати кілька операцій одночасно без блокування користувацького інтерфейсу.

```
// Демонстрація асинхронної природи
console.log('1. Початок виконання');
```

```

setTimeout(() => {
    console.log('3. Асинхронна операція завершена');
}, 0);

console.log('2. Синхронний код продовжується');

// Вивід: 1 → 2 → 3, хоча setTimeout має затримку 0

```

Подієвий цикл постійно перевіряє черги подій та виконує callback функції, коли настає їхній час. Це дозволяє браузеру обробляти кліки користувача, завантажувати дані з сервера, виконувати анімації та інші операції, при цьому не "заморожуючи" інтерфейс.

Глобальний об'єкт window служить кореневим контейнером для всього браузерного середовища. У серверному Node.js глобальним об'єктом є global, але в браузері цю роль виконує window. Всі глобальні змінні, функції та об'єкти автоматично стають властивостями об'єкта window.

```

// Ці записи еквівалентні в браузері
var userName = 'Іван';
window.userName = 'Іван';

function greetUser() {
    return 'Привіт!';
}

// Функція також стає властивістю window
console.log(window.greetUser()); // "Привіт!"

// Доступ до браузерних API через window
console.log(window.location.href); // поточна URL
console.log(window.navigator.userAgent); // інформація про браузер

```

Розробка веб-технологій координується кількома міжнародними організаціями, які створюють та підтримують стандарти, що забезпечують сумісність між різними браузерами та платформами.

W3C (World Wide Web Consortium) є найстарішою та найавторитетнішою організацією у сфері веб-стандартів. Заснована Тімом Бернерсом-Лі, винахідником World Wide Web, ця організація відповідає за розробку фундаментальних стандартів веб-технологій. W3C працює через систему робочих груп, де представники різних компаній та організацій спільно розробляють специфікації.

WHATWG (Web Hypertext Application Technology Working Group) виникла як альтернатива W3C у відповідь на потребу в швидшому розвитку веб-стандартів. Організація була створена представниками основних браузерних компаній (Apple, Mozilla, Opera) і сьогодні активно співпрацює з Google. WHATWG відома своїм підходом "living standards" - стандартами, які постійно еволюціонують замість того, щоб фіксуватися у певних версіях.

HTML5 представляє собою кардинальне оновлення мови розмітки, яке перетворило HTML з простої мови структурування документів у потужну платформу для створення веб-додатків. Специфікація HTML5 визначає не лише нові семантичні елементи (такі як <article>, <section>, <nav>), але й численні JavaScript API для роботи з мультимедіа, локальним сховищем даних, геолокацією та іншими можливостями сучасних пристроїв.

```
<!-- Семантичні елементи HTML5 -->
<article>
  <header>
    <h1>Заголовок статті</h1>
    <time datetime="2024-03-15">15 березня
2024</time>
  </header>
  <section>
    <p>Основний вміст статті...</p>
```

```

    </section>
    <footer>
        <p>Автор: Іван Петренко</p>
    </footer>
</article>

```

```

// HTML5 API для геолокації
navigator.geolocation.getCurrentPosition(function(position) {
    console.log('Широта:', position.coords.latitude);
    console.log('Довгота:', position.coords.longitude);
});

```

```

// HTML5 API для локального сховища
localStorage.setItem('userName', 'Марія');
const savedName = localStorage.getItem('userName');

```

DOM Level 4 є сучасною версією специфікації Document Object Model, яка стандартизує програмний інтерфейс для роботи з структурою документів. Ця специфікація визначає, як браузері повинні представляти HTML документи у вигляді об'єктної моделі, доступної для JavaScript маніпуляцій. DOM Level 4 включає нові методи пошуку елементів, покращені можливості навігації по дереву документа та розширені можливості обробки подій.

```

// Сучасні методи DOM Level 4
const elements =
document.querySelectorAll('.item:not(.hidden)');
elements.forEach(el => el.classList.add('processed'));

// Нові можливості вставки контенту
const targetElement =
document.querySelector('.target');

```

```
targetElement.insertAdjacentHTML('afterbegin',
'<span>Вставлено на початок</span>');
```

CSS3 не є єдиною специфікацією, а скоріше збірним терміном для великої кількості модулів, кожен з яких описує окремі аспекти стилізації. Ці модули включають селектори, анімації, трансформації, гнучкі макети (Flexbox), сіткові макети (Grid), медіа-запити та багато інших можливостей. CSS3 модулі розробляються незалежно один від одного, що дозволяє швидше впроваджувати нові можливості в браузері.

```
/* Приклади можливостей CSS3 */
.modern-card {
    background: linear-gradient(135deg, #667eea 0%,
#764ba2 100%);
    border-radius: 12px;
    box-shadow: 0 10px 25px rgba(0,0,0,0.2);
    transform: translateY(0);
    transition: transform 0.3s ease;
}

.modern-card:hover {
    transform: translateY(-5px);
}

/* Grid Layout для сучасних макетів */
.grid-container {
    display: grid;
    grid-template-columns: repeat(auto-fit,
minmax(300px, 1fr));
    gap: 20px;
}
```

ECMAScript є офіційним стандартом мови JavaScript, який регулярно оновлюється новими можливостями та синтаксичними конструкціями. Починаючи з ES6 (ES2015), стандарт оновлюється щорічно, додаючи нові функції, які роблять код більш експресивним, читабельним та потужним. Сучасні браузерери швидко впроваджують нові можливості ECMAScript, але розробники часто використовують інструменти транспіляції (такі як Babel) для забезпечення сумісності зі старшими браузерами.

```
// Сучасні можливості ECMAScript
class Component {
  constructor(element) {
    this.element = element;
    this.state = new Map();
  }

  // Методи класу з стрілковими функціями
  handleClick = (event) => {
    const { target } = event; // деструктуризація
    const data = target.dataset.value ?? 'default';
    // nullish coalescing

    this.updateState(data);
  }

  async updateState(newValue) {
    // Асинхронні операції
    try {
      const response = await
fetch('/api/update');
      const result = await response.json();
      this.state.set('value', result.data);
    }
  }
}
```

```

        } catch (error) {
            console.error('Помилка оновлення:', error);
        }
    }
}

```

Глибоке розуміння веб-специфікацій дозволяє розробникам створювати надійні, продуктивні та сумісні веб-додатки. Знання стандартів допомагає передбачити поведінку коду в різних браузерах, вибирати найкращі рішення для конкретних завдань та слідувати кращим практикам розробки.

Специфікації також визначають майбутній розвиток веб-платформи. Розуміючи напрямки розвитку стандартів, розробники можуть готуватися до впровадження нових можливостей та писати код, який буде релевантним протягом тривалого часу.

```

// Приклад використання сучасних стандартів
class ModernWebComponent extends HTMLElement {
    constructor() {
        super();
        // Використання Shadow DOM (Web Components
стандарт)
        this.shadow = this.attachShadow({mode:
'open'});
    }

    connectedCallback() {
        // Custom Elements API
        this.shadow.innerHTML = `
            <style>
                :host {
                    display: block;
                    padding: 1rem;

```



```

        border: 1px solid #ccc;
    }
</style>
<slot></slot>
`
;
}
}

// Реєстрація користувацького елемента
customElements.define('modern-component',
ModernWebComponent);

```

Крім того, розуміння специфікацій допомагає розробникам ефективно використовувати інструменти розробки, такі як браузерні консолі для налагодження, профайлери продуктивності та валідатори коду. Знання стандартів також полегшує вивчення нових фреймворків та бібліотек, оскільки всі вони базуються на цих фундаментальних принципах.

### 3.2.2 Document Object Model (DOM)

Document Object Model є революційною концепцією, яка перетворила статичні HTML документи на живі, інтерактивні структури, якими можна керувати програмно. Уявіть HTML документ як архітектурний креслення будівлі - він описує структуру, але сам по собі не дозволяє змінювати цю структуру після побудови. DOM, натомість, перетворює цей статичний план на динамічну модель, де кожна частина будівлі може бути переміщена, замінена або модифікована в реальному часі.

Коли браузер завантажує HTML сторінку, він не просто відображає її на екрані - він створює внутрішню об'єктну модель цього документа, де кожен HTML елемент стає живим об'єктом JavaScript з власними методами та властивостями. Ця трансформація дозволяє розробникам взаємодіяти з

документом так, наче це звичайна структура даних у програмі, з можливістю читання, модифікації та реорганізації її елементів.

```
// DOM перетворює цей HTML:
// <div id="container">
//     <p>Привіт, світ!</p>
// </div>

// У такі JavaScript об'єкти:
const container = document.getElementById('container');
console.log(container instanceof HTMLDivElement); //
true
console.log(container.children[0] instanceof
HTMLParagraphElement); // true
```

Критично важливо розуміти, що DOM не є вбудованою частиною мови JavaScript - це зовнішній API, який надають браузері для взаємодії з документами. Це означає, що той самий JavaScript код, який працює з DOM у браузері, не буде функціонувати в серверному середовищі Node.js без спеціальних бібліотек, які емулюють DOM функціональність.

```
// У браузері це працює
const title = document.querySelector('title');
title.textContent = 'Новий заголовок';

// У Node.js потрібна додаткова бібліотека
// const { JSDOM } = require('jsdom');
// const dom = new JSDOM('<!DOCTYPE
html><html><head><title>Test</title></head></html>');
```

DOM відкриває перед розробниками безмежні можливості для створення інтерактивних веб-додатків. Читання та зміна вмісту елементів дозволяє створювати динамічні інтерфейси, які реагують на дані, отримані з сервера, або на дії користувача. Наприклад, можна створити лічильник, який

оновлюється в реальному часі, або форму, яка показує результати валідації одразу після введення даних.

```
// Динамічне оновлення вмісту
const counter = document.querySelector('.counter');
let count = 0;

setInterval(() => {
    count++;
    counter.textContent = `Лічильник: ${count}`;
}, 1000);
```

Додавання та видалення елементів перетворює статичні сторінки на живі додатки, де контент може з'являтися та зникати залежно від контексту. Це особливо важливо для односторінкових додатків (SPA), де весь інтерфейс керується JavaScript без перезавантаження сторінки.

```
// Динамічне створення списку задач
const taskList = document.querySelector('.task-list');
const tasks = ['Вивчити DOM', 'Написати код',
    'Протестувати додаток'];

tasks.forEach(taskText => {
    const taskItem = document.createElement('li');
    taskItem.textContent = taskText;
    taskItem.addEventListener('click', () => {
        taskItem.remove(); // видалення при кліку
    });
    taskList.appendChild(taskItem);
});
```

Зміна атрибутів та стилів дозволяє створювати візуально привабливі та інтерактивні інтерфейси, які реагують на дії користувача миттєвими візуальними змінами. Це може включати зміну кольорів при наведенні миші,

анімації переходів між станами або адаптацію інтерфейсу під різні розміри екранів.

```
// Інтерактивна зміна стилів
const buttons =
document.querySelectorAll('.interactive-btn');

buttons.forEach(button => {
    button.addEventListener('mouseenter', () => {
        button.style.backgroundColor = '#007bff';
        button.style.transform = 'scale(1.05)';
    });

    button.addEventListener('mouseleave', () => {
        button.style.backgroundColor = '';
        button.style.transform = '';
    });
});
```

DOM дерево складається з різних типів вузлів, кожен з яких має специфічні характеристики та призначення. Розуміння цих типів є критично важливим для ефективної роботи з документом.

Element nodes (вузли елементів) представляють HTML теги та є основними будівельними блоками веб-сторінок. Кожен елемент DOM має багатий набір властивостей та методів, які дозволяють маніпулювати його виглядом, поведінкою та вмістом. Елементи можуть містити інші елементи, текстові вузли або коментарі, створюючи ієрархічну структуру документа.

```
const divElement = document.querySelector('div');

// Властивості елемента
console.log(divElement.tagName);           // "DIV"
```

```

console.log(divElement.id);           // значення
атрибута id
console.log(divElement.className);     // значення
атрибута class
console.log(divElement.children.length); // кількість
дочірніх елементів

// Методи елемента
divElement.setAttribute('data-status', 'active');
divElement.classList.add('highlighted');
divElement.appendChild(document.createElement('span'));

```

Text nodes (текстові вузли) містять фактичний текстовий вміст, який відображається на сторінці. Важливою особливістю текстових вузлів є те, що браузер створює окремий текстовий вузол для кожного фрагмента тексту, включаючи пробіли та символи переходу на новий рядок між HTML тегами. Це може здаватися дивним на перший погляд, але така поведінка дозволяє точно зберігати форматування документа.

```

// HTML: <p>Це <strong>важливий</strong> текст</p>
const paragraph = document.querySelector('p');

console.log(paragraph.childNodes.length); // 3 вузли
console.log(paragraph.childNodes[0].nodeType); // 3
(TEXT_NODE) - "Це "
console.log(paragraph.childNodes[1].nodeType); // 1
(ELEMENT_NODE) - <strong>
console.log(paragraph.childNodes[2].nodeType); // 3
(TEXT_NODE) - " текст"

// Отримання тільки текстового вмісту

```

```
console.log(paragraph.textContent); // "Це важливий
текст"
```

Розуміння поведінки текстових вузлів особливо важливе при програмному створенні та модифікації контенту, оскільки неврахування пробілів може призвести до неочікуваного форматування.

```
// Створення елемента з правильним текстовим вмістом
const container = document.createElement('div');
const textNode = document.createTextNode('Чистий текст
без HTML');
const emphasizedText = document.createElement('em');
emphasizedText.textContent = 'виділений текст';

container.appendChild(textNode);
container.appendChild(document.createTextNode(' з '));
container.appendChild(emphasizedText);
// Результат: "Чистий текст без HTML з виділений текст"
```

Comment nodes (вузли коментарів) представляють HTML коментарі, які зазвичай використовуються розробниками для документування коду або тимчасового відключення частин розмітки. Хоча коментарі не відображаються на сторінці, вони залишаються частиною DOM дерева і можуть бути корисними для програмного аналізу структури документа або зберігання метаданих.

```
// HTML: <!-- Це важливий коментар -->
const comment = document.createComment('Динамічно
створений коментар');
document.body.appendChild(comment);

// Пошук коментарів у документі
function findComments(element) {
    const comments = [];
```

```

    const walker = document.createTreeWalker(
        element,
        NodeFilter.SHOW_COMMENT,
        null,
        false
    );

    let node;
    while (node = walker.nextNode()) {
        comments.push(node.textContent);
    }
    return comments;
}

console.log(findComments(document.body)); // масив
текстів коментарів

```

**Document node** (вузол документа) є абсолютним кореневим вузлом DOM дерева, який представляє весь документ як цілісну структуру. Цей вузол надає глобальні методи для роботи з документом та служить точкою входу для всіх операцій пошуку та маніпулювання елементами. Через **document node** розробники отримують доступ до метаданих про документ, можуть створювати нові елементи та виконувати глобальні операції.

```

// Властивості документа
console.log(document.nodeType);           // 9
(DOCUMENT_NODE)
console.log(document.nodeName);           // "#document"
console.log(document.documentElement);    // <html>
елемент
console.log(document.body);               // <body>
елемент

```

```

console.log(document.head);           // <head>
елемент

// Метаінформація документа
console.log(document.title);          // заголовок
сторінки
console.log(document.URL);            // повна адреса
документа
console.log(document.domain);         // домен сайту
console.log(document.readyState);     // стан
завантаження: loading/interactive/complete

// Глобальні методи створення
const newElement = document.createElement('article');
const textNode = document.createTextNode('Текстовий
вміст');
const fragment = document.createDocumentFragment();

```

Різні типи вузлів DOM взаємодіють між собою, створюючи складну, але логічну структуру. Елементи можуть містити інші елементи, текстові вузли та коментарі. Текстові вузли завжди є листовими вузлами дерева - вони не можуть містити інших вузлів. Коментарі також є листовими вузлами, але на відміну від текстових вузлів, вони не впливають на візуальне відображення сторінки.

```

// Аналіз структури складного елемента
function analyzeElement(element) {
    console.log(`Елемент: ${element.tagName}`);
    console.log(`Всього дочірніх вузлів:
${element.childNodes.length}`);

    let elementCount = 0;
    let textCount = 0;

```



```

let commentCount = 0;

element.childNodes.forEach(node => {
    switch(node.nodeType) {
        case Node.ELEMENT_NODE:
            elementCount++;
            console.log(`  Елемент:
${node.tagName}`);
            break;
        case Node.TEXT_NODE:
            if (node.textContent.trim()) { //
ігноруємо пусті текстові вузли
                textCount++;
                console.log(`  Текст:
"${node.textContent.trim()}"`);
            }
            break;
        case Node.COMMENT_NODE:
            commentCount++;
            console.log(`  Коментар:
"${node.textContent}"`);
            break;
    }
});

console.log(`Підсумок: ${elementCount} елементів,
${textCount} текстових вузлів, ${commentCount}
коментарів`);
}

```

```
// Використання для аналізу структури
analyzeElement(document.body);
```

Розуміння цієї структури особливо важливе при програмному створенні контенту, оскільки невірне поводження з різними типами вузлів може призвести до неочікуваних результатів у відображенні або поведінці сторінки.

### **DOM як живе дерево об'єктів**

На відміну від статичного HTML коду, DOM є живою структурою, яка миттєво реагує на зміни. Коли JavaScript код змінює властивості DOM вузла, ці зміни одразу ж відображаються на екрані користувача без потреби в перезавантаженні сторінки або додаткових операціях оновлення.

```
// Демонстрація "живої" природи DOM
const liveElement = document.createElement('div');
liveElement.textContent = 'Початковий текст';
document.body.appendChild(liveElement);

// Зміни одразу відображаються на екрані
setTimeout(() => {
    liveElement.style.color = 'red';
    liveElement.textContent = 'Змінений текст';
}, 1000);

setTimeout(() => {
    liveElement.classList.add('animated');
    liveElement.style.transform = 'scale(1.2)';
}, 2000);
```

Ця живість DOM також означає, що зміни, внесені через інші способи (наприклад, через інструменти розробника браузера або інші скрипти), одразу стають доступними через JavaScript API. Це створює потужну екосистему, де різні частини додатка можуть взаємодіяти між собою через спільну DOM модель.

## Ієрархічна природа DOM та об'єктне наслідування

Кожен тип DOM вузла наслідує властивості та методи від базових інтерфейсів, створюючи ієрархію об'єктів. Базовий інтерфейс Node надає загальні властивості (такі як `nodeType`, `nodeName`, `parentNode`), які потім розширюються більш специфічними інтерфейсами як `Element`, `HTMLElement`, та конкретними елементами як `HTMLDivElement` або `HTMLInputElement`.

```
const inputElement =
document.querySelector('input[type="text"]');

// Перевірка ієрархії наслідування
console.log(inputElement instanceof Node);           //
true - базовий інтерфейс
console.log(inputElement instanceof Element);         //
true - загальний елемент
console.log(inputElement instanceof HTMLElement);     //
true - HTML елемент
console.log(inputElement instanceof HTMLInputElement);
// true - специфічний тип

// Доступ до загальних властивостей Node
console.log(inputElement.nodeType);                   // 1
console.log(inputElement.parentNode);                 // батьківський
елемент

// Доступ до властивостей Element
console.log(inputElement.tagName);                    // "INPUT"
console.log(inputElement.classList);                  // DOMTokenList

// Доступ до специфічних властивостей HTMLInputElement
console.log(inputElement.type);                        // "text"
```

```
console.log(inputElement.value);           // поточне
значення
console.log(inputElement.placeholder);    // placeholder
текст
```

### Особливості роботи з різними типами вузлів

Робота з текстовими вузлами вимагає особливої уваги, оскільки вони поведуться інакше ніж елементи. Текстові вузли не мають властивостей для стилізації або атрибутів, але вони мають власні методи для маніпулювання текстовим вмістом.

```
// HTML: <p>Початок <em>середина</em> кінець</p>
const paragraph = document.querySelector('p');
const firstTextNode = paragraph.firstChild;

console.log(firstTextNode.nodeType === Node.TEXT_NODE);
// true
console.log(firstTextNode.textContent); // "Початок "
```

// Зміна текстового вузла

```
firstTextNode.textContent = 'Новий початок ';
```

// Розділення текстового вузла

```
const splitNode = firstTextNode.splitText(5); //
розділити після 5-го символу
console.log(firstTextNode.textContent); // "Новий"
console.log(splitNode.textContent);      // " початок "
```

Робота з коментарями може бути корисною для створення системи шаблонів або збереження метаданих у документі:

```
// Використання коментарів як маркерів
function insertContentAfterComment(commentText,
newContent) {
```

```

const walker = document.createTreeWalker(
    document.body,
    NodeFilter.SHOW_COMMENT,
    node => node.textContent.includes(commentText)
? NodeFilter.FILTER_ACCEPT : NodeFilter.FILTER_REJECT
);

const commentNode = walker.nextNode();
if (commentNode) {
    const newElement =
document.createElement('div');
    newElement.innerHTML = newContent;
    commentNode.parentNode.insertBefore(newElement,
commentNode.nextSibling);
}
}

// HTML: <!-- insert-dynamic-content -->
insertContentAfterComment('insert-dynamic-content',
'<p>Динамічно вставлений контент</p>');

```

Розуміння цих різних типів вузлів та їх взаємодії є фундаментом для створення складних, інтерактивних веб-додатків, які ефективно маніпулюють структурою та вмістом документа в відповідь на дії користувача або зміни в даних додатка.

### 3.2.3 Browser Object Model (BOM)

Browser Object Model є другим фундаментальним API браузерного середовища, який часто залишається в тіні DOM, але насправді є не менш важливим для створення повноцінних веб-додатків. Якщо DOM дозволяє нам працювати з вмістом веб-сторінки, то BOM надає можливості для взаємодії з

самим браузером як програмним середовищем - його вікнами, вкладками, історією навігації, налаштуваннями та системною інформацією.

Принципова відмінність між DOM та BOM полягає в об'єкті їх впливу: DOM керує структурою та вмістом документа (тобто тим, що знаходиться всередині HTML сторінки), тоді як BOM контролює контекст, в якому цей документ існує - вікно браузера, його розташування на екрані, історію переходів між сторінками, інформацію про пристрій користувача тощо.

```
// DOM - робота з вмістом сторінки
const pageTitle = document.querySelector('h1');
pageTitle.textContent = 'Новий заголовок сторінки';

// BOM - робота з браузером як додатком
const windowWidth = window.innerWidth;
const currentUrl = window.location.href;
const browserInfo = window.navigator.userAgent;

console.log(`Сторінка "${pageTitle.textContent}"
відкрита в браузері шириною ${windowWidth}px`);
```

BOM є менш стандартизованим ніж DOM, що історично призводило до різної поведінки в різних браузерах. Однак сучасні браузери значною мірою уніфікували реалізацію основних BOM API, що дозволяє розробникам впевнено використовувати ці можливості в кросбраузерних додатках.

### **Об'єкт window - центральний хаб браузерного середовища**

Об'єкт window виконує роль центрального координатора всієї браузерної екосистеми і є унікальним тим, що він одночасно служить глобальним об'єктом JavaScript та представляє вікно браузера як програмну сутність. Кожна вкладка браузера має свій власний об'єкт window, що забезпечує ізоляцію між різними веб-сторінками.

Розміри та позиціонування вікна надають розробникам можливість створювати адаптивні інтерфейси, які реагують на зміни розмірів вікна та можуть оптимально використовувати доступний простір екрана:

```
// Детальна інформація про розміри вікна
console.log('Внутрішня ширина (без прокруток):',
window.innerWidth);
console.log('Внутрішня висота (без панелей):',
window.innerHeight);
console.log('Зовнішня ширина (з рамками):',
window.outerWidth);
console.log('Зовнішня висота (з панелями):',
window.outerHeight);

// Позиція вікна на екрані
console.log('Позиція X на екрані:', window.screenX);
console.log('Позиція Y на екрані:', window.screenY);

// Адаптивний інтерфейс на основі розмірів
function adaptInterface() {
    const isMobile = window.innerWidth < 768;
    const navigation =
document.querySelector('.navigation');

    if (isMobile) {
        navigation.classList.add('mobile-layout');
        console.log('Перемикаємося на мобільний
інтерфейс');
    } else {
        navigation.classList.remove('mobile-layout');
```

```

        console.log('Використовуємо десктопний
інтерфейс');
    }
}

// Відстеження змін розміру вікна
window.addEventListener('resize', adaptInterface);

```

### **Об'єкт location - навігаційний центр**

Об'єкт location є надзвичайно потужним інструментом для роботи з URL адресами та навігацією, який дозволяє не тільки отримувати інформацію про поточну сторінку, але й програмно керувати переходами між сторінками та модифікувати адресний рядок браузера.

```

// Детальний розбір URL адреси
// Для URL:
https://example.com:8080/products/shoes?color=red&size=
42#reviews
console.log(location.protocol);    // "https:"
console.log(location.hostname);    // "example.com"
console.log(location.port);        // "8080"
console.log(location.pathname);    // "/products/shoes"
console.log(location.search);      //
"?color=red&size=42"
console.log(location.hash);        // "#reviews"
console.log(location.origin);      //
"https://example.com:8080"

// Парсинг параметрів URL
const urlParams = new URLSearchParams(location.search);
console.log(urlParams.get('color')); // "red"
console.log(urlParams.get('size'));  // "42"

```



```
// Модифікація URL без перезавантаження сторінки
function updateUrlParameter(key, value) {
    const params = new
URLSearchParams(location.search);
    params.set(key, value);
    const newUrl =
`${location.pathname}?${params.toString()}`;
    window.history.pushState({}, '', newUrl);
}

updateUrlParameter('page', '2'); // URL зміниться на
?color=red&size=42&page=2
```

Програмна навігація через об'єкт `location` дозволяє створювати складні сценарії переходів між сторінками:

```
// Різні способи навігації
function navigateToPage(url, forceReload = false) {
    if (forceReload) {
        location.assign(url); // перехід з додаванням в
історію
    } else {
        location.replace(url); // перехід без додавання
в історію
    }
}

// Умовна навігація з перевітками
function smartNavigation(targetUrl) {
    // Перевірка, чи це внутрішнє посилання
    const currentDomain = location.hostname;
```

```

const targetDomain = new URL(targetUrl).hostname;

if (currentDomain === targetDomain) {
    // Внутрішня навігація
    location.href = targetUrl;
} else {
    // Зовнішнє посилання - відкриваємо в новій
    вкладці
    window.open(targetUrl, '_blank',
'noopener,noreferrer');
}
}

```

### Об'єкт history - керування браузерною історією

Сучасний об'єкт history надає розширені можливості для створення односторінкових додатків (SPA), де навігація між "сторінками" відбувається без фактичного перезавантаження браузера. Це досягається через History API, який дозволяє програмно додавати записи в історію браузера та реагувати на навігаційні події.

```

// Сучасна навігація без перезавантаження
function navigateToSection(sectionName, data) {
    // Створення нового запису в історії
    const state = { section: sectionName, timestamp:
Date.now(), ...data };
    const url = `/app/${sectionName}`;

    history.pushState(state, '', url);

    // Оновлення інтерфейсу відповідно до нової
    "сторінки"
    updateUserInterface(sectionName, data);
}

```

```

}

// Обробка навігації назад/вперед
window.addEventListener('popstate', function(event) {
    if (event.state) {
        console.log('Повертаємося до секції:',
event.state.section);
        updateUserInterface(event.state.section,
event.state);
    } else {
        // Початковий стан сторінки
        updateUserInterface('home', {});
    }
});

function updateUserInterface(section, data) {
    // Приховуємо всі секції
    document.querySelectorAll('.section').forEach(s =>
s.style.display = 'none');

    // Показуємо активну секцію
    const activeSection =
document.querySelector(`#section-${section}`);
    if (activeSection) {
        activeSection.style.display = 'block';
        document.title = `Додаток - ${section}`;
    }
}

```

Розширені можливості history API включають роботу з метаданими навігації:

```
// Збереження складного стану додатка
class AppStateManager {
  constructor() {
    this.currentState = this.getInitialState();
    window.addEventListener('popstate',
this.handlePopState.bind(this));
  }

  saveState(newState, url, title) {
    this.currentState = { ...this.currentState,
...newState };
    history.pushState(this.currentState, title,
url);
    document.title = title;
  }

  handlePopState(event) {
    if (event.state) {
      this.currentState = event.state;
      this.restoreApplicationState(event.state);
    }
  }

  getInitialState() {
    return {
      user: null,
      activeTab: 'home',
      scrollPosition: 0,
      filters: {}
    };
  }
}
```

```

    }

    restoreApplicationState(state) {
        // Відновлення інтерфейсу згідно збереженого
        стану
        if (state.scrollTop) {
            window.scrollTo(0, state.scrollTop);
        }

        if (state.activeTab) {
            this.activateTab(state.activeTab);
        }
    }
}

```

### **Об'єкт navigator - детектор браузера та пристрою**

Об'єкт navigator є цінним джерелом інформації про браузер користувача, операційну систему та доступні можливості пристрою. Ця інформація дозволяє створювати адаптивні додатки, які оптимізують свою поведінку залежно від платформи та можливостей браузера.

```

// Детальна інформація про браузер та систему
console.log('Браузер:', navigator.userAgent);
console.log('Платформа:', navigator.platform);
console.log('Мова:', navigator.language);
console.log('Мови користувача:', navigator.languages);
console.log('Онлайн статус:', navigator.onLine);

// Перевірка підтримки різних технологій
const capabilities = {
    geolocation: 'geolocation' in navigator,
    webgl: !!window.WebGLRenderingContext,

```

```

    webWorkers: typeof Worker !== 'undefined',
    localStorage: typeof Storage !== 'undefined',
    notifications: 'Notification' in window,
    serviceWorker: 'serviceWorker' in navigator
};

console.log('Можливості браузера:', capabilities);

// Адаптація функціональності під можливості браузера
function setupAdvancedFeatures() {
    if (capabilities.geolocation) {
        setupLocationBasedFeatures();
    }

    if (capabilities.notifications) {
        setupPushNotifications();
    }

    if (capabilities.serviceWorker) {
        registerServiceWorker();
    }
}

```

**Робота з мережевим статусом особливо важлива для створення надійних веб-додатків:**

```

// Моніторинг мережевого з'єднання
class NetworkManager {
    constructor() {
        this.isOnline = navigator.onLine;
        this.setupEventListeners();
        this.updateUI();
    }
}

```

```

    }

    setupEventListeners() {
        window.addEventListener('online', () => {
            this.isOnline = true;
            console.log('З\'єднання відновлено');
            this.syncPendingData();
            this.updateUI();
        });

        window.addEventListener('offline', () => {
            this.isOnline = false;
            console.log('З\'єднання втрачено');
            this.enableOfflineMode();
            this.updateUI();
        });
    }

    updateUI() {
        const statusIndicator =
document.querySelector('.network-status');
        if (statusIndicator) {
            statusIndicator.textContent = this.isOnline
? 'Онлайн' : 'Офлайн';
            statusIndicator.className = `network-status
${this.isOnline ? 'online' : 'offline'}`;
        }
    }

    syncPendingData() {

```

```

        // Синхронізація даних після відновлення
з'єднання
        const pendingData = this.getPendingData();
        pendingData.forEach(data =>
this.sendToServer(data));
    }
}

```

### Розширені можливості об'єкта screen

Об'єкт `screen`, доступний через `window.screen`, надає детальну інформацію про екран користувача, що особливо корисно для створення повноекранних додатків або адаптації інтерфейсу під характеристики дисплея:

```

// Інформація про екран
console.log('Роздільність екрана:', screen.width, 'x',
screen.height);
console.log('Доступна область:', screen.availWidth,
'x', screen.availHeight);
console.log('Глибина кольору:', screen.colorDepth,
'біт');
console.log('Орієнтація екрана:',
screen.orientation?.type || 'невідома');

// Адаптація під роздільність екрана
function optimizeForScreen() {
    const pixelRatio = window.devicePixelRatio;
    const isHighDPI = pixelRatio > 1.5;

    if (isHighDPI) {
        // Завантаження зображень високої роздільності
        document.querySelectorAll('img[data-
 hires]').forEach(img => {

```



```

        img.src = img.dataset.hires;
    });

    console.log(`Оптимізація для Retina дисплея
(${pixelRatio}x)`);
}

// Перевірка на мобільний пристрій за розміром
екрана
const isMobileScreen = screen.width < 768;
if (isMobileScreen) {
    document.body.classList.add('mobile-device');
}
}

```

### Таймери та асинхронні операції

BOM включає важливі функції для роботи з часом та асинхронними операціями, які є основою для створення інтерактивних та анімованих інтерфейсів:

```

// Різні типи таймерів та їх використання
class TimerManager {
    constructor() {
        this.timers = new Map();
        this.intervals = new Map();
    }

    // Відкладене виконання з можливістю скасування
    delayedAction(name, callback, delay) {
        // Скасовуємо попередній таймер з таким ім'ям
        if (this.timers.has(name)) {
            clearTimeout(this.timers.get(name));
        }
    }
}

```

```

        const timerId = setTimeout(() => {
            callback();
            this.timers.delete(name);
        }, delay);

        this.timers.set(name, timerId);
    }

    // Періодичне виконання
    startPeriodicAction(name, callback, interval) {
        this.stopPeriodicAction(name); // зупиняємо
попередній

        const intervalId = setInterval(callback,
interval);
        this.intervals.set(name, intervalId);
    }

    stopPeriodicAction(name) {
        if (this.intervals.has(name)) {
            clearInterval(this.intervals.get(name));
            this.intervals.delete(name);
        }
    }

    // Очищення всіх таймерів
    cleanup() {
        this.timers.forEach(timerId =>
clearTimeout(timerId));

```

```

        this.intervals.forEach(intervalId =>
clearInterval(intervalId));
        this.timers.clear();
        this.intervals.clear();
    }
}

// Практичне використання
const timerManager = new TimerManager();

// Автозбереження форми через 2 секунди після останньої
зміни
document.querySelector('#form').addEventListener('input
', () => {
    timerManager.delayedAction('autosave', () => {
        console.log('Автоматичне збереження форми');
        saveFormData();
    }, 2000);
});

```

### **Діалогові вікна та взаємодія з користувачем**

ВОМ надає базові методи для взаємодії з користувачем через системні діалогові вікна, хоча сучасні веб-додатки часто замінюють їх власними, більш гнучкими рішеннями:

```

// Традиційні діалогові вікна
function showUserDialog() {
    // Просте повідомлення
    alert('Операція завершена успішно!');

    // Підтвердження дії

```

```

    const userConfirmed = confirm('Ви впевнені, що
хочете видалити цей елемент?');

    if (userConfirmed) {
        console.log('Користувач підтвердив видалення');
    }

    // Запит введення даних
    const userName = prompt('Введіть ваше ім\'я:',
'Гість');
    if (userName !== null) { // користувач не натиснув
"Скасувати"
        console.log('Ім\'я користувача:', userName);
    }
}

// Сучасна альтернатива з кращим UX
class CustomDialog {
    static async confirm(message, options = {}) {
        return new Promise((resolve) => {
            const dialog =
document.createElement('div');
            dialog.className = 'custom-dialog-overlay';
            dialog.innerHTML = `
                <div class="dialog-content">
                    <p>${message}</p>
                    <button class="btn-
confirm">Підтвердити</button>
                    <button class="btn-
cancel">Скасувати</button>
                </div>

```

```

        `;

        document.body.appendChild(dialog);

        dialog.querySelector('.btn-
confirm').onclick = () => {
            document.body.removeChild(dialog);
            resolve(true);
        };

        dialog.querySelector('.btn-cancel').onclick
= () => {
            document.body.removeChild(dialog);
            resolve(false);
        };
    });
}
}

// Використання сучасного діалогу
CustomDialog.confirm('Видалити цей
елемент?').then(confirmed => {
    if (confirmed) {
        console.log('Користувач підтвердив дію');
    }
});

```

### **Інтеграція BOM з сучасними веб-технологіями**

Сучасні браузерери розширюють традиційні BOM можливості новими API, які дозволяють створювати більш потужні та функціональні веб-додатки:

```
// Використання сучасних BOM API
```

```
class ModernBrowserFeatures {
    static async requestFullscreen(element =
document.documentElement) {
        try {
            await element.requestFullscreen();
            console.log('Перехід в повноекранний
режим');
        } catch (error) {
            console.log('Повноекранний режим не
підтримується або заблокований');
        }
    }

    static async shareContent(data) {
        if (navigator.share) {
            try {
                await navigator.share({
                    title: data.title,
                    text: data.description,
                    url: data.url
                });
                console.log('Контент успішно
поділений');
            } catch (error) {
                console.log('Поділитися скасовано
користувачем');
            }
        } else {
            // Fallback для браузерів без Web Share API
            this.fallbackShare(data);
        }
    }
}
```

```

    }
}

static fallbackShare(data) {
    const shareUrl =
`https://twitter.com/intent/tweet?text=${encodeURIComponent(
data.text)}&url=${encodeURIComponent(data.url)}`;
    window.open(shareUrl, '_blank',
'width=550,height=420');
}

static detectDeviceCapabilities() {
    return {
        touchScreen: 'ontouchstart' in window,
        deviceMotion: 'DeviceMotionEvent' in
window,
        vibration: 'vibrate' in navigator,
        camera: 'mediaDevices' in navigator &&
'getUserMedia' in navigator.mediaDevices,
        bluetooth: 'bluetooth' in navigator,
        nfc: 'nfc' in navigator
    };
}
}

// Практичне використання
const capabilities =
ModernBrowserFeatures.detectDeviceCapabilities();
console.log('Можливості пристрою:', capabilities);

```

```

if (capabilities.vibration) {
    // Вібрація при важливих подіях
    navigator.vibrate([100, 50, 100]); // короткий-
пауза-короткий
}

```

ВОМ продовжує еволюціонувати з появою нових веб-стандартів та API, надаючи розробникам все більше можливостей для створення насичених, інтерактивних веб-додатків, які можуть конкурувати з нативними програмами за функціональністю та користувацьким досвідом.

### 3.2.4 DOM дерево

DOM (Document Object Model) дерево являє собою ієрархічну структуру даних, яка точно відображає організацію HTML елементів у документі. Кожен HTML тег, текстовий вміст, атрибут та навіть коментарі стають окремими вузлами цього дерева, створюючи складну, але логічно організовану структуру.

Якщо уявити HTML документ як генеалогічне дерево родини, то кореневим елементом завжди буде об'єкт `document`, від якого "походять" усі інші елементи. Елемент `<html>` стає першим нащадком документа, а вже від нього розгалужуються `<head>` та `<body>`, які між собою є "братами" або сусідніми елементами.

Розглянемо докладніше наш приклад HTML документа та його DOM представлення:

```

<!DOCTYPE html>
<html>
<head>
    <title>Заголовок сторінки</title>
</head>
<body>
    <h1>Головний заголовок</h1>
    <p>Перший абзац тексту</p>

```



```

    <div>
        <span>Вкладений текст</span>
    </div>
</body>
</html>

```

У DOM дереві цей документ матиме наступну ієрархію:

- На найвищому рівні знаходиться document - це кореневий об'єкт
- Елемент <html> стає дочірнім вузлом document
- <head> та <body> є дочірніми вузлами <html> та одночасно сусідніми вузлами один відносно одного
- Усередині <head> знаходиться <title>, а текст "Заголовок сторінки" стає текстовим вузлом всередині title
- В <body> послідовно розташовані <h1>, <p> та <div>, які також є сусідніми елементами
- Кожен текстовий вміст, наприклад "Головний заголовок" або "Перший абзац тексту", створює окремий текстовий вузол

### Властивості для навігації по DOM дереву

JavaScript надає нам потужний набір властивостей для переміщення по DOM дереву, які можна умовно поділити на кілька категорій: доступ до батьківських елементів, роботу з дочірніми вузлами та навігацію між сусідніми елементами.

Властивість parentNode дозволяє отримати прямого батька будь-якого елемента в DOM дереві. Наприклад, якщо ми маємо посилання на елемент <body>, то його батьківським елементом буде <html>:

```

const bodyElement = document.body;
console.log(bodyElement.parentNode); // виведе <html>
елемент

```

Також існує властивість parentElement, яка працює аналогічно до parentNode, але гарантовано повертає саме Element (а не будь-який тип вузла).

Для доступу до дочірніх елементів DOM надає декілька різних властивостей, кожна з яких має свої особливості використання.

Властивість `childNodes` повертає `NodeList`, який містить усі дочірні вузли, включаючи текстові вузли, коментарі та пробільні символи. Це означає, що між HTML елементами можуть з'явитися додаткові текстові вузли, які представляють пробіли та переноси рядків у вихідному HTML коді:

```
const bodyElement = document.body;
console.log(bodyElement.childNodes); // NodeList з
усіма вузлами, включаючи текстові
```

Властивість `children` є більш зручною для практичного використання, оскільки повертає `HTMLCollection`, який містить тільки елементи (теги), ігноруючи текстові вузли та коментарі:

```
console.log(bodyElement.children); // HTMLCollection
тільки з HTML елементами
```

Для швидкого доступу до першого та останнього дочірнього елемента існують спеціальні властивості:

- `firstChild` та `lastChild` - включають усі типи вузлів
- `firstElementChild` та `lastElementChild` - тільки HTML елементи

Сусідні елементи - це вузли, які знаходяться на одному рівні ієрархії та мають спільного батька. Для переміщення між ними використовуються наступні властивості:

```
const headerElement = document.querySelector('h1');

// Отримання наступного вузла (може бути текстовим)
console.log(headerElement.nextSibling);

// Отримання попереднього вузла
console.log(headerElement.previousSibling);
```

```
// Отримання наступного HTML елемента (пропускає
текстові вузли)
console.log(headerElement.nextElementSibling); //
поверне <p> елемент

// Отримання попереднього HTML елемента
console.log(headerElement.previousElementSibling);
```

Важливо розуміти різницю між властивостями, які включають усі типи вузлів (`nextSibling`, `previousSibling`), та тими, які працюють тільки з HTML елементами (`nextElementSibling`, `previousElementSibling`). Перші можуть повернути текстовий вузол, який представляє пробіли між тегами, тоді як другі завжди повертають саме HTML елементи або `null`, якщо сусіднього елемента немає.

### Практичне застосування навігації по DOM

Розуміння структури DOM дерева та вміння ефективно по ньому навігувати є критично важливим для динамічної роботи з веб-сторінками. Наприклад, якщо нам потрібно знайти всі абзаци, які йдуть після певного заголовка, ми можемо використовувати комбінацію цих властивостей:

```
const header = document.querySelector('h1');
let currentElement = header.nextElementSibling;

while (currentElement && currentElement.tagName !==
'H1') {
    if (currentElement.tagName === 'P') {
        console.log('Знайдено абзац:',
currentElement.textContent);
    }
    currentElement = currentElement.nextElementSibling;
}
```

Така навігація дозволяє створювати гнучкі та потужні скрипти для обробки контенту веб-сторінок без необхідності знати точну структуру документа заздалегідь.

### 3.2.5 Навігація по DOM

Уявіть собі, що DOM (Document Object Model) - це розгалужене дерево, де кожен елемент HTML-сторінки є вузлом цього дерева, а відносини між елементами утворюють гілки, що з'єднують батьківські та дочірні вузли. Навігація по DOM дозволяє нам програмно переміщатися між цими вузлами, знаходити потрібні елементи та працювати з ними динамічно.

DOM-дерево складається з різних типів вузлів, але найчастіше ми працюємо з двома основними типами: вузлами елементів (element nodes) та текстовими вузлами (text nodes). Елементи - це HTML-теги на кшталт `<div>`, `<p>`, `<span>`, а текстові вузли містять власне текстовий вміст між тегами. Розуміння цієї різниці критично важливе для ефективної навігації.

#### Навігація серед усіх типів вузлів

Коли ми використовуємо властивості на кшталт `childNodes`, `nextSibling` або `previousSibling`, браузер повертає нам всі вузли, включаючи текстові та коментарі. Це означає, що навіть пробіли та переноси рядків між HTML-тегами створюють текстові вузли.

```
// Припустимо, у нас є така HTML-структура:  
// <div>  
//   <p>Перший параграф</p>  
//   <p>Другий параграф</p>  
// </div>  
  
const container = document.querySelector('div');  
const allNodes = container.childNodes;
```

```

console.log(allNodes.length); // Може бути 5:
текст(пробіл), р, текст(пробіл), р, текст(пробіл)

// Щоб відфільтрувати тільки елементи, перевіряємо
nodeType
for (let node of allNodes) {
    if (node.nodeType === Node.ELEMENT_NODE) {
        console.log('Знайшли елемент:', node.tagName);
    } else if (node.nodeType === Node.TEXT_NODE &&
node.textContent.trim()) {
        console.log('Знайшли текст:',
node.textContent.trim());
    }
}

```

### Навігація тільки серед елементів

Для більшості практичних завдань зручніше використовувати властивості, що працюють виключно з елементами, ігноруючи текстові вузли та коментарі. Це робить код чистішим та передбачуванішим.

```

const mainContainer =
document.getElementById('content');

// Отримуємо тільки дочірні елементи (без текстових
вузлів)
const childElements = mainContainer.children;

// Знаходимо перший дочірній елемент
const firstElement = mainContainer.firstElementChild;
console.log('Перший елемент:', firstElement?.tagName);

// Знаходимо останній дочірній елемент

```

```

const lastElement = mainContainer.lastElementChild;
console.log('Останній елемент:', lastElement?.tagName);

// Переміщення між сусідніми елементами
const currentElement =
document.querySelector('.active');
const nextElement = currentElement.nextElementSibling;
const previousElement =
currentElement.previousElementSibling;

if (nextElement) {
    console.log('Наступний елемент:',
nextElement.className);
}

```

### Рекурсивний обхід DOM-дерева

Рекурсивна навігація особливо корисна, коли потрібно обробити всі елементи в дереві незалежно від їхньої глибини вкладеності. Такий підхід дозволяє створювати універсальні функції для аналізу або модифікації вмісту сторінки.

```

function findAllElementsWithClass(startElement,
className) {
    const foundElements = [];

    function searchRecursively(element) {
        // Перевіряємо поточний елемент
        if (element.classList &&
element.classList.contains(className)) {
            foundElements.push(element);
        }
    }
}

```

```

        // Рекурсивно обходимо всіх дочірніх елементів
        for (let child of element.children) {
            searchRecursively(child);
        }
    }

    searchRecursively(startElement);
    return foundElements;
}

// Практичне використання
const warningElements =
findAllElementsWithClass(document.body, 'warning');
console.log(`Знайдено ${warningElements.length}
попереджень на сторінці`);

```

### **Практичні застосування навігації**

Навігація по DOM часто використовується для створення інтерактивних інтерфейсів, наприклад, для реалізації табів, акордеонів або динамічного меню.

```

// Приклад: створення простої навігації по табях
function initializeTabs() {
    const tabContainer =
document.querySelector('.tabs');
    const tabButtons =
tabContainer.querySelectorAll('.tab-button');

    tabButtons.forEach((button, index) => {
        button.addEventListener('click', function() {
            // Знаходимо поточний активний таб

```

```

        const currentActive =
tabContainer.querySelector('.tab-button.active');
        if (currentActive) {

currentActive.classList.remove('active');
        }

        // Активуємо натиснутий таб
        this.classList.add('active');

        // Знаходимо відповідний контент через
навігацію

        const contentContainer =
tabContainer.nextElementSibling;
        const allPanels =
contentContainer.children;

        // Ховаємо всі панелі та показуємо потрібну
        Array.from(allPanels).forEach((panel,
panelIndex) => {
            panel.style.display = panelIndex ===
index ? 'block' : 'none';
        });
    });
}

```

### Оптимізація навігації

При роботі з великими DOM-структурами важливо пам'ятати про продуктивність та кешувати результати пошуку, особливо якщо ви часто звертаєтесь до одних і тих же елементів.



```
// Неєфективно: повторні запити до DOM
function updateUserInterface() {
    document.querySelector('#status').textContent =
'Завантаження...';
    document.querySelector('#progress').style.width =
'50%';
    document.querySelector('#status').style.color =
'blue';
}

// Ефективно: кешування елементів
const uiElements = {
    status: document.querySelector('#status'),
    progress: document.querySelector('#progress')
};

function updateUserInterface() {
    uiElements.status.textContent = 'Завантаження...';
    uiElements.status.style.color = 'blue';
    uiElements.progress.style.width = '50%';
}
```

Розуміння принципів навігації по DOM дозволяє створювати більш інтерактивні та динамічні веб-додатки, ефективно маніпулювати вмістом сторінки та реагувати на дії користувача в режимі реального часу.

### 3.2.6 Пошук елементів: `getElement*` та `querySelector*`

#### Традиційні методи пошуку: сімейство `getElement*`

Почнемо з традиційних методів, які існують у JavaScript вже дуже давно і досі залишаються надзвичайно корисними завдяки своїй швидкості та простоті використання.

Метод `getElementById` є найшвидшим способом знайти елемент на сторінці, оскільки він шукає за унікальним ідентифікатором, який браузер може знайти дуже ефективно. Уявіть собі, що ваша веб-сторінка - це велика бібліотека, а ідентифікатори - це номери на полицях: ви можете миттєво знайти потрібну книгу, знаючи її точний номер.

```
// Припустимо, у нас є такий HTML:  
// <header id="main-header">Головна сторінка</header>  
  
const header = document.getElementById('main-header');  
if (header) {  
    header.style.color = 'blue';  
    header.style.fontSize = '24px';  
    console.log('Знайшли заголовок:',  
header.textContent);  
}
```

Зверніть увагу на перевірку `if (header)` - це хороша практика, оскільки якщо елемент з таким ідентифікатором не існує, метод поверне `null`, і спроба працювати з неіснуючим елементом призведе до помилки.

Метод `getElementsByName` - робота з колекціями однотипних елементів, цей метод повертає так звану "живу колекцію" всіх елементів з певним ім'ям тега. "Жива" означає, що якщо ви додасте або видалите елементи зі сторінки після того, як отримали колекцію, вона автоматично оновиться.

```
// Знаходимо всі параграфи на сторінці  
const allParagraphs =  
document.getElementsByTagName('p');  
console.log('Кількість параграфів:',  
allParagraphs.length);  
  
// Демонструємо "живість" колекції  
const initialCount = allParagraphs.length;
```

```

console.log('Початкова кількість:', initialCount);

// Створюємо новий параграф і додаємо його до сторінки
const newParagraph = document.createElement('p');
newParagraph.textContent = 'Це новий параграф';
document.body.appendChild(newParagraph);

// Колекція автоматично оновилася!
console.log('Нова кількість:', allParagraphs.length);
// збільшилася на 1

// Можемо працювати з кожним елементом колекції
for (let i = 0; i < allParagraphs.length; i++) {
    allParagraphs[i].style.marginBottom = '10px';
}

```

Метод `getElementsByClassName` - пошук за CSS-класами, цей метод також повертає живу колекцію, але шукає елементи за їхніми CSS-класами. Це дуже зручно, коли ви хочете застосувати однакові зміни до групи елементів з однаковим стилем.

```

// HTML: <div class="highlight">Важливий текст</div>
//       <span class="highlight">Ще важливий
текст</span>

const highlightedElements =
document.getElementsByClassName('highlight');

// Використовуємо сучасний цикл for...of для ітерації
for (let element of highlightedElements) {
    element.style.backgroundColor = 'yellow';
    element.style.fontWeight = 'bold';
}

```

```

    element.style.padding = '5px';
}

// Альтернативний спосіб через Array.from для
використання методів масивів
Array.from(highlightedElements).forEach(element => {
    element.addEventListener('click', () => {
        alert('Клікнули на виділений елемент!');
    });
});

```

### Сучасні методи: могутнє сімейство `querySelector*`

Тепер перейдемо до більш сучасних і гнучких методів, які з'явилися пізніше, але надають значно більше можливостей завдяки використанню CSS-селекторів.

Метод `querySelector` - знаходження першого відповідного елемента. Цей метод приймає будь-який валідний CSS-селектор і повертає перший елемент, який йому відповідає. Це як мати універсальний ключ, який може відкрити будь-які двері в DOM-дереві.

```

// Простий пошук за ідентифікатором
const header = document.querySelector('#main-header');

// Пошук за класом
const firstButton = document.querySelector('.btn');

// Пошук за атрибутами - знаходимо посилання в
навігації, що містить "contact"
const contactLink = document.querySelector('nav
a[href*="contact"]');

// Використання псевдо-селекторів

```

```

const lastListItem = document.querySelector('ul
li:last-child');
const firstEvenRow = document.querySelector('table
tr:nth-child(even)');

// Комбіновані селектори
const importantButton =
document.querySelector('button.btn.important');
const formInput = document.querySelector('form#login-
form input[type="password"]');

if (contactLink) {
    contactLink.style.textDecoration = 'none';
    contactLink.style.color = 'green';
}

```

Метод `querySelectorAll` - отримання всіх відповідних елементів. На відміну від методів `getElementsBy*`, `querySelectorAll` повертає статичну `NodeList` - вона не оновлюється автоматично при змінах DOM, але зате підтримує сучасні методи масивів.

```

// Знаходимо всі посилання на сторінці
const allLinks = document.querySelectorAll('a');
console.log(`Знайдено ${allLinks.length} посилань`);

// Складні селектори для знаходження різних типів полів
вводу
const textInputs =
document.querySelectorAll('input[type="text"],
input[type="email"], input[type="password"]');

// Знаходимо всі елементи з певними класами

```

```

const cards = document.querySelectorAll('.card,
.product-item, .news-article');

// Сучасний спосіб роботи з колекцією через forEach
allLinks.forEach((link, index) => {
    // Додаємо обробник події для кожного посилання
    link.addEventListener('click', (event) => {
        console.log(`Клікнули на посилання №${index +
1}: ${link.href}`);
    });

    // Додаємо візуальні ефекти
    link.addEventListener('mouseenter', () => {
        link.style.backgroundColor = '#f0f0f0';
    });

    link.addEventListener('mouseleave', () => {
        link.style.backgroundColor = '';
    });
});

// Фільтрація результатів
const externalLinks = Array.from(allLinks).filter(link
=>
    link.href.startsWith('http') &&
    !link.href.includes(window.location.hostname)
);

console.log(`Знайдено ${externalLinks.length} зовнішніх
посилань`);

```

### 3.2.7 Властивості вузлів: тип, тег та вміст

Кожен елемент у DOM дереві має фундаментальну характеристику, яка визначає його природу та можливості - це тип вузла, який зберігається у властивості `nodeType`. Розуміння різних типів вузлів є критично важливим для ефективної роботи з DOM, оскільки кожен тип має свої унікальні властивості, методи та обмеження, які впливають на те, як ми можемо з ними взаємодіяти програмно.

Браузер автоматично присвоює кожному вузлу відповідний тип під час парсингу HTML документа, і ця інформація залишається незмінною протягом життєвого циклу вузла. Знання типів вузлів дозволяє писати більш надійний код, який коректно обробляє різні ситуації та уникає помилок при спробі застосувати неприйнятні операції до певних типів вузлів.

```
// Практична функція для визначення та обробки різних
типів вузлів
function processNode(node) {
  switch (node.nodeType) {
    case Node.ELEMENT_NODE:
      console.log(`Знайдено елемент:
${node.tagName.toLowerCase()}`);
      if (node.id) console.log(`  ID:
${node.id}`);
      if (node.className) console.log(`  Класи:
${node.className}`);
      return 'element';

    case Node.TEXT_NODE:
      const text = node.textContent.trim();
      if (text) {
```

```

        console.log(`Знайдено текст:
"${text}"`);

        return 'text';
    }

    return 'whitespace';

    case Node.COMMENT_NODE:
        console.log(`Знайдено коментар:
"${node.textContent}"`);
        return 'comment';

    default:
        console.log(`Невідомий тип вузла:
${node.nodeType}`);
        return 'unknown';
    }
}

// Аналіз всіх дочірніх вузлів елемента
const container = document.querySelector('.container');
Array.from(container.childNodes).forEach(processNode);

```

Element nodes (вузли елементів) представляють HTML теги і є найбільш функціональними та універсальними вузлами в DOM дереві. Ці вузли мають найширший набір властивостей та методів, включаючи можливості стилізації, керування атрибутами, обробки подій та вкладення інших вузлів. Елементи можуть містити інші елементи, текстові вузли, коментарі, створюючи ієрархічну структуру документа.

```

const divElement = document.createElement('div');

// Елементи мають розширені можливості

```



```

divElement.id = 'dynamic-container';
divElement.className = 'content-block highlighted';
divElement.style.padding = '20px';
divElement.setAttribute('data-component', 'user-card');

// Елементи можуть містити інші вузли
const heading = document.createElement('h2');
heading.textContent = 'Заголовок блоку';
divElement.appendChild(heading);

const text = document.createTextNode('Основний текст блоку');
divElement.appendChild(text);

console.log(`Створено елемент ${divElement.tagName} з
${divElement.childNodes.length} дочірніми вузлами`);

```

Text nodes (текстові вузли) містять фактичний текстовий вміст, який бачить користувач на веб-сторінці. Важливою особливістю текстових вузлів є те, що браузер створює окремий текстовий вузол для кожного безперервного фрагмента тексту, включаючи пробіли, табуляції та символи переходу на новий рядок між HTML тегами. Ця поведінка може здаватися дивною початківцям, але вона забезпечує точне збереження форматування та пробілів у документі.

```

// HTML розмітка:
// <div>
//     Перший рядок
//     <span>вкладений елемент</span>
//     другий рядок
// </div>

const div = document.querySelector('div');

```

```

console.log(`Кількість дочірніх вузлів:
${div.childNodes.length}`); // 5 вузлів

div.childNodes.forEach((node, index) => {
    if (node.nodeType === Node.TEXT_NODE) {
        const content = node.textContent;
        console.log(`Текстовий вузол ${index}:
"${content}"`);
        // Покаже пробіли та переходи рядків
    }
});

// Очищення від пустих текстових вузлів
function removeEmptyTextNodes(element) {
    Array.from(element.childNodes).forEach(node => {
        if (node.nodeType === Node.TEXT_NODE &&
!node.textContent.trim()) {
            element.removeChild(node);
        }
    });
}

```

Властивості `tagName` та `nodeName` надають інформацію про ідентифікацію вузлів, але поведуться по-різному залежно від типу вузла. Розуміння цих відмінностей допомагає писати більш точний та передбачуваний код при роботі з різними типами DOM вузлів.

Властивість `tagName` доступна тільки для `element nodes` і завжди повертає ім'я тега у верхньому регістрі, незалежно від того, як було написано тег у вихідному HTML коді. Ця поведінка забезпечує консистентність при порівнянні імен тегів та дозволяє уникнути проблем з регістром символів.

```
// HTML може бути написаний у будь-якому регістрі:
<DiV>, <div>, <DIV>
const elements = document.querySelectorAll('div, span,
p');

elements.forEach(element => {
    console.log(`Тег: ${element.tagName}`); // завжди
"DIV", "SPAN", "P"

    // Безпечне порівняння імен тегів
    if (element.tagName === 'DIV') {
        element.classList.add('div-element');
    }

    // Або використання toLowerCase() для читабельності
    const tagLower = element.tagName.toLowerCase();
    console.log(`Тег в нижньому регістрі:
${tagLower}`);
});

// Функція для перевірки типу елемента
function isElementOfType(element, tagName) {
    return element.nodeType === Node.ELEMENT_NODE &&
        element.tagName === tagName.toUpperCase();
}

console.log(isElementOfType(divElement, 'div')); //
true
```

Властивість `nodeName` є більш універсальною і працює з усіма типами вузлів, повертаючи різну інформацію залежно від типу вузла. Для елементів

nodeName поводиться ідентично до tagName, але для інших типів вузлів повертає специфічні ідентифікатори.

```
// Демонстрація nodeName для різних типів вузлів
function exploreNodeNames(parentElement) {
    parentElement.childNodes.forEach((node, index) => {
        console.log(`Вузол ${index}:`);
        console.log(`  Тип: ${node.nodeType}`);
        console.log(`  nodeName: "${node.nodeName}"`);

        if (node.nodeType === Node.ELEMENT_NODE) {
            console.log(`  tagName:
"${node.tagName}"`);
            console.log(`  Це HTML елемент`);
        } else if (node.nodeType === Node.TEXT_NODE) {
            console.log(`  Вміст:
"${node.textContent}"`);
        } else if (node.nodeType === Node.COMMENT_NODE) {
            console.log(`  Коментар:
"${node.textContent}"`);
        }
        console.log('---');
    });
}

// Тестування з різними типами вузлів
const testContainer = document.createElement('div');
testContainer.innerHTML = 'Текст <!-- коментар -->
<span>елемент</span> ще текст';
exploreNodeNames(testContainer);
```

DOM надає кілька принципово різних способів роботи з вмістом елементів, кожен з яких має свої переваги, недоліки та специфічні випадки використання. Розуміння цих відмінностей є ключовим для вибору правильного підходу в конкретних ситуаціях.

`innerHTML` представляє HTML вміст елемента як рядок, який може містити теги, атрибути та текст. Ця властивість надзвичайно потужна для швидкого створення складного контенту, але вимагає обережного поводження з точки зору безпеки та продуктивності.

```
const article = document.querySelector('.article');

// Отримання поточного HTML вмісту
const currentHTML = article.innerHTML;
console.log('Поточний HTML:', currentHTML);

// Створення складного HTML контенту
article.innerHTML = `
    <header>
        <h1>Заголовок статті</h1>
        <time datetime="${new
Date().toISOString()}">${new
Date().toLocaleDateString()}</time>
    </header>
    <section>
        <p>Перший абзац з <strong>важливим</strong>
текстом.</p>
        <p>Другий абзац з <a
href="#more">посиланням</a>.</p>
    </section>
`;
```

```
// Додавання HTML до існуючого вмісту
article.innerHTML += '<footer><p>Автор: Іван
Іваненко</p></footer>';

// Функція для безпечного додавання HTML
function safeAppendHTML(container, htmlString) {
    const tempDiv = document.createElement('div');
    tempDiv.innerHTML = htmlString;

    // Переносимо створені елементи в цільовий
контейнер
    while (tempDiv.firstChild) {
        container.appendChild(tempDiv.firstChild);
    }
}
```

Важливо розуміти, що зміна `innerHTML` повністю замінює весь вміст елемента, що призводить до втрати всіх раніше прив'язаних обробників подій та може викликати проблеми з продуктивністю при частих оновленнях великих структур.

`textContent` працює виключно з текстовим вмістом, ігноруючи всі HTML теги та відображаючи їх як звичайний текст. Ця властивість є безпечнішою альтернативою `innerHTML` для роботи з користувацькими даними, оскільки автоматично екранує всі HTML теги, запобігаючи XSS атакам.

```
const userDataContainer =
document.querySelector('.user-content');

// Безпечне встановлення користувацького тексту
const userInput = '<script>alert("XSS
атака!");</script>Звичайний текст';
userDataContainer.textContent = userInput;
```

```
// Результат: відображається як звичайний текст, скрипт
не виконується

// Отримання чистого тексту з елемента, що містить HTML
const mixedContent = document.querySelector('.mixed-
content');
mixedContent.innerHTML = 'Текст з
<strong>жирним</strong> та <em>курсивним</em>
форматуванням';

console.log('HTML вміст:', mixedContent.innerHTML);
// "Текст з <strong>жирним</strong> та
<em>курсивним</em> форматуванням"

console.log('Текстовий вміст:',
mixedContent.textContent);
// "Текст з жирним та курсивним форматуванням"

// Практичне використання для очищення тексту
function extractCleanText(element) {
    return element.textContent.replace(/\s+/g, '
').trim();
}

const cleanText = extractCleanText(mixedContent);
console.log('Очищений текст:', cleanText);
```

`innerText` є найбільш складною властивістю для роботи з текстом, оскільки вона враховує CSS стилізацію та поводиться максимально наближено до того, як користувач бачить текст на екрані. Ця властивість ігнорує приховані

елементи, враховує CSS властивості white-space та намагається відтворити візуальну структуру тексту.

```
// Демонстрація відмінностей між textContent та
innerText
const styledElement = document.createElement('div');
styledElement.innerHTML = `
    <p>Видимий текст</p>
    <p style="display: none;">Прихований текст</p>
    <p style="visibility: hidden;">Невидимий текст</p>
    <span>Інлайн текст</span>
`;
document.body.appendChild(styledElement);

console.log('textContent (весь текст):');
console.log(styledElement.textContent);
// Покаже весь текст, включаючи прихований

console.log('innerText (тільки видимий):');
console.log(styledElement.innerText);
// Покаже тільки "Видимий текст" та "Інлайн текст"

// Практичне використання для аналізу видимого контенту
function getVisibleTextLength(element) {
    return element.innerText.length;
}

function hasHiddenContent(element) {
    return element.textContent.length !==
element.innerText.length;
}
```



```
console.log('Елемент має прихований контент:',
hasHiddenContent(styledElement));
```

Розуміння типів вузлів дозволяє створювати універсальні функції для обробки DOM структур, які коректно працюють з будь-якими типами вмісту:

```
// Універсальна функція для обходу DOM дерева
class DOMWalker {
    constructor(rootElement) {
        this.root = rootElement;
        this.elementCount = 0;
        this.textCount = 0;
        this.commentCount = 0;
    }

    walkAndAnalyze() {
        this.processNode(this.root);

        return {
            elements: this.elementCount,
            textNodes: this.textCount,
            comments: this.commentCount,
            total: this.elementCount + this.textCount +
this.commentCount
        };
    }

    processNode(node) {
        // Обробка поточного вузла
        switch(node.nodeType) {
            case Node.ELEMENT_NODE:
```

```

        this.elementCount++;
        this.processElement(node);
        break;
    case Node.TEXT_NODE:
        if (node.textContent.trim()) {
            this.textCount++;
            this.processText(node);
        }
        break;
    case Node.COMMENT_NODE:
        this.commentCount++;
        this.processComment(node);
        break;
    }

    // Рекурсивна обробка дочірніх вузлів
    node.childNodes.forEach(child =>
this.processNode(child));
    }

    processElement(element) {
        console.log(`Елемент:
<${element.tagName.toLowerCase()}>`);
        if (element.id) console.log(` ID:
${element.id}`);
    }

    processText(textNode) {
        const content = textNode.textContent.trim();

```

```

        console.log(`Текст: "${content.substring(0,
30)}${content.length > 30 ? '...' : ''}"`);
    }

    processComment(commentNode) {
        console.log(`Коментар:
"${commentNode.textContent}"`);
    }
}

// Використання аналізатора
const walker = new DOMWalker(document.body);
const stats = walker.walkAndAnalyze();
console.log('Статистика DOM:', stats);

```

### **Специфіка роботи з HTML та текстовим вмістом**

Вибір між `innerHTML`, `textContent` та `innerText` залежить від конкретного завдання та вимог безпеки. Кожна з цих властивостей має свої переваги та обмеження, які важливо враховувати при розробці.

`innerHTML` найкращий для створення складної розмітки, коли потрібно швидко згенерувати HTML структуру з динамічними даними:

```

// Генерація HTML таблиці з даних
function createDataTable(data, container) {
    const tableHTML = `
        <table class="data-table">
            <thead>
                <tr>
                    ${Object.keys(data[0]).map(key =>
`<th>${key}</th>`).join('')}
                </tr>
            </thead>
    `
}

```

```

        <tbody>
            ${data.map(row => `
                <tr>
                    ${Object.values(row).map(value
=> `<td>${value}</td>`).join('')}
                </tr>
            `).join('')}
        </tbody>
    </table>
`;

    container.innerHTML = tableHTML;
}

// Використання
const sampleData = [
    { name: 'Іван', age: 25, city: 'Київ' },
    { name: 'Марія', age: 30, city: 'Львів' }
];
createDataTable(sampleData,
document.querySelector('.table-container'));

```

`textContent` ідеальний для безпечної роботи з користувацькими даними та для випадків, коли потрібно отримати або встановити тільки текстовий вміст:

```

// Безпечне відображення користувацьких даних
function displayUserMessage(messageContainer,
userMessage) {
    // Навіть якщо userMessage містить HTML теги, вони
    будуть показані як текст
    messageContainer.textContent = userMessage;
}

```

```

    // Додавання часової мітки
    const timestamp = document.createElement('span');
    timestamp.className = 'timestamp';
    timestamp.textContent = new
Date().toLocaleTimeString();
    messageContainer.appendChild(timestamp);
}

// Функція для підрахунку слів у тексті
function countWords(element) {
    const text = element.textContent.trim();
    return text ? text.split(/\s+/).length : 0;
}

const article = document.querySelector('article');
console.log(`Стаття містить ${countWords(article)}
слів`);

```

`innerText` корисний для роботи з видимим контентом та створення функцій, які імітують поведінку копіювання тексту користувачем:

```

// Функція для копіювання видимого тексту
function copyVisibleText(element) {
    const visibleText = element.innerText;
    navigator.clipboard.writeText(visibleText).then(()
=> {
        console.log('Видимий текст скопійовано в буфер
обміну');
    });
}

```

```
// Порівняння обсягу всього та видимого контенту
function analyzeContentVisibility(element) {
    const totalText = element.textContent.length;
    const visibleText = element.innerText.length;
    const hiddenPercentage = ((totalText - visibleText)
/ totalText * 100).toFixed(1);

    console.log(`Загальний текст: ${totalText}
символів`);
    console.log(`Видимий текст: ${visibleText}
символів`);
    console.log(`Прихований контент:
${hiddenPercentage}%`);

    return {
        total: totalText,
        visible: visibleText,
        hiddenPercentage: parseFloat(hiddenPercentage)
    };
}

// Тестування з елементом, що має приховані частини
const testElement = document.createElement('div');
testElement.innerHTML = `
    <p>Видимий абзац</p>
    <p style="display: none;">Прихований абзац</p>
    <span>Видимий спан</span>
`;
document.body.appendChild(testElement);
```

```
analyzeContentVisibility(testElement);
```

Правильне розуміння та використання цих властивостей дозволяє створювати більш ефективні, безпечні та передбачувані веб-додатки, які коректно обробляють різні типи контенту та забезпечують кращий користувацький досвід.

### 3.2.8 Атрибути та властивості

Для початку важливо зрозуміти, що атрибути та властивості - це два різні способи зберігання інформації про HTML елементи, які працюють на різних рівнях.

HTML атрибути існують в самій HTML розмітці як текстові значення, які ми прописуємо прямо в тегах. Наприклад, коли ми пишемо `<input type="text" value="Привіт">`, то `type` та `value` - це атрибути, які браузер бачить під час парсингу HTML.

DOM властивості з'являються тоді, коли браузер перетворює HTML у живі JavaScript об'єкти. Кожен HTML елемент стає об'єктом з багатьма властивостями, деякі з яких відповідають атрибутам, а інші існують незалежно.

Уявіть собі, що HTML атрибути - це як написи на папері, а DOM властивості - це як значення у змінних комп'ютерної програми. Іноді вони синхронізуються, але не завжди.

Що відбувається під час завантаження веб-сторінки? Коли ви відкриваєте веб-сторінку, відбувається складний процес перетворення статичного HTML коду на інтерактивний інтерфейс. Давайте простежимо цей процес крок за кроком:

Крок 1: Браузер читає HTML

```
<input type="text" value="Привіт" class="my-input"
id="greeting">
```

На цьому етапі браузер бачить просто текст з тегами. Атрибути type, value, class, id - це просто рядки символів у файлі.

## Крок 2: Парсинг та створення DOM дерева

Браузер аналізує цей HTML і створює в пам'яті JavaScript об'єкт, який представляє цей елемент:

```
// Браузер створює щось подібне до цього об'єкта:
const inputObject = {
  tagName: 'INPUT',
  type: 'text',           // властивість, створена на
основі атрибута
  value: 'Привіт',        // властивість, створена на
основі атрибута
  className: 'my-input',  // властивість, створена
на основі атрибута class
  id: 'greeting',         // властивість, створена на
основі атрибута

  // Плюс багато інших властивостей, які браузер
додає автоматично:
  disabled: false,
  readOnly: false,
  required: false,
  placeholder: '',
  // ... і ще десятки інших
};
```

Чому це два різні рівні? HTML рівень (статичний): HTML атрибути існують на декларативному рівні - вони описують, яким елемент повинен бути спочатку. Це як рецепт для приготування страви:

```
<!-- Це "рецепт" для створення поля вводу -->
```



```
<input type="email" value="" placeholder="Введіть
email" required>
```

Ці атрибути:

- завжди є рядками (текстом);
- не змінюються автоматично під час роботи сторінки;
- зберігають початкові налаштування;
- видимі в HTML коді сторінки.

DOM рівень (динамічний): DOM властивості існують на функціональному рівні - це живі характеристики об'єкта, з якими може працювати JavaScript:

```
const emailInput =
document.querySelector('input[type="email"]');

// Це вже не просто текст, а справжні JavaScript
значення:
console.log(emailInput.type);      // "email" (рядок)
console.log(emailInput.required); // true (булеве
значення!)
console.log(emailInput.value);     // "" (поточне
значення, може змінюватися)
```

Давайте простежимо, як один HTML елемент перетворюється на DOM об'єкт:

Початковий HTML:

```
<button type="submit" disabled class="btn primary"
data-action="save">
    Зберегти
</button>
```

Що створює браузер в пам'яті:

```
// Псевдокод того, що створює браузер:
const buttonObject = {
```

```

// Властивості, що відповідають атрибутам:
type: 'submit',          // з атрибута type
disabled: true,          // з атрибута disabled
(перетворено в boolean!)

className: 'btn primary', // з атрибута class

// Властивості dataset для data-атрибутів:
dataset: {
    action: 'save'       // з атрибута data-action
},

// Властивості, які браузер додає автоматично:
tagName: 'BUTTON',
innerHTML: 'Зберегти',
textContent: 'Зберегти',
offsetWidth: 120,        // ширина елемента в
підселях
offsetHeight: 40,        // висота елемента в
підселях

// Методи для роботи з атрибутами:
getAttribute: function(name) { /* ... */ },
setAttribute: function(name, value) { /* ... */ },
hasAttribute: function(name) { /* ... */ },
removeAttribute: function(name) { /* ... */ }
};

```

Давайте розглянемо конкретний приклад, який демонструє цю різницю.

```

// У нас є HTML: <input type="text" value="початковий
текст">
const input = document.querySelector('input');

```

```

console.log('=== ПОЧАТКОВИЙ СТАН ===');
console.log('Атрибут value:',
input.getAttribute('value')); // "початковий текст"
console.log('Властивість value:', input.value);
// "початковий текст"

// Користувач вводить новий текст (або ми змінюємо
програмно)
input.value = 'новий текст';

console.log('=== ПІСЛЯ ЗМІНИ ===');
console.log('Атрибут value:',
input.getAttribute('value')); // все ще "початковий
текст"
console.log('Властивість value:', input.value);
// "новий текст"

// Якщо ми змінимо атрибут:
input.setAttribute('value', 'атрибут змінено');

console.log('=== ПІСЛЯ ЗМІНИ АТРИБУТА ===');
console.log('Атрибут value:',
input.getAttribute('value')); // "атрибут змінено"
console.log('Властивість value:', input.value);
// все ще "новий текст"

```

Властивість `value` зберігає поточне значення поля (те, що бачить користувач), а атрибут `value` зберігає початкове значення з HTML (або останнє значення, встановлене через `setAttribute`).

Розуміння цієї різниці критично важливе для:

1. Роботи з формами - щоб знати, як отримати поточні значення полів.
2. Збереження початкових станів - щоб можна було скинути форму до початкового стану.
3. Налаштування коду - щоб розуміти, чому іноді зміни не відображаються так, як очікується.
4. Продуктивності - властивості швидші за методи роботи з атрибутами.
5. Правильної типізації - властивості мають правильні типи даних, атрибути завжди рядки.

Це як різниця між кресленням будинку (HTML атрибути) та реальним будинком, в якому живуть люди (DOM властивості). Креслення показує, яким будинок задумувався, а реальний будинок може змінюватися - в ньому включають і виключають світло, відчиняють двері, переставляють меблі.

### Детальна робота з атрибутами

Для роботи з атрибутами JavaScript надає нам чотири основні методи, які дозволяють повністю контролювати атрибути будь-якого елемента:

```
// Припустимо, у нас є такий HTML:
// <a href="https://example.com" title="Перейти на
сайт" class="external-link">Посилання</a>

const link = document.querySelector('a');

// getAttribute() - отримує значення атрибута як рядок
const hrefValue = link.getAttribute('href');
console.log(hrefValue); // "https://example.com"

const titleValue = link.getAttribute('title');
console.log(titleValue); // "Перейти на сайт"

// Якщо атрибута не існує, повертається null
const nonExistent = link.getAttribute('data-test');
```

```
console.log(nonExistent); // null
```

Метод `setAttribute()` дозволяє нам додавати нові атрибути або змінювати існуючі:

```
// Встановлення нового атрибута
link.setAttribute('target', '_blank');
// Тепер HTML виглядає так: <a href="..."
target="_blank">

// Зміна існуючого атрибута
link.setAttribute('title', 'Новий заголовок');

// Можна встановлювати будь-які атрибути, навіть
нестандартні
link.setAttribute('data-tracking-id', 'external-link-
123');
```

Для перевірки наявності атрибута використовуємо `hasAttribute()`:

```
if (link.hasAttribute('target')) {
    console.log('Посилання відкриється в новому
вікні');
} else {
    console.log('Посилання відкриється в поточному
вікні');
}

// Перевірка кастомних атрибутів
if (link.hasAttribute('data-tracking-id')) {
    console.log('У цього елемента є ідентифікатор для
аналітики');
}
```

Видалення атрибутів здійснюється через `removeAttribute()`:

```
// Видаляємо атрибут title
link.removeAttribute('title');

// Тепер getAttribute('title') поверне null
console.log(link.getAttribute('title')); // null
```

Data-атрибути - це спеціальний тип кастомних атрибутів, які починаються з data-. Вони призначені для зберігання додаткової інформації про елемент, яка не впливає на його відображення, але може бути корисною для JavaScript.

```
// HTML: <div class="product" data-id="p123" data-
category="electronics" data-price="999.99" data-in-
stock="true">

const product = document.querySelector('.product');

// Доступ через dataset (рекомендований спосіб)
console.log(product.dataset.id);          // "p123"
console.log(product.dataset.category);    // "electronics"
console.log(product.dataset.price);       // "999.99"
console.log(product.dataset.inStock);     // "true"
(зверніть увагу на camelCase)

// Встановлення нових data-атрибутів
product.dataset.discount = '10';          // створює
data-discount="10"
product.dataset.lastViewed = '2025-01-15'; // створює
data-last-viewed="2025-01-15"
```

Зверніть увагу, що JavaScript автоматично перетворює назви з дефісами у camelCase. Атрибут data-in-stock стає dataset.inStock, а data-last-viewed стає dataset.lastViewed.

## Глибоке розуміння DOM властивостей

DOM властивості - це справжні JavaScript властивості об'єктів, які представляють HTML елементи. Вони можуть мати різні типи даних: рядки, числа, булеві значення, об'єкти тощо.

Різниця типів між атрибутами та властивостями:

```
const checkbox =
document.querySelector('input[type="checkbox"]');

// Атрибут завжди рядок або null
console.log(checkbox.getAttribute('checked')); //
"checked" або null

// Властивість має відповідний тип
console.log(checkbox.checked); // true або false
(булеве значення)

// Числові значення
const range =
document.querySelector('input[type="range"]');
console.log(range.getAttribute('min')); // "0" (рядок)
console.log(range.min); // 0 (число)
```

Одна з найцікавіших особливостей - це різниця між атрибутами та властивостями для URL:

```
// HTML: <a href="contact.html">Контакти</a>
const link = document.querySelector('a');

console.log(link.getAttribute('href')); //
"contact.html" (як написано в HTML)
console.log(link.href); //
"http://localhost:3000/contact.html" (повна URL)
```

```
// Те саме з зображеннями
// HTML: 
const img = document.querySelector('img');
console.log(img.getAttribute('src')); // "logo.png"
console.log(img.src); //
"http://localhost:3000/logo.png"
```

Елементи форм мають багато корисних властивостей, які відрізняються від атрибутів:

```
const textInput =
document.querySelector('input[type="text"]');

// value - одна з найважливіших властивостей
textInput.value = 'Новий текст';
console.log(textInput.value); // "Новий текст"

// Але атрибут value залишається незмінним
console.log(textInput.getAttribute('value')); //
початкове значення з HTML

// Інші корисні властивості форм
console.log(textInput.disabled); // булеве значення
console.log(textInput.required); // булеве значення
console.log(textInput.readOnly); // булеве значення
```

### **Складні випадки синхронізації**

Найскладніша частина розуміння атрибутів і властивостей - це знати, коли вони синхронізуються, а коли ні.

Одностороння синхронізація (атрибут → властивість):

Більшість стандартних атрибутів синхронізуються з властивостями в одну сторону:



```
const element = document.querySelector('#my-element');

// Зміна атрибута впливає на властивість
element.setAttribute('id', 'new-id');
console.log(element.id); // "new-id"

// Зміна властивості впливає на атрибут
element.className = 'new-class';
console.log(element.getAttribute('class')); // "new-
class"
```

Відсутність синхронізації для value:

Найважливіший виняток - це атрибут value у полях вводу:

```
// HTML: <input type="text" value="початковий текст">
const input = document.querySelector('input');

console.log(input.getAttribute('value')); //
"початковий текст"
console.log(input.value); //
"початковий текст"

// Користувач вводить новий текст або ми змінюємо
програмно
input.value = 'новий текст';

console.log(input.getAttribute('value')); // все ще
"початковий текст"
console.log(input.value); // "новий
текст"
```

```
// Атрибут value зберігає початкове значення, а
властивість value - поточне
```

Це дуже важливо розуміти при роботі з формами, оскільки `defaultValue` зберігає початкове значення, а `value` - поточне.

### Практичні приклади використання

Динамічне керування атрибутами зображень:

```
const gallery = document.querySelector('.gallery');
const images = gallery.querySelectorAll('img');

images.forEach((img, index) => {
    // Додаємо атрибути для кращої доступності
    img.setAttribute('alt', `Зображення галереї номер
${index + 1}`);

    // Додаємо data-атрибути для функціональності
    img.dataset.originalSrc = img.src;
    img.dataset.index = index;

    // Додаємо обробник події
    img.addEventListener('click', function() {
        const originalSrc = this.dataset.originalSrc;
        const imageIndex = this.dataset.index;
        console.log(`Клік по зображенню ${imageIndex}:
${originalSrc}`);
    });
});
```

Робота з формами та валідацією:

```
const form = document.querySelector('#registration-
form');
```

```
const emailInput =
form.querySelector('input[type="email"]');
const submitButton =
form.querySelector('button[type="submit"]');

// Функція для перевірки валідності
function validateEmail() {
    const email = emailInput.value;
    const isValid = email.includes('@') && email.length
> 5;

    if (isValid) {
        // Видаляємо атрибут, що показує помилку
        emailInput.removeAttribute('aria-invalid');
        emailInput.classList.remove('error');

        // Активуємо кнопку
        submitButton.disabled = false;
        submitButton.removeAttribute('title');
    } else {
        // Додаємо атрибути для позначення помилки
        emailInput.setAttribute('aria-invalid',
'true');
        emailInput.classList.add('error');

        // Блокуємо кнопку
        submitButton.disabled = true;
        submitButton.setAttribute('title', 'Спочатку
виправте помилки у формі');
    }
}
```

```
}
```

```
emailInput.addEventListener('input', validateEmail);
```

### Створення динамічних компонентів:

```
// Функція для створення картки товару
function createProductCard(productData) {
    const card = document.createElement('div');
    card.className = 'product-card';

    // Встановлюємо data-атрибути для зберігання даних
    card.dataset.productId = productData.id;
    card.dataset.category = productData.category;
    card.dataset.price = productData.price;

    // Встановлюємо атрибути для доступності
    card.setAttribute('role', 'article');
    card.setAttribute('aria-label', `Товар:
    ${productData.name}`);

    // Додаємо контент
    card.innerHTML = `
        <h3>${productData.name}</h3>
        <p>Ціна: ${productData.price} грн</p>
        <button data-action="add-to-cart">Додати в
кошик</button>
    `;

    return card;
}
```

```
// Використання
const productContainer =
document.querySelector('.products');
const newProduct = createProductCard({
    id: 'p456',
    name: 'Смартфон',
    category: 'electronics',
    price: '15999'
});

productContainer.appendChild(newProduct);
```

### **Коли використовувати атрибути, а коли властивості**

Коли використовувати АТРИБУТИ:

#### **1. Робота з кастомними даними (Data-атрибути)**

Data-атрибути - це ваш спосіб додати до HTML елементів додаткову інформацію, яка не впливає на відображення, але потрібна для JavaScript логіки.

```
// HTML: <div class="product-card" data-product-
id="123" data-category="electronics" data-
price="999.99">

const productCard = document.querySelector('.product-
card');

// Отримуємо інформацію про товар
const productId = productCard.dataset.productId;      //
"123"
const category = productCard.dataset.category;        //
"electronics"
```

```

const price = productCard.dataset.price; //
"999.99"

// Використовуємо для бізнес-логіки
function addToCart() {
    const item = {
        id: productCard.dataset.productId,
        category: productCard.dataset.category,
        price: parseFloat(productCard.dataset.price)
    };

    console.log('Додаємо до кошика:', item);
    // Відправляємо дані на сервер...
}

```

Чому data-атрибути, а не властивості?

- Вони зберігаються в HTML і видимі при перегляді коду сторінки
- Їх можна легко прочитати через CSS селектори
- Вони не конфліктують зі стандартними властивостями браузера
- Вони автоматично доступні через dataset

## 2. Збереження початкових значень

Атрибути зберігають оригінальний стан елемента, що дуже корисно для функцій скидання або порівняння:

```

// HTML форма з початковими значеннями
// <input type="text" value="Іван Петренко"
name="fullName">
// <input type="email" value="ivan@example.com"
name="email">

const form = document.querySelector('form');
const inputs = form.querySelectorAll('input');

```

```

// Функція для скидання форми до початкових значень
function resetToOriginal() {
    inputs.forEach(input => {
        // Отримуємо початкове значення з атрибута
        const originalValue =
input.getAttribute('value') || '';

        // Встановлюємо як поточне значення
        input.value = originalValue;

        console.log(`Скинуто ${input.name}:
"${originalValue}"`);
    });
}

// Функція для перевірки, чи змінилася форма
function hasFormChanged() {
    return Array.from(inputs).some(input => {
        const currentValue = input.value;
        const originalValue =
input.getAttribute('value') || '';
        return currentValue !== originalValue;
    });
}

// Попередження перед закриттям сторінки, якщо є
незбережені зміни
window.addEventListener('beforeunload', function(event)
{

```

```

    if (hasFormChanged()) {
        event.preventDefault();
        return 'У вас є незбережені зміни. Точно хочете
закрити сторінку?';
    }
});

```

### 3. Робота з HTML розміткою (серверний рендеринг)

Коли ви працюєте з серверними технологіями (PHP, Node.js, Python), часто потрібно, щоб зміни відображалися в HTML коді:

```

// Припустимо, ви створюєте систему управління
контентом
const article = document.querySelector('article');

// Зміна через атрибути зберігається в HTML
article.setAttribute('data-status', 'published');
article.setAttribute('data-author-id', '456');
article.setAttribute('data-last-modified', new
Date().toISOString());

// Тепер якщо сервер "прочитає" цей HTML, він побачить
ці зміни
// Це важливо для SEO, серверного рендерингу та
збереження стану

```

### 4. Сумісність зі старими браузерми

У старих браузерах (особливо Internet Explorer) деякі DOM властивості працювали нестабільно:

```

// Для максимальної сумісності з IE8-IE10
const element = document.querySelector('.legacy-
element');

```



```
// Надійніше в старих браузерх
element.setAttribute('aria-expanded', 'true');
element.setAttribute('role', 'button');

// Замість (що могло не працювати в IE):
// element.ariaExpanded = 'true'; // ця властивість не
існувала в старих браузерх
```

Коли використовувати ВЛАСТИВОСТІ:

#### 1. Робота з поточними значеннями (особливо форми)

Властивості відображають поточний стан елемента, що особливо важливо для інтерактивних елементів:

```
const searchInput = document.querySelector('#search');
const searchButton = document.querySelector('#search-
btn');
const resultsContainer =
document.querySelector('#results');

// Відстежуємо введення в реальному часі
searchInput.addEventListener('input', function() {
    const currentQuery = searchInput.value; // поточне
значення (властивість)

    // Активуємо кнопку лише якщо є текст
    if (currentQuery.length > 0) {
        searchButton.disabled = false;
        searchButton.textContent = `Шукати
"${currentQuery}"`;
    } else {
        searchButton.disabled = true;
        searchButton.textContent = 'Введіть запит';
```

```

    }

    // Показуємо кількість символів
    console.log(`Введено символів:
    ${currentQuery.length}`);
  });

  // Для порівняння - атрибут value не змінюється:
  searchInput.addEventListener('input', function() {
    console.log('Поточне значення (властивість):',
    this.value);
    console.log('Початкове значення (атрибут):',
    this.getAttribute('value'));
    // Друга console.log завжди показуватиме те саме
    значення!
  });

```

## 2. Правильні типи даних

Властивості автоматично перетворюють значення у відповідні типи, що робить код більш надійним:

```

// HTML: <input type="range" min="0" max="100"
value="50">
const slider =
document.querySelector('input[type="range"]');

console.log('=== АТРИБУТИ (завжди рядки) ===');
console.log(typeof slider.getAttribute('min'));    //
"string"
console.log(typeof slider.getAttribute('max'));    //
"string"

```

```

console.log(typeof slider.getAttribute('value')); //
"string"

console.log('=== ВЛАСТИВОСТІ (правильні типи) ===');
console.log(typeof slider.min);                // "number"
console.log(typeof slider.max);                // "number"
console.log(typeof slider.valueAsNumber); // "number"

// Практичне використання
function updateSliderDisplay() {
    const currentValue = slider.valueAsNumber; //
число, не рядок!
    const percentage = (currentValue / slider.max) *
100;

    console.log(`Слайдер на
${percentage.toFixed(1)}%`);

    // Можемо робити математичні операції без
parseInt/parseFloat
    if (currentValue > slider.max * 0.8) {
        console.log('Високе значення!');
    }
}

```

### 3. Швидкість виконання

Доступ до властивостей значно швидший, що важливо в циклах або при частих зверненнях:

```

const items = document.querySelectorAll('.list-item');

console.time('Через властивості');

```

```

for (let i = 0; i < items.length; i++) {
    // Швидко - прямий доступ до властивості
    const isHidden = items[i].hidden;
    const text = items[i].textContent;
}
console.timeEnd('Через властивості');

console.time('Через атрибути');
for (let i = 0; i < items.length; i++) {
    // Повільніше - виклик методів
    const isHidden = items[i].hasAttribute('hidden');
    const text = items[i].getAttribute('textContent');
    // не працюватиме правильно!
}
console.timeEnd('Через атрибути');

```

#### 4. Булеві значення (найважливіший випадок!)

Булеві атрибути в HTML працюють по-особливому - їхня присутність означає true, відсутність - false. Але з властивостями набагато простіше:

```

const checkbox =
document.querySelector('input[type="checkbox"]');
const submitButton = document.querySelector('#submit');

// СКЛАДНО з атрибутами:
if (checkbox.hasAttribute('checked')) {
    // checkbox активований
    submitButton.removeAttribute('disabled');
} else {
    // checkbox не активований
    submitButton.setAttribute('disabled', '');
}

```

```
// ПРОСТО з властивостями:
if (checkbox.checked) {
    submitButton.disabled = false;
} else {
    submitButton.disabled = true;
}

// Або ще простіше:
submitButton.disabled = !checkbox.checked;
```

В практичній розробці часто доводиться комбінувати обидва підходи:

```
// Компонент модального вікна
class Modal {
    constructor(element) {
        this.element = element;
        this.isOpen = false;
    }

    open() {
        // Властивості для швидкої роботи з поточним
        станом
        this.isOpen = true;
        this.element.hidden = false;

        // Атрибути для доступності та CSS
        this.element.setAttribute('aria-modal',
'true');
        this.element.setAttribute('role', 'dialog');

        // Data-атрибут для збереження стану
```

```
        this.element.dataset.state = 'open';

        console.log('Модальне вікно відкрито');
    }

    close() {
        this.isOpen = false;
        this.element.hidden = true;

        this.element.setAttribute('aria-modal',
'false');
        this.element.dataset.state = 'closed';

        console.log('Модальне вікно закрито');
    }

    // Метод для скидання до початкового стану
    reset() {
        // Використовуємо атрибути для відновлення
        // початкового стану
        const originalState =
this.element.getAttribute('data-original-state') ||
'closed';

        if (originalState === 'open') {
            this.open();
        } else {
            this.close();
        }
    }
}
```

```
}
```

Підсумок:

Використовуйте атрибути, коли:

- Зберігаєте метадані або конфігурацію (data-config, data-id)
- Працюєте з доступністю (aria-\*, role)
- Потрібно зберегти початковий стан
- Хочете, щоб зміни були видимі в HTML коді

Використовуйте властивості, коли:

- Читаєте/змінюєте поточний стан елемента
- Працюєте з числами, булевими значеннями
- Потрібна максимальна швидкість
- Логіка вимагає частих змін

Пам'ятайте: не існує "правильного" або "неправильного" вибору в абсолютному сенсі. Вибір залежить від конкретної задачі та контексту використання. Розуміння різниці дозволяє вам приймати обґрунтовані рішення у кожній ситуації.

### Складні випадки та підводні камені

Булеві атрибути в HTML працюють по-особливому - їхня присутність означає true, а відсутність - false:

```
const button = document.querySelector('button');

// Неправильно!
button.setAttribute('disabled', 'false'); // Кнопка все
одно буде заблокована!

// Правильно для атрибутів
button.removeAttribute('disabled'); // Розблоковує
кнопку
button.setAttribute('disabled', ''); // Блокує кнопку
```

```
// Правильно для властивостей
button.disabled = false; // Розблоковує кнопку
button.disabled = true;  // Блокує кнопку
```

Для роботи з класами є декілька способів:

```
const element = document.querySelector('.my-element');

// Через атрибут class
element.setAttribute('class', 'new-class another-
class');
console.log(element.getAttribute('class')); // "new-
class another-class"

// Через властивість className
element.className = 'different-class';
console.log(element.className); // "different-class"

// Через сучасний classList (найзручніший)
element.classList.add('active');
element.classList.remove('inactive');
element.classList.toggle('highlighted');
```

Стили CSS:

```
const element = document.querySelector('.styled-
element');

// Через атрибут style (рядок)
element.setAttribute('style', 'color: red; font-size:
16px;');

// Через властивість style (об'єкт)
```



```

element.style.color = 'blue';
element.style.fontSize = '18px';

// style.cssText для встановлення декількох стилів
одночасно
element.style.cssText = 'color: green; background:
yellow; padding: 10px;';

```

### **Продвинуті техніки роботи**

Масове керування атрибутами:

```

function setMultipleAttributes(element, attributes) {
    Object.entries(attributes).forEach(([key, value])
=> {
        if (value === null || value === undefined) {
            element.removeAttribute(key);
        } else {
            element.setAttribute(key, value);
        }
    });
}

// Використання
const modal = document.querySelector('.modal');
setMultipleAttributes(modal, {
    'aria-hidden': 'false',
    'role': 'dialog',
    'data-modal-id': 'confirmation',
    'title': null // видалить атрибут title
});

```

Спостереження за змінами атрибутів:

```

// MutationObserver для відстеження змін атрибутів

```

```

const observer = new
MutationObserver(function(mutations) {
    mutations.forEach(function(mutation) {
        if (mutation.type === 'attributes') {
            console.log(`Атрибут
${mutation.attributeName} змінився`);
            console.log(`Старе значення:
${mutation.oldValue}`);
            console.log(`Нове значення:
${mutation.target.getAttribute(mutation.attributeName)}
`);
        }
    });
});

const watchedElement =
document.querySelector('#watched');
observer.observe(watchedElement, {
    attributes: true,
    attributeOldValue: true,
    attributeFilter: ['class', 'data-status'] //
спостерігаємо лише за певними атрибутами
});

```

#### Практичні рекомендації:

1. Для зберігання даних використовуйте data-атрибути, але пам'ятайте, що всі значення зберігаються як рядки
2. Для швидкої роботи з поточними значеннями використовуйте властивості
3. Для булевих значень завжди використовуйте властивості (element.hidden = true)

4. Пам'ятайте про різницю між початковими та поточними значеннями в полях форм
5. Використовуйте `hasAttribute()` перед `getAttribute()` для уникнення помилок
6. Для роботи з класами використовуйте `classList` замість `className` або атрибута `class`

Типові помилки та як їх уникнути:

```
// ПОМИЛКА: Плутанина з булевими атрибутами
element.setAttribute('hidden', 'false'); // Елемент все
одно буде прихований!
```

```
// ПРАВИЛЬНО:
element.hidden = false; // або
element.removeAttribute('hidden');
```

```
// ПОМИЛКА: Очікування автоматичного перетворення типів
в атрибутах
element.setAttribute('data-count', 5);
console.log(element.dataset.count + 10); // "510"
(рядкова конкатенація!)
```

```
// ПРАВИЛЬНО:
element.setAttribute('data-count', '5');
console.log(parseInt(element.dataset.count) + 10); //
15
```

```
// ПОМИЛКА: Плутанина між початковим та поточним
значенням
const input = document.querySelector('input');
```

```
console.log(input.getAttribute('value')); // початкове  
значення  
console.log(input.value); // поточне  
значення
```

Розуміння різниці між атрибутами та властивостями є ключовим для ефективної роботи з DOM у JavaScript. Це знання дозволить вам писати більш надійний та передбачуваний код для веб-додатків.

### 3.2.9 Внесення змін в документ

Робота з документом у веб-розробці передбачає можливість не лише читання та пошуку елементів, але й активного втручання у структуру сторінки шляхом створення нових елементів, їх переміщення та видалення. Ця функціональність є основою для створення динамічних та інтерактивних веб-застосунків, де контент може змінюватися у відповідь на дії користувача або інші події.

#### Створення нових елементів

Процес створення нових HTML-елементів програмним шляхом починається з використання методу `createElement`, який є одним з найважливіших інструментів для динамічної побудови DOM-структури. Цей метод приймає в якості параметра назву HTML-тега і повертає новий елемент, який поки що існує лише в пам'яті браузера, але ще не відображається на сторінці.

```
// Створення базового div-елемента  
const newDiv = document.createElement('div');  
newDiv.className = 'dynamic-content';  
newDiv.textContent = 'Цей елемент створено динамічно';
```

У цьому прикладі ми створюємо новий `div`-елемент, присвоюємо йому CSS-клас для стилізації та додаємо текстовий вміст. Важливо розуміти, що на цьому етапі елемент існує як JavaScript-об'єкт, але ще не є частиною видимої структури документа.

Створення більш складних елементів, таких як посилання, вимагає налаштування додаткових атрибутів, які визначають поведінку та властивості елемента:

```
// Створення повнофункціонального посилання
const newLink = document.createElement('a');
newLink.href = 'https://example.com';
newLink.textContent = 'Перейти на сайт';
newLink.target = '_blank';
```

Для роботи з чистим текстом, без HTML-розмітки, DOM пропонує метод `createTextNode`, який створює спеціальний тип вузла, що містить лише текстову інформацію. Це особливо корисно, коли потрібно безпечно вставити текст, не турбуючись про можливе виконання HTML-коду:

```
const textNode = document.createTextNode('Простий
текст');
const paragraph = document.createElement('p');
paragraph.appendChild(textNode);
```

### **Вставка елементів у документ**

Після створення елементів наступним кроком є їх розміщення у потрібному місці документа. Традиційний метод `appendChild` дозволяє додати новий елемент як останню дитину вказаного батьківського елемента:

```
const container = document.querySelector('.container');
const newElement = document.createElement('p');
newElement.textContent = 'Новий абзац';

// Елемент буде додано в самий кінець контейнера
container.appendChild(newElement);
```

Коли потрібен більш точний контроль над позицією нового елемента, використовується метод `insertBefore`, який дозволяє вставити елемент перед конкретним існуючим елементом:

```
const referenceElement =
container.querySelector('.reference');
container.insertBefore(newElement, referenceElement);
```

Сучасні браузеры підтримують більш гнучкий та зручний метод `insertAdjacentHTML`, який дозволяє вставляти HTML-код у чотири різні позиції відносно вибраного елемента. Цей метод особливо зручний, коли потрібно швидко додати готову HTML-розмітку:

```
const element = document.querySelector('.target');

// Кожна з цих команд вставляє HTML у різну позицію
element.insertAdjacentHTML('beforebegin', '<div>Перед
елементом</div>');
element.insertAdjacentHTML('afterbegin', '<span>На
початок вмісту</span>');
element.insertAdjacentHTML('beforeend', '<span>В кінець
вмісту</span>');
element.insertAdjacentHTML('afterend', '<div>Після
елемента</div>');
```

Аналогічно працює метод `insertAdjacentElement`, але замість HTML-рядка він приймає готовий елемент-об'єкт:

### **Клонування та видалення елементів**

Клонування елементів є потужним інструментом для створення копій існуючих DOM-вузлів, що особливо корисно при роботі з шаблонами або повторюваними елементами. Метод `cloneNode` може працювати у двох режимах: поверхневому та глибокому клонуванні.

При поверхневому клонуванні копіюється лише сам елемент з його атрибутами, але без дочірніх елементів:

```
const originalElement =
document.querySelector('.template');
const shallowClone = originalElement.cloneNode(false);
```

Глибоке клонування створює повну копію елемента разом з усіма його нащадками, зберігаючи всю внутрішню структуру:

```
const deepClone = originalElement.cloneNode(true);  
document.body.appendChild(deepClone);
```

Видалення елементів з документа може здійснюватися кількома способами. Сучасний та найпростіший підхід полягає у використанні методу `remove`, який видаляє елемент безпосередньо:

```
const elementToRemove = document.querySelector('.old-content');  
elementToRemove.remove();
```

Традиційний спосіб вимагає звернення до батьківського елемента та використання методу `removeChild`:

```
elementToRemove.parentNode.removeChild(elementToRemove);
```

Коли потрібно очистити весь вміст контейнера, найшвидшим способом є присвоєння порожнього рядка властивості `innerHTML`:

```
const container = document.querySelector('.container');  
container.innerHTML = ''; // Миттєво видаляє весь  
внутрішній вміст
```

Всі ці методи разом формують потужний арсенал інструментів для динамічного управління структурою веб-сторінки, дозволяючи створювати складні інтерактивні інтерфейси, які можуть адаптуватися до потреб користувача та змінюватися в реальному часі.

### 3.2.10 Стилi та класи

Управління зовнішнім виглядом елементів веб-сторінки є одним з найважливіших аспектів створення сучасних інтерактивних інтерфейсів. JavaScript надає розробникам потужні інструменти для динамічного керування стилями та класами, що дозволяє створювати адаптивні та реактивні

користувацькі інтерфейси, які можуть змінюватися у відповідь на різноманітні події та умови.

Властивість `classList` є найсучаснішим та найзручнішим способом управління CSS-класами елементів, оскільки вона надає інтуїтивний набір методів для всіх основних операцій з класами. Ця властивість працює як спеціальний об'єкт, який містить список всіх класів елемента та методи для їх маніпулювання.

Додавання класів до елемента здійснюється за допомогою методу `add`, який може приймати як один клас, так і кілька класів одночасно:

```
const element = document.querySelector('.my-element');

// Додавання одного класу змінює стан елемента
element.classList.add('active');

// Можливість додати кілька класів одним викликом
значно спрощує код
element.classList.add('highlighted', 'important');
```

Видалення класів відбувається аналогічно через метод `remove`, який безпечно видаляє вказані класи, навіть якщо їх не існувало на елементі:

```
element.classList.remove('inactive');
```

Особливо корисним є метод `toggle`, який реалізує логіку перемикання: якщо клас присутній на елементі, він буде видалений, а якщо відсутній - додан. Це ідеально підходить для створення інтерактивних елементів, таких як меню, що розгортаються, або кнопки з двома станами:

```
// При кожному виклику стан класу 'visible' буде
змінюватися на протилежний
element.classList.toggle('visible');
```

Для перевірки наявності певного класу використовується метод `contains`, який повертає логічне значення:

```
if (element.classList.contains('active')) {
```



```
console.log('Елемент знаходиться в активному  
стані');  
}
```

Метод `replace` дозволяє одним викликом замінити один клас на інший, що корисно при зміні станів елемента:

### Пряме керування стилями

Властивість `style` надає можливість безпосереднього встановлення CSS-стилів на рівні конкретного елемента. Ці стилі мають найвищий пріоритет серед CSS-правил (окрім тих, що мають `!important`) і застосовуються як вбудовані стилі безпосередньо в HTML-атрибут `style`.

При роботі з окремими CSS-властивостями важливо пам'ятати про конвертацію імен: CSS-властивості, що містять дефіси, в JavaScript записуються у `camelCase` форматі:

```
const element = document.querySelector('.styled-  
element');  
  
// Базові стилі встановлюються через присвоєння  
рядкових значень  
element.style.color = 'red';  
element.style.fontSize = '18px';  
element.style.backgroundColor = '#f0f0f0';  
  
// CSS властивості з дефісами перетворюються в  
camelCase  
element.style.borderTopWidth = '2px';  
element.style.marginLeft = '10px';
```

Для встановлення великої кількості стилів одночасно зручно використовувати властивість `cssText`, яка приймає рядок з CSS-правилами у стандартному форматі:

```
element.style.cssText = 'color: blue; font-size: 20px;
margin: 10px;';
```

Видалення стилів може здійснюватися двома способами: через метод `removeProperty` або шляхом присвоєння порожнього рядка:

```
// Явне видалення конкретної CSS-властивості
element.style.removeProperty('color');

// Альтернативний спосіб через присвоєння порожнього
значення
element.style.color = '';
```

### Отримання обчислених стилів

Функція `getComputedStyle` є незамінним інструментом для отримання фінальних стилів елемента після обробки всіх CSS-правил браузером. На відміну від властивості `style`, яка показує лише вбудовані стилі, `getComputedStyle` повертає всі стилі, що фактично застосовуються до елемента, включаючи стилі з таблиць стилів, успадковані стилі та стилі браузера за замовчуванням:

```
const element = document.querySelector('.complex-
styled');
const computedStyles = getComputedStyle(element);

// Отримання фінальних значень стилів
console.log(computedStyles.color);           // реальний
колір тексту
console.log(computedStyles.fontSize);        // обчислений
розмір шрифту в пікселях
console.log(computedStyles.display);         // тип
відображення елемента
```

Особливою можливістю `getComputedStyle` є отримання стилів псевдоелементів, таких як `::before` та `::after`:

```
const beforeStyles = getComputedStyle(element,
'::before');
console.log(beforeStyles.content);
```

### Динамічна зміна CSS правил

Для більш складних сценаріїв, коли потрібно створювати стилі, що впливають на кілька елементів одночасно або створювати повністю нові CSS-правила, JavaScript дозволяє програмно створювати та вставляти елементи стилів у документ:

```
function addGlobalStyle(className, styles) {
    const style = document.createElement('style');
    style.textContent = `.${className} { ${styles} }`;
    document.head.appendChild(style);
}

// Створення глобального стилю для класу
addGlobalStyle('dynamic-highlight', 'background-color:
yellow; border: 2px solid orange;');
```

Після виконання цієї функції всі елементи з класом 'dynamic-highlight' автоматично отримають нові стилі, що особливо корисно для створення тем оформлення або динамічної зміни зовнішнього вигляду всього сайту.

Всі ці методи та властивості разом формують комплексну систему для повного контролю над зовнішнім виглядом веб-сторінки, дозволяючи створювати динамічні, адаптивні та візуально привабливі користувацькі інтерфейси, які можуть реагувати на дії користувача та зміни у стані додатку в реальному часі.

### 3.2.11 Координати елементів у DOM

Браузер використовує декілька різних систем координат, і розуміння їх відмінностей є критично важливим для ефективної роботи з позиціонуванням елементів на веб-сторінці.

## Системи координат у веб-браузері

Координати відносно вікна браузера (viewport coordinates) відраховуються від лівого верхнього кута видимої області браузера, тобто тієї частини сторінки, яку користувач бачить на екрані в даний момент. Ці координати не змінюються при прокручуванні сторінки, оскільки вони завжди прив'язані до видимого вікна. Для отримання таких координат використовується метод `getBoundingClientRect()`, який повертає об'єкт з властивостями `left`, `top`, `right`, `bottom`, а також `width` і `height` елемента.

```
// Простий приклад отримання координат кнопки
const button = document.querySelector('#my-button');
const rect = button.getBoundingClientRect();

console.log(`Кнопка знаходиться на позиції:`);
console.log(`Ліворуч від краю вікна: ${rect.left}px`);
console.log(`Зверху від краю вікна: ${rect.top}px`);
console.log(`Розміри: ${rect.width} x
${rect.height}px`);

// Приклад: перевірка, чи елемент видимий у вікні
function isVisible(element) {
    const rect = element.getBoundingClientRect();
    return rect.top >= 0 &&
        rect.left >= 0 &&
        rect.bottom <= window.innerHeight &&
        rect.right <= window.innerWidth;
}

if (isVisible(button)) {
    console.log('Кнопка повністю видима');
}
```

Координати відносно документа (document coordinates) відраховуються від самого початку HTML-документа і враховують величину прокрутки сторінки. Ці координати показують абсолютну позицію елемента в межах усього документа, незалежно від того, яка його частина видима користувачу в даний момент. Щоб отримати координати відносно документа, потрібно до координат відносно вікна додати поточну позицію прокрутки, яка зберігається у властивостях `window.pageXOffset` та `window.pageYOffset`.

```
// Функція для отримання координат відносно всього
документа
function getDocumentPosition(element) {
    const rect = element.getBoundingClientRect();
    return {
        x: rect.left + window.pageXOffset,
        y: rect.top + window.pageYOffset
    };
}

const docPos = getDocumentPosition(button);
console.log(`Позиція в документі: ${docPos.x},
${docPos.y}`);

// Приклад: прокрутка до елемента
function scrollToElement(element) {
    const pos = getDocumentPosition(element);
    window.scrollTo(pos.x, pos.y - 100); // відступ
100px зверху
}

// Використання
```

```
scrollToElement(document.querySelector('#target-
section'));
```

### Розміри елементів та їх особливості

DOM надає кілька різних способів отримання розмірів елементів, кожен з яких включає різні компоненти елемента. Властивості `clientWidth` та `clientHeight` повертають розміри елемента включно з внутрішніми відступами (`padding`), але без рамки (`border`) та смуги прокрутки (`scrollbar`). Ці властивості корисні, коли потрібно знати робочу область елемента.

```
const container = document.querySelector('.content-
box');

// Приклад отримання різних типів розмірів
console.log('=== Розміри елемента ===');
console.log(`clientWidth (без border):
${container.clientWidth}px`);
console.log(`offsetWidth (з border):
${container.offsetWidth}px`);
console.log(`scrollWidth (повний контент):
${container.scrollWidth}px`);

// Практичний приклад: визначення, чи потрібна
прокрутка
function needsScrolling(element) {
    return element.scrollWidth > element.clientWidth ||
           element.scrollHeight > element.clientHeight;
}

if (needsScrolling(container)) {
    console.log('Елемент має прокручуваний контент');
    container.style.overflowX = 'auto';
```

```

}

// Приклад: отримання позиції прокрутки
container.addEventListener('scroll', function() {
    const scrollPercent = (this.scrollTop /
    (this.scrollHeight - this.clientHeight)) * 100;
    console.log(`Прокручено на
    ${scrollPercent.toFixed(1)}%`);
});

```

Властивості `offsetWidth` та `offsetHeight` включають у себе все: внутрішні відступи, рамку та смугу прокрутки, тобто повертають повний видимий розмір елемента. Властивості `scrollWidth` та `scrollHeight` показують повні розміри вмісту елемента, включно з тією частиною, що прихована від перегляду через обмеження розмірів контейнера.

### Позиціонування та координати миші

При роботі з позиціонуванням елементів важливо розуміти концепцію `offsetParent` – це найближчий предок елемента, який має CSS-властивість `position` встановлену в значення, відмінне від `static`. Координати `offsetLeft` та `offsetTop` показують позицію елемента відносно цього батьківського контейнера.

```

// Простий приклад позиціонування елемента
function moveElement(element, x, y) {
    element.style.position = 'absolute';
    element.style.left = x + 'px';
    element.style.top = y + 'px';
}

// Центрування елемента на сторінці
function centerElement(element) {
    const rect = element.getBoundingClientRect();

```

```

    const centerX = (window.innerWidth - rect.width) /
2;
    const centerY = (window.innerHeight - rect.height)
/ 2;

    moveElement(element, centerX, centerY);
}

// Приклад використання
const popup = document.querySelector('.popup');
centerElement(popup);

// Позиціонування елемента біля іншого елемента
function positionNear(element, target, side = 'right')
{
    const targetRect = target.getBoundingClientRect();
    const elementRect =
element.getBoundingClientRect();

    let x, y;

    switch(side) {
        case 'right':
            x = targetRect.right + 10;
            y = targetRect.top;
            break;
        case 'bottom':
            x = targetRect.left;
            y = targetRect.bottom + 10;
            break;
    }
}

```



```

    }

    moveElement(element, x, y);
}

```

Події миші надають різні типи координат: `clientX` та `clientY` показують позицію курсора відносно вікна браузера, `pageX` та `pageY` – відносно документа (з врахуванням прокрутки), а `offsetX` та `offsetY` – відносно елемента, на який було здійснено клік.

```

// Простий приклад відстеження кліків
document.addEventListener('click', function(event) {
    console.log('=== Клік мишею ===');
    console.log(`Позиція в вікні: ${event.clientX},
${event.clientY}`);
    console.log(`Позиція в документі: ${event.pageX},
${event.pageY}`);

    // Створюємо маркер в місці кліка
    const marker = document.createElement('div');
    marker.style.cssText = `
        position: absolute;
        left: ${event.pageX - 5}px;
        top: ${event.pageY - 5}px;
        width: 10px;
        height: 10px;
        background: red;
        border-radius: 50%;
        pointer-events: none;
    `;
    document.body.appendChild(marker);
}

```

```

    // Видаляємо маркер через 2 секунди
    setTimeout(() => marker.remove(), 2000);
});

// Приклад: перетягування елемента
let isDragging = false;
let dragElement = null;
let startX, startY;

document.addEventListener('mousedown', function(event)
{
    if (event.target.classList.contains('draggable')) {
        isDragging = true;
        dragElement = event.target;
        startX = event.clientX -
dragElement.offsetLeft;
        startY = event.clientY - dragElement.offsetTop;
        dragElement.style.position = 'absolute';
    }
});

document.addEventListener('mousemove', function(event)
{
    if (isDragging && dragElement) {
        dragElement.style.left = (event.clientX -
startX) + 'px';
        dragElement.style.top = (event.clientY -
startY) + 'px';
    }
});

```

```
document.addEventListener('mouseup', function() {
    isDragging = false;
    dragElement = null;
});
```

### Оптимізація роботи з DOM

Одним із найважливіших аспектів ефективної роботи з DOM є мінімізація кількості звернень до нього. Кожен пошук елементів у DOM або зміна їх властивостей може бути ресурсномістким, особливо на складних сторінках з великою кількістю елементів. Тому рекомендується кешувати результати пошуку елементів у змінних та використовувати DocumentFragment для групової вставки великої кількості елементів.

При створенні анімацій слід використовувати requestAnimationFrame, який синхронізує виконання коду з частотою оновлення екрана браузера, забезпечуючи плавність анімації та оптимальне використання ресурсів системи. Функції пом'якшення (easing functions) допомагають створювати більш природні та привабливі анімаційні ефекти.

```
// Неефективний спосіб - багато звернень до DOM
function addItemBad() {
    const list = document.querySelector('#item-list');
    for (let i = 0; i < 100; i++) {
        const item = document.createElement('li');
        item.textContent = `Елемент ${i + 1}`;
        list.appendChild(item); // Кожен раз змінюємо
        DOM!
    }
}

// Ефективний спосіб - використання DocumentFragment
function addItemGood() {
```

```

const list = document.querySelector('#item-list');
const fragment = document.createDocumentFragment();

for (let i = 0; i < 100; i++) {
    const item = document.createElement('li');
    item.textContent = `Елемент ${i + 1}`;
    fragment.appendChild(item);
}

list.appendChild(fragment); // Одна зміна DOM
}

// Кешування результатів пошуку
const cachedElements = {
    header: document.querySelector('.header'),
    sidebar: document.querySelector('.sidebar'),
    content: document.querySelector('.main-content')
};

function updateInterface() {
    // Використовуємо кешовані елементи
    cachedElements.header.style.backgroundColor =
'blue';
    cachedElements.sidebar.classList.add('collapsed');
    cachedElements.content.style.marginLeft = '60px';
}

```

### Сучасні підходи та безпека

Для відстеження змін у DOM сучасні браузерери надають MutationObserver API, який дозволяє ефективно спостерігати за додаванням, видаленням або

зміною елементів та їх атрибутів. Цей підхід є набагато більш ефективним порівняно зі старими методами періодичної перевірки.

```
// Простий приклад MutationObserver
const targetContainer =
document.querySelector('.watched-container');

const observer = new
MutationObserver(function(mutations) {
    mutations.forEach(function(mutation) {
        if (mutation.type === 'childList') {
            console.log(`Додано
${mutation.addedNodes.length} елементів`);
            console.log(`Видалено
${mutation.removedNodes.length} елементів`);
        }

        if (mutation.type === 'attributes') {
            console.log(`Змінено атрибут:
${mutation.attributeName}`);
        }
    });
});

// Налаштування спостереження
observer.observe(targetContainer, {
    childList: true,    // слідкувати за дочірніми
елементами
    attributes: true,   // слідкувати за атрибутами
    subtree: true       // слідкувати за всіма нащадками
});
```

```
// Тестування - додамо елемент
setTimeout(() => {
    const newElement = document.createElement('p');
    newElement.textContent = 'Новий елемент';
    targetContainer.appendChild(newElement);
}, 2000);
```

При роботі з великими списками даних варто розглянути використання техніки віртуалізації, яка передбачає відображення лише тих елементів, що знаходяться в видимій області, значно покращуючи продуктивність сторінки.

Безпека залишається критично важливим аспектом при роботі з DOM, особливо при вставці користувацького контенту. Слід завжди використовувати безпечні методи створення та наповнення елементів, уникаючи прямої вставки HTML-коду через `innerHTML`, щоб запобігти XSS-атакам. Також важливо дотримуватися принципів веб-доступності, додаючи відповідні ARIA-атрибути та забезпечуючи підтримку навігації за допомогою клавіатури.

```
// НЕБЕЗПЕЧНО - може призвести до XSS атак
function unsafeMethod(userInput) {
    document.body.innerHTML += userInput; // НІКОЛИ ТАК
    НЕ РОБІТЬ!
}

// БЕЗПЕЧНО - правильний спосіб
function safeMethod(userText) {
    const element = document.createElement('div');
    element.textContent = userText; // автоматично
    екранує HTML
    document.body.appendChild(element);
}
```

```
// Приклад безпечного створення посилання
function createSafeLink(url, text) {
    const link = document.createElement('a');
    link.href = url;
    link.textContent = text;
    link.rel = 'noopener nofollow'; // додаткова
безпека
    link.target = '_blank';
    return link;
}

// Безпечна функція для перевірки елементів
function safeElementOperation(selector, callback) {
    const element = document.querySelector(selector);

    if (element) {
        try {
            callback(element);
        } catch (error) {
            console.error(`Помилка при роботі з
елементом ${selector}:`, error);
        }
    } else {
        console.warn(`Елемент "${selector}" не
знайдено`);
    }
}

// Використання безпечної функції
safeElementOperation('.my-button', function(button) {
```

```
button.addEventListener('click', function() {  
    console.log('Безпечний клік!');  
});  
});
```

### 3.3 Браузер: події

#### 3.3.1 Вступ до подій браузера

У світі веб-розробки події браузера є фундаментальним механізмом, який дозволяє створювати інтерактивні веб-сторінки, що реагують на дії користувача. Події в браузері представляють собою систему сповіщень, яка інформує наш код про те, що щось сталося в документі або з елементами сторінки - користувач клікнув мишею, натиснув клавішу на клавіатурі, завантажилася сторінка, або змінився розмір вікна браузера.

Концептуально подія - це сигнал від браузера про те, що відбулася якась взаємодія або зміна стану. Браузер постійно відслідковує різноманітні дії та автоматично генерує відповідні події. Наприклад, коли користувач наводить курсор миші на кнопку, браузер генерує подію `mouseover`, коли клікає - подію `click`, коли натискає клавішу - подію `keydown`.

Важливо розуміти, що події не існують самі по собі - вони завжди пов'язані з конкретними елементами DOM (Document Object Model). Кожен HTML-елемент може бути джерелом різноманітних подій, і ми, як розробники, можемо написати код, який буде реагувати на ці події, виконуючи певні дії або обчислення.

Система подій браузера побудована на принципі асинхронності - наш код не блокується в очікуванні подій, натомість браузер викликає наші функції-обробники тоді, коли відповідні події відбуваються. Це дозволяє створювати відзивчивий користувацький інтерфейс, який не "замерзає" під час виконання операцій.



### 3.3.2 Використання атрибута HTML

Найпростішим та найбільш інтуїтивним способом обробки подій є використання спеціальних атрибутів безпосередньо в HTML-розмітці. Ці атрибути мають назви, що починаються з префікса "on", після якого йде назва події. Наприклад, атрибут `onclick` призначений для обробки кліків миші, `onmouseover` - для обробки наведення курсора, `onload` - для обробки завантаження елемента.

Синтаксис використання атрибутів подій досить простий - ми вказуємо назву атрибута та в лапках записуємо JavaScript-код, який повинен виконатися при настанні події:

```
<button onclick="alert('Кнопка була  
натиснута!')">Натисни мене</button>  
  
<div onmouseover="this.style.backgroundColor =  
'yellow'"  
    onmouseout="this.style.backgroundColor = ''">  
    Наведи на мене курсор  
</div>  
  
<input type="text" onchange="console.log('Значення  
змінилось: ' + this.value)">
```

У наведених прикладах ми бачимо три різні випадки використання атрибутів подій. Перший приклад показує найпростіший варіант - при кліку на кнопку виводиться спливаюче повідомлення. Другий приклад демонструє, як можна змінювати стилі елемента при наведенні та зніманні курсора миші, використовуючи ключове слово `this`, яке в контексті обробника події посилається на сам елемент. Третій приклад показує, як можна відслідковувати зміни в полях введення.

Хоча цей підхід простий у використанні та зрозумілий для початківців, він має декілька суттєвих недоліків. По-перше, змішування HTML-розмітки з JavaScript-кодом порушує принцип розділення відповідальностей та ускладнює підтримку коду. По-друге, така організація коду робить складним повторне використання логіки обробки подій. По-третє, атрибутний підхід не дозволяє легко додавати декілька обробників для однієї події на одному елементі.

### 3.3.3 Використання властивостей DOM-об'єкта

Більш гнучким та програмно коректним підходом є призначення обробників подій через властивості DOM-об'єктів безпосередньо в JavaScript-коді. Кожен елемент DOM має властивості, що відповідають різним типам подій, і ми можемо присвоювати цим властивостям функції, які будуть викликатися при настанні відповідних подій.

Назви властивостей для подій збігаються з назвами HTML-атрибутів - onclick, onmouseover, onkeydown тощо. Відмінність полягає в тому, що замість рядка з кодом ми присвоюємо функцію:

```
<button id="myButton">Натисни мене</button>
<div id="colorBox">Наведи курсор</div>
```

```
// Отримуємо посилання на елементи
const button = document.getElementById('myButton');
const colorBox = document.getElementById('colorBox');

// Призначаємо обробник через властивість
button.onclick = function() {
    alert('Кнопка була натиснута!');
};

// Можна використовувати стрілочні функції
```

```
colorBox.onmouseover = () => {
    colorBox.style.backgroundColor = 'lightblue';
};

colorBox.onmouseout = () => {
    colorBox.style.backgroundColor = '';
};
```

Цей підхід має кілька переваг порівняно з атрибутним способом. По-перше, він забезпечує чітке розділення HTML-розмітки та JavaScript-логіки. По-друге, обробники подій можна легко змінювати або видаляти під час виконання програми. По-третє, в обробниках можна використовувати повноцінні функції з локальними змінними та складною логікою.

Однак і цей підхід має обмеження - для кожної властивості події можна призначити лише одну функцію-обробник. Якщо ми спробуємо призначити другий обробник, він просто замінить попередній:

```
button.onclick = function() { console.log('Перший
обробник'); };
button.onclick = function() { console.log('Другий
обробник'); };
// Буде виконуватися тільки другий обробник
```

### 3.3.4 Поширені помилки

При роботі з подіями браузера розробники часто стикаються з типовими помилками, які можуть призвести до неправильної поведінки веб-додатків або до повної їх нефункціональності. Розуміння цих помилок та способів їх уникнення є критично важливим для створення надійних веб-додатків.

Однією з найпоширеніших помилок є спроба призначити обробник події до елемента, який ще не існує в DOM на момент виконання JavaScript-коду. Це трапляється, коли скрипт розміщений у секції `<head>` документа або виконується до повного завантаження сторінки:

```

<!-- НЕПРАВИЛЬНО -->
<head>
  <script>
    const button =
document.getElementById('myButton'); // null
    button.onclick = function() { // Помилка!
      alert('Клік!');
    };
  </script>
</head>
<body>
  <button id="myButton">Кнопка</button>
</body>

```

Для уникнення цієї помилки код призначення обробників слід розмішувати після відповідних HTML-елементів або використовувати події DOMContentLoaded або load:

```

// ПРАВИЛЬНО
document.addEventListener('DOMContentLoaded',
function() {
  const button = document.getElementById('myButton');
  button.onclick = function() {
    alert('Клік!');
  };
});

```

Інша типова помилка пов'язана з контекстом виконання функцій-обробників та некоректним використанням ключового слова `this`. У різних способах призначення обробників `this` може посилатися на різні об'єкти:

```

const obj = {
  name: 'Тестовий об\''ект',
  handleClick: function() {

```

```

        console.log(this.name); // Що буде виведено?
    }
};

button.onclick = obj.handleClick; // this буде
посилатися на button, не на obj

```

Також поширеною є помилка забування про те, що деякі події мають поведінку за замовчуванням, яку іноді потрібно скасовувати. Наприклад, клік на посилання переходить на іншу сторінку, відправка форми перезавантажує сторінку:

```

// Потрібно скасувати поведінку за замовчуванням
document.querySelector('a').onclick = function(event) {
    event.preventDefault(); // Скасовуємо перехід по
посиланню
    // Наша власна логіка
};

```

### 3.3.5 addEventListener

Метод `addEventListener` по праву вважається найбільш потужним та гнучким інструментом для роботи з подіями в сучасній веб-розробці, оскільки він надає розробникам повний контроль над процесом обробки подій та дозволяє створювати складні інтерактивні системи. На відміну від попередніх методів, які мають істотні обмеження, `addEventListener` дозволяє додавати необмежену кількість обробників для однієї події на одному елементі, що відкриває можливості для створення модульних та легко розширюваних додатків. Крім того, цей метод надає точний контроль над фазою обробки події в процесі її поширення по DOM-дереву та забезпечує зручне керування життєвим циклом обробників подій.

Синтаксичну структуру методу `addEventListener` можна представити у наступному вигляді:

```
element.addEventListener(тип_події, функція_обробник,  
опції);
```

Перший параметр цього методу являє собою рядок, який визначає тип події, яку ми хочемо відслідковувати, причому важливо пам'ятати, що назва події записується без традиційного префікса "on", який використовувався в попередніх підходах. Таким чином, замість "onclick" ми пишемо просто "click", замість "onmouseover" - "mouseover", замість "onkeydown" - "keydown" і так далі. Другий параметр представляє собою функцію-обробник, яка буде автоматично викликатися браузером кожного разу, коли на вказаному елементі відбудеться відповідна подія. Третій параметр є необов'язковим і може приймати різні форми - це може бути або простий булев параметр для забезпечення зворотної сумісності зі старими версіями специфікації, або об'єкт з детальними налаштуваннями поведінки обробника події.

Розглянемо детальні приклади використання цього методу в різних ситуаціях:

```
const button = document.getElementById('myButton');

// Найпростіший варіант використання з анонімною  
функцією
button.addEventListener('click', function() {
    console.log('Кнопка була натиснута користувачем!');
});

// Демонстрація можливості додавання декількох  
обробників
button.addEventListener('click', function() {
    console.log('Це перший обробник, який завжди  
виконується');
});
```

```

button.addEventListener('click', function() {
    console.log('Це другий обробник, який виконується
після першого');
});

button.addEventListener('click', function() {
    console.log('А це третій обробник - всі вони
працюють разом');
});

// Використання сучасного синтаксису стрілочних функцій
button.addEventListener('mouseover', () => {
    button.style.backgroundColor = 'lightblue';
    button.style.transition = 'background-color 0.3s
ease';
});

button.addEventListener('mouseout', () => {
    button.style.backgroundColor = '';
});

```

Однією з найбільш значущих переваг методу `addEventListener` порівняно з іншими способами призначення обробників є можливість динамічного видалення раніше доданих обробників за допомогою парного методу `removeEventListener`. Ця функціональність виявляється надзвичайно корисною в ситуаціях, коли потрібно тимчасово деактивувати певну поведінку елемента, або коли обробник повинен спрацювати лише обмежену кількість разів, або коли потрібно очистити пам'ять від непотрібних обробників для уникнення витоків пам'яті. Однак важливо розуміти, що для успішного видалення обробника необхідно передати в `removeEventListener` точно ту саму функцію,

яка була передана в `addEventListener` - браузер порівнює функції за їх посиланням в пам'яті, а не за вмістом:

```
// Правильний спосіб - створюємо іменовану функцію
function clickHandler() {
    console.log('Цей обробник кліку може бути
видалений');
    // Самовидалення після першого спрацювання
    button.removeEventListener('click', clickHandler);
}

// Додаємо обробник
button.addEventListener('click', clickHandler);

// Можна видалити обробник і з зовнішнього коду
// button.removeEventListener('click', clickHandler);
```

Третій параметр методу `addEventListener` відкриває широкі можливості для тонкого налаштування поведінки обробників подій і може приймати дві різні форми залежно від потреб розробника. Якщо передати булеве значення `true`, це буде означати, що обробник повинен працювати в фазі захоплення (`capture phase`) замість стандартної фази спливання (`bubbling phase`) - цей параметр залишений для забезпечення сумісності зі старими версіями специфікації DOM. Однак більш потужним та гнучким є використання об'єкта з налаштуваннями, який може містити декілька корисних властивостей:

```
const specialButton =
document.getElementById('specialButton');

// Обробник, який спрацює тільки один раз
specialButton.addEventListener('click', function() {
    console.log('Цей обробник виконається лише один
раз');
```



```
// Після цього він автоматично видалиться
}, {
  once: true
});

// Пасивний обробник для оптимізації продуктивності
document.addEventListener('scroll', function() {
  console.log('Сторінка прокручується');
  // Браузер знає, що ми не викликатимемо
preventDefault()
  // і може оптимізувати обробку події
}, {
  passive: true
});

// Обробник для фази захоплення
document.body.addEventListener('click', function(event)
{
  console.log('Клік перехоплено в фазі захоплення');
}, {
  capture: true
});

// Комбінування різних опцій
specialButton.addEventListener('dblclick', function() {
  console.log('Подвійний клік з усіма опціями');
}, {
  once: true,      // Спрацює один раз
  passive: false,  // Може використовувати
preventDefault()
```

```
capture: false    // Працює в фазі спливання
});
```

Особливо важливою є опція `passive: true`, яка повідомляє браузеру, що обробник гарантовано не буде викликати метод `preventDefault()` для скасування стандартної поведінки події. Ця інформація дозволяє браузеру виконувати певні оптимізації, особливо для подій прокручування та дотику на мобільних пристроях, що значно покращує відгук інтерфейсу та загальну продуктивність веб-додатка.

### 3.3.6 Об'єкт події

Коли в браузері відбувається будь-яка подія - чи то клік миші, натискання клавіші, завантаження сторінки або зміна розміру вікна - браузер автоматично створює спеціальний об'єкт `Event`, який служить детальним звітом про те, що саме відбулося в конкретний момент часу. Цей об'єкт є своєрідним контейнером, який містить всю необхідну інформацію про подію: від базових деталей, таких як тип події та час її настання, до специфічних характеристик, як-от координати курсора миші або код натиснутої клавіші. Браузер автоматично передає цей об'єкт як перший параметр у всі функції-обробники подій, забезпечуючи розробникам повний доступ до контексту та всіх деталей події, що дозволяє створювати точні та адаптивні реакції на дії користувача.

Архітектура об'єкта події побудована за принципом ієрархії властивостей, де існують універсальні властивості, доступні для абсолютно всіх типів подій незалежно від їх природи, а також спеціалізовані властивості, які з'являються лише для конкретних типів подій і містять специфічну для них інформацію. До найважливіших універсальних властивостей належить `type`, яка містить рядкове представлення типу події (наприклад, `"click"`, `"keydown"`, `"load"`), `target`, яка вказує на DOM-елемент, який безпосередньо став джерелом або ініціатором події, `currentTarget`, яка вказує на елемент, на якому в даний момент обробляється подія (що може відрізнятися від `target` через механізм

поширення подій), та `timeStamp`, яка містить точний час настання події у вигляді числа мілісекунд, що пройшли з моменту створення документа.

```
const button = document.getElementById('myButton');

button.addEventListener('click', function(event) {
    console.log('Тип події:', event.type);           //
    виведе "click"
    console.log('Джерело події:', event.target);     //
    покаже елемент <button>
    console.log('Поточна ціль:', event.currentTarget);
    // також покаже <button>
    console.log('Час події:', event.timeStamp);      //
    число мілісекунд від початку
    console.log('Подія можна скасувати:',
event.cancelable); // true/false
    console.log('Подія спливає:', event.bubbles);    //
    true/false
});
```

Для подій миші, які включають в себе кліки, рухи курсора, натискання та відпускання кнопок миші, об'єкт події збагачується додатковими властивостями, які надають детальну інформацію про стан миші та контекст її використання. Особливо важливими є різні системи координат, кожна з яких має своє специфічне призначення: `clientX` та `clientY` надають координати відносно видимої частини вікна браузера (`viewport`), що корисно для позиціонування елементів інтерфейсу; `pageX` та `pageY` показують координати відносно всієї сторінки з урахуванням прокручування, що важливо для роботи з великими документами; `offsetX` та `offsetY` вказують координати відносно самого елемента-цілі, що дозволяє точно визначити, де саме в межах елемента відбувся клік. Крім координат, об'єкт події містить інформацію про натиснуті кнопки миші через властивість `button` та стан клавіш-модифікаторів:

```

document.addEventListener('click', function(event) {
    console.log('Координати відносно вікна браузера:',
event.clientX, event.clientY);
    console.log('Координати відносно всієї сторінки:',
event.pageX, event.pageY);
    console.log('Координати відносно елемента-цілі:',
event.offsetX, event.offsetY);
    console.log('Координати відносно екрана:',
event.screenX, event.screenY);

    // Інформація про натиснуті кнопки миші
    console.log('Ліва кнопка миші натиснута:',
event.button === 0);
    console.log('Середня кнопка миші натиснута:',
event.button === 1);
    console.log('Права кнопка миші натиснута:',
event.button === 2);

    // Стан клавіш-модифікаторів
    console.log('Клавіша Ctrl натиснута:',
event.ctrlKey);
    console.log('Клавіша Shift натиснута:',
event.shiftKey);
    console.log('Клавіша Alt натиснута:',
event.altKey);
    console.log('Клавіша Meta (Cmd/Win) натиснута:',
event.metaKey);
});

```

Для подій клавіатури, які охоплюють натискання, утримування та відпускання клавіш, об'єкт події надає кілька різних способів ідентифікації

натиснутих клавіш, кожен з яких має свої переваги та область застосування. Найсучасніша та рекомендована властивість `key` повертає фактичний символ або назву клавіші з урахуванням поточної розкладки клавіатури та стану клавіш-модифікаторів, що робить її ідеальною для обробки текстового введення. Властивість `code` надає фізичний код клавіші незалежно від розкладки клавіатури, що корисно для ігрових додатків або коли важлива фізична позиція клавіші. Застаріла властивість `keyCode` залишається для підтримки старого коду, але не рекомендується для використання в нових проектах через її непослідовність та залежність від браузера:

```
document.addEventListener('keydown', function(event) {
    console.log('Символ або назва клавіші:',
event.key);          // "a", "A", "Enter", "Escape"
    console.log('Фізичний код клавіші:', event.code);
// "KeyA", "Enter", "Space", "Escape"
    console.log('Числовий код (застарілий):',
event.keyCode);      // 65, 13, 32, 27

    // Практичні приклади обробки різних клавіш
    if (event.key === 'Escape') {
        console.log('Користувач натиснув Escape - можна
закрити модальне вікно');
    }

    if (event.key === 'Enter') {
        console.log('Користувач натиснув Enter - можна
підтвердити введення');
    }

    if (event.ctrlKey && event.key === 's') {
```

```

        event.preventDefault(); // Скасовуємо
стандартне збереження браузера
        console.log('Користувач натиснув Ctrl+S - можна
зберегти документ');
    }

    // Перевірка на спеціальні клавіші
    if (event.code.startsWith('Arrow')) {
        console.log('Натиснута клавіша стрілки:',
event.code);
    }
});

```

Об'єкт події не тільки надає інформацію про подію, але й забезпечує потужні методи для контролю її поведінки та поширення в DOM-дереві. Найважливішим з цих методів є `preventDefault()`, який дозволяє повністю скасувати стандартну поведінку браузера для конкретної події, надаючи розробнику повний контроль над тим, що має відбутися. Цей метод виявляється особливо корисним при роботі з формами, посиланнями, контекстними меню та іншими елементами, які мають вбудовану поведінку браузера. Важливо розуміти, що не всі події можна скасувати - це залежить від значення властивості `cancelable` об'єкта події:

```

// Повний контроль над відправкою форми
const form = document.getElementById('myForm');
form.addEventListener('submit', function(event) {
    event.preventDefault(); // Скасовуємо стандартну
відправку форми

    // Тепер можемо виконати власну валідацію
    const formData = new FormData(form);
    if (validateForm(formData)) {

```

```

        console.log('Форма валідна, відправляємо через
AJAX');
        sendFormWithAjax(formData);
    } else {
        console.log('Форма містить помилки, показуємо
повідомлення користувачу');
        showValidationErrors();
    }
});

// Скасування стандартної поведінки посилань
document.querySelectorAll('a.custom-link').forEach(link
=> {
    link.addEventListener('click', function(event) {
        event.preventDefault(); // Скасовуємо перехід
за посиланням

        // Власна логіка обробки кліку по посиланню
        const url = this.getAttribute('href');
        console.log('Користувач клікнув на посилання:',
url);
        loadContentWithAjax(url);
    });
});

// Скасування контекстного меню для створення власного
document.addEventListener('contextmenu',
function(event) {
    event.preventDefault(); // Скасовуємо стандартне
контекстне меню

```

```
showCustomContextMenu(event.pageX, event.pageY);
});
```

Другий важливий метод об'єкта події - `stopPropagation()` - контролює механізм поширення події по DOM-дереву, дозволяючи зупинити її подальший шлях від дочірніх елементів до батьківських або навпаки. Цей метод виявляється критично важливим в ситуаціях, коли у вас є вкладені елементи з власними обробниками подій, і ви хочете запобігти небажаному спрацюванню обробників батьківських елементів. Розуміння та правильне використання цього методу є ключовим для створення складних інтерфейсів з множинними рівнями взаємодії:

```
// HTML структура: <div class="parent"><div
class="child">Дочірній елемент</div></div>

const parentElement =
document.querySelector('.parent');
const childElement = document.querySelector('.child');

parentElement.addEventListener('click', function(event)
{
    console.log('Клік на батьківському елементі');
});

childElement.addEventListener('click', function(event)
{
    console.log('Клік на дочірньому елементі');

    // Зупиняємо поширення події, щоб не спрацював
    обробник батьківського елемента
    event.stopPropagation();
```



```

    console.log('Обробник батьківського елемента не
    виконається');
  });

  // Альтернативний метод для повної зупинки події
  // event.stopImmediatePropagation(); // зупиняє навіть
  інші обробники на тому ж елементі

```

### 3.3.7 Об'єкт-обробник: `handleEvent`

Окрім традиційного підходу з використанням звичайних функцій в якості обробників подій, сучасна специфікація DOM надає альтернативний та надзвичайно потужний механізм, який полягає у використанні об'єктів, що мають спеціальний метод `handleEvent`. Цей підхід відкриває перед розробниками нові горизонти в організації коду та виявляється особливо корисним в контексті об'єктно-орієнтованого програмування, коли виникає необхідність зберегти контекст конкретного об'єкта, інкапсулювати складну логіку обробки подій, та організувати централізовану систему управління різноманітними типами подій в межах одного класу або об'єкта-контролера.

Механізм роботи цього підходу базується на інтелектуальній поведінці браузера при виклику обробників подій: коли настає момент виконання обробника події, браузер спочатку перевіряє, чи є переданий обробник звичайною функцією, і якщо так, то просто викликає її з передачею об'єкта події як першого параметра. Однак якщо переданий обробник не є функцією, браузер автоматично шукає в цьому об'єкті метод з назвою `handleEvent` і викликає саме його, передаючи об'єкт події як параметр, що дозволяє об'єкту самостійно визначити логіку обробки події на основі її типу, джерела або будь-яких інших характеристик.

Найпростіший приклад використання об'єкта-обробника демонструє базові принципи цього підходу:

```

const simpleHandler = {

```

```

    clickCount: 0,

    handleEvent(event) {
        console.log(`Обробляється подія типу:
${event.type}`);

        if (event.type === 'click') {
            this.clickCount++;
            console.log(`Кількість кліків:
${this.clickCount}`);
            console.log(`Елемент, який було
натиснуто:`, event.target.tagName);
        }
    }
};

// Використання об'єкта-обробника замість функції
const button = document.getElementById('myButton');
button.addEventListener('click', simpleHandler);

```

Цей підхід стає особливо потужним при створенні складних систем обробки подій, де один об'єкт може відповідати за управління множинними типами подій та елементами інтерфейсу. Розглянемо більш розвинений приклад, який демонструє організацію централізованого контролера подій:

```

const advancedEventController = {
    // Внутрішній стан об'єкта
    elements: new Map(),
    statistics: {
        totalClicks: 0,
        totalHovers: 0,
        totalKeyPresses: 0
    }
};

```

```
    },  
  
    // Основний метод обробки всіх подій  
    handleEvent(event) {  
        // Оновлюємо загальну статистику  
        this.updateStatistics(event.type);  
  
        // Делегуємо обробку конкретним методам залежно  
        від типу події  
        switch(event.type) {  
            case 'click':  
                this.handleClickEvent(event);  
                break;  
            case 'mouseover':  
                this.handleMouseEvent(event);  
                break;  
            case 'mouseout':  
                this.handleMouseOutEvent(event);  
                break;  
            case 'keydown':  
                this.handleKeyDownEvent(event);  
                break;  
            case 'focus':  
                this.handleFocusEvent(event);  
                break;  
            case 'blur':  
                this.handleBlurEvent(event);  
                break;  
            default:
```

```

        console.log(`Необроблена подія типу:
${event.type}`);
    }

    // Логування для відлагодження
    console.log(`Подія ${event.type} оброблена
об'єктом-контролером`);
},

// Методи для обробки конкретних типів подій
handleClickEvent(event) {
    const element = event.target;
    console.log(`Клік по елементу:
${element.tagName}#${element.id}`);

    // Додавання візуального ефекту кліку
    element.style.transform = 'scale(0.95)';
    setTimeout(() => {
        element.style.transform = 'scale(1)';
    }, 150);

    // Спеціальна логіка для кнопок
    if (element.tagName === 'BUTTON') {
        this.handleButtonClick(element, event);
    }
},

handleMouseOverEvent(event) {
    const element = event.target;
    element.style.backgroundColor = '#f0f8ff';

```

```

        element.style.transition = 'background-color
0.3s ease';

        console.log(`Курсор наведено на елемент:
${element.tagName}`);
    },

    handleMouseOutEvent(event) {
        const element = event.target;
        element.style.backgroundColor = '';
        console.log(`Курсор знято з елемента:
${element.tagName}`);
    },

    handleKeyDownEvent(event) {
        console.log(`Натиснута клавіша: ${event.key}
(код: ${event.code})`);

        // Обробка гарячих клавіш
        if (event.ctrlKey && event.key === 's') {
            event.preventDefault();
            this.saveCurrentState();
        } else if (event.key === 'Escape') {
            this.closeAllModals();
        }
    },

    handleFocusEvent(event) {
        console.log(`Фокус встановлено на елементі:
${event.target.tagName}`);
    }
}

```

```

        event.target.style.outline = '2px solid
#4CAF50';
    },

    handleBlurEvent(event) {
        console.log(`Фокус знято з елемента:
${event.target.tagName}`);
        event.target.style.outline = '';
    },

    // Допоміжні методи
    updateStatistics(eventType) {
        switch(eventType) {
            case 'click':
                this.statistics.totalClicks++;
                break;
            case 'mouseover':
                this.statistics.totalHovers++;
                break;
            case 'keydown':
                this.statistics.totalKeyPresses++;
                break;
        }
    },

    handleButtonClick(button, event) {
        const action = button.dataset.action;
        if (action) {
            console.log(`Виконується дія: ${action}`);
            this.executeAction(action, button, event);
        }
    }
}

```

```

    }

    },

    executeAction(actionName, element, event) {
        // Реалізація конкретних дій на основі data-
        атрибутів
        switch(actionName) {
            case 'save':
                console.log('Збереження даних...');
                break;
            case 'delete':
                console.log('Видалення елемента...');
                break;
            case 'edit':
                console.log('Редагування елемента...');
                break;
        }
    },

    saveCurrentState() {
        console.log('Збереження поточного стану
        додатка');
        console.log('Статистика:', this.statistics);
    },

    closeAllModals() {
        console.log('Закриття всіх модальних вікон');

        document.querySelectorAll('.modal.open').forEach(modal
        => {

```

```

        modal.classList.remove('open');
    });
},

// Метод для ініціалізації контролера
init() {
    // Додаємо обробники до різних елементів

document.querySelectorAll('button').forEach(button => {
    button.addEventListener('click', this);
    button.addEventListener('mouseover', this);
    button.addEventListener('mouseout', this);
});

    document.querySelectorAll('input,
textarea').forEach(input => {
        input.addEventListener('focus', this);
        input.addEventListener('blur', this);
    });

    document.addEventListener('keydown', this);

    console.log('Контролер подій ініціалізовано');
},

// Метод для очищення обробників
destroy() {

document.querySelectorAll('button').forEach(button => {
    button.removeEventListener('click', this);

```



```

        button.removeEventListener('mouseover',
this);

        button.removeEventListener('mouseout',
this);

    });

    document.querySelectorAll('input,
textarea').forEach(input => {
        input.removeEventListener('focus', this);
        input.removeEventListener('blur', this);
    });

    document.removeEventListener('keydown', this);

    console.log('Контролер подій очищено');
}
};

// Ініціалізація контролера
advancedEventController.init();

```

Особливо елегантним цей підхід стає при використанні сучасного синтаксису ES6 класів, який дозволяє створювати ще більш структуровані та повторно використовувані компоненти для обробки подій:

```

class InteractiveComponent {
    constructor(element) {
        this.element = element;
        this.state = {
            isActive: false,
            interactionCount: 0,
            lastInteractionTime: null

```

```

    };

    // Автоматична ініціалізація обробників подій
    this.attachEventListeners();
  }

  // Центральний метод обробки всіх подій компонента
  handleEvent(event) {
    // Оновлюємо час останньої взаємодії
    this.state.lastInteractionTime =
event.timeStamp;
    this.state.interactionCount++;

    // Логування для відлагодження
    console.log(`Компонент обробляє подію:
${event.type}`);
    console.log(`Загальна кількість взаємодій:
${this.state.interactionCount}`);

    // Делегування обробки конкретним методам
    const methodName =
`handle${this.capitalizeFirstLetter(event.type)}`;
    if (typeof this[methodName] === 'function') {
      this[methodName](event);
    } else {
      console.warn(`Метод ${methodName} не
знайдено для обробки події ${event.type}`);
    }

    // Виклик callback'ів, якщо вони визначені

```

```
        this.triggerCallbacks(event.type, event);
    }

    // Методи для обробки конкретних подій
    handleClick(event) {
        this.toggleActiveState();
        console.log(`Стан активності змінено на:
${this.state.isActive}`);

        // Додавання візуальних ефектів
        this.element.classList.toggle('active',
this.state.isActive);
    }

    handleMouseover(event) {
        this.element.classList.add('hovered');
        console.log('Компонент в стані hover');
    }

    handleMouseout(event) {
        this.element.classList.remove('hovered');
        console.log('Компонент вийшов зі стану hover');
    }

    handleFocus(event) {
        this.element.classList.add('focused');
        console.log('Компонент отримав фокус');
    }

    handleBlur(event) {
```

```

        this.element.classList.remove('focused');
        console.log('Компонент втратив фокус');
    }

    // Методи управління станом
    toggleActiveState() {
        this.state.isActive = !this.state.isActive;
        return this.state.isActive;
    }

    getState() {
        return { ...this.state }; // Повертаємо копію
        стану
    }

    // Методи управління подіями
    attachEventListeners() {
        const events = ['click', 'mouseover',
        'mouseout', 'focus', 'blur'];
        events.forEach(eventType => {
            this.element.addEventListener(eventType,
            this);
        });
        console.log('Обробники подій приєднано до
        компонента');
    }

    detachEventListeners() {
        const events = ['click', 'mouseover',
        'mouseout', 'focus', 'blur'];

```

```

        events.forEach(eventType => {
            this.element.removeEventListener(eventType,
this);

        });

        console.log('Обробники подій від\'єднано від
компонента');
    }

    // Система callback'ів для розширюваності
    triggerCallbacks(eventType, event) {
        const callbackProperty =
`on${this.capitalizeFirstLetter(eventType)}`;
        if (typeof this[callbackProperty] ===
'function') {
            this[callbackProperty](event, this.state);
        }
    }

    // Допоміжні методи
    capitalizeFirstLetter(string) {
        return string.charAt(0).toUpperCase() +
string.slice(1);
    }

    // Метод для очищення ресурсів
    destroy() {
        this.detachEventListeners();
        this.state = null;
        console.log('Компонент знищено');
    }

```

```

}

// Приклад використання класу
const interactiveElement =
document.getElementById('myInteractiveElement');
const component = new
InteractiveComponent(interactiveElement);

// Додавання власних callback'ів
component.onClick = function(event, state) {
    console.log('Власний callback для кліку:', state);
};

component.onMouseover = function(event, state) {
    console.log('Власний callback для hover:', state);
};

```

Головними перевагами підходу з об'єктами-обробниками є збереження контексту виконання (ключове слово `this` завжди вказує на наш об'єкт), можливість інкапсуляції складної логіки в межах одного об'єкта, легкість додавання нової функціональності через наслідування або композицію, централізоване управління множинними типами подій, та значне спрощення процесу тестування коду завдяки чіткій структурі та передбачуваний поведінці.

### 3.4 Форми та інтерактивні елементи

Форми являють собою один з найважливіших механізмів взаємодії користувача з веб-додатками, оскільки саме через них користувачі можуть надсилати дані на сервер, здійснювати пошук, реєструватися, авторизуватися та виконувати безліч інших дій. У цьому розділі ми детально розглянемо, як JavaScript дозволяє нам працювати з формами на клієнтській стороні, керувати

їхньою поведінкою, валідувати введені дані та створювати інтерактивний користувацький досвід.

Сучасні веб-додатки вимагають складної логіки обробки форм, яка включає в себе динамічну валідацію, автоматичне збереження даних, інтерактивні підказки та багато інших функцій, які неможливо реалізувати лише за допомогою HTML та CSS.

### 3.4.1 Властивості та методи форми

Коли ми говоримо про роботу з формами в JavaScript, то перш за все необхідно розуміти, що кожна HTML-форма на веб-сторінці автоматично перетворюється браузером на спеціальний об'єкт під назвою `HTMLFormElement`. Цей об'єкт є досить складною структурою даних, яка містить у собі не лише всі звичайні властивості та методи, притаманні будь-якому HTML-елементу (такі як `innerHTML`, `style`, `className` тощо), але й надає додаткові, специфічні можливості для роботи саме з формами.

Найбільш фундаментальною та корисною властивістю форми є `elements`. Ця властивість містить у собі живу колекцію всіх елементів управління, які знаходяться всередині форми. Під елементами управління ми розуміємо різноманітні поля для введення тексту, кнопки відправлення та скидання, прапорці для позначення опцій, радіо-кнопки для вибору одного варіанта з декількох, випадаючі списки, текстові області та багато інших інтерактивних елементів. Колекція є "живою" в тому сенсі, що якщо ми динамічно додамо або видалимо якийсь елемент з форми за допомогою JavaScript, то колекція `elements` автоматично оновиться та буде відображати актуальний стан форми.

```
const form = document.getElementById('userForm');

// Отримуємо доступ до всіх елементів форми
console.log(form.elements); //
HTMLFormControlsCollection
```

```
// Можемо звернутися до конкретного елемента за його
індексом
const firstElement = form.elements[0];
console.log('Перший елемент:', firstElement.name);

// Або за його ім'ям (атрибут name)
const usernameField = form.elements['username'];
console.log('Поле імені:', usernameField.value);
```

Тісно пов'язаною з `elements` є властивість `length`, яка повертає числове значення кількості елементів управління в формі. Ця властивість особливо корисна тоді, коли нам потрібно програмно пройти по всіх елементах форми, наприклад, для виконання масової валідації або збирання всіх введених користувачем даних в один об'єкт.

```
// Простий перебір всіх елементів
for (let i = 0; i < form.length; i++) {
    const element = form.elements[i];
    if (element.type !== 'submit') {
        console.log(`Поле "${element.name}" має
значення: "${element.value}"`);
    }
}
```

Наступною групою важливих властивостей є ті, що безпосередньо відповідають HTML-атрибутам форми та визначають, як саме буде відбуватися процес надсилання даних на сервер. Властивість `action` зберігає URL-адресу, на яку будуть відправлені дані форми після її надсилання. Якщо цей атрибут не вказано в HTML, то дані будуть відправлені на ту саму сторінку, з якої відбувається надсилання.

Властивість `method` визначає HTTP-метод, який буде використовуватися для передачі даних. Найчастіше використовуються два методи: GET для



отримання даних (при цьому параметри форми додаються до URL-адреси) та POST для відправлення даних (параметри передаються в тілі HTTP-запиту). POST-метод є більш безпечним та підходящим для передачі конфіденційної інформації або великих обсягів даних.

```
const loginForm = document.getElementById('loginForm');

// Можемо змінювати ці властивості динамічно
loginForm.action = '/api/user/login';
loginForm.method = 'POST';

// Перевіримо, які значення встановлені
console.log(`Дані будуть відправлені на:
${loginForm.action}`);
console.log(`Використовуючи метод:
${loginForm.method}`);
```

Властивість `enctype` (скорочено від "encoding type") визначає спосіб кодування даних форми перед їх відправленням на сервер. За замовчуванням використовується значення `"application/x-www-form-urlencoded"`, яке підходить для звичайних текстових даних. Однак, якщо форма містить поля для завантаження файлів, необхідно встановити значення `"multipart/form-data"`, що дозволяє передавати як текстові дані, так і файли в одному запиті.

```
const fileUploadForm =
document.getElementById('uploadForm');

// Для завантаження файлів обов'язково потрібно
встановити правильний enctype
fileUploadForm.enctype = 'multipart/form-data';
fileUploadForm.method = 'POST';
```

Переходячи до методів форми, найважливішими є `submit()` та `reset()`. Метод `submit()` дозволяє програмно ініціювати процес надсилання форми, що

може бути корисним у багатьох сценаріях. Наприклад, ми можемо автоматично відправити форму після того, як користувач заповнив усі обов'язкові поля, або після успішної валідації даних на стороні клієнта. Також цей метод може використовуватися для створення багатоступеневих форм, де завершення одного кроку автоматично веде до наступного.

```
const form = document.getElementById('wizardForm');
const nextButton = document.getElementById('nextStep');

nextButton.addEventListener('click', function() {
    // Перевіряємо, чи заповнені всі поля поточного
    кроку
    const currentStepFields =
form.querySelectorAll('.current-step input');
    let allFilled = true;

    currentStepFields.forEach(field => {
        if (!field.value.trim()) {
            allFilled = false;
        }
    });

    if (allFilled) {
        // Якщо все заповнено, програмно відправляємо
        форму
        form.submit();
    } else {
        alert('Будь ласка, заповніть всі обов\'язкові
поля');
    }
});
```

Метод `reset()` виконує протилежну функцію - він скидає всі поля форми до їхніх початкових значень, які були встановлені в HTML-кодi або програмно до першого заповнення користувачем. Цей метод особливо корисний для створення кнопок "Очистити форму" або "Почати спочатку", а також може використовуватися для автоматичного очищення форми після успішного надсилання даних.

```
const contactForm =
document.getElementById('contactForm');
const clearButton =
document.getElementById('clearForm');

clearButton.addEventListener('click', function() {
    // Запитуємо підтвердження перед очищенням
    const confirmClear = confirm('Це дія видалить всі
введені дані. Продовжити?');

    if (confirmClear) {
        contactForm.reset();
        console.log('Форму очищено');
    }
});
```

Дуже важливою особливістю, яку необхідно розуміти при роботі з методом `submit()`, є те, що його програмний виклик не генерує події "submit". Це означає, що якщо у вашому кодi є обробники події `submit`, які виконують додаткову валідацію або логування, вони не спрацюють при використанні `form.submit()`. Натомість, подія `submit` генерується лише тоді, коли користувач натискає на кнопку типу "submit" або натискає клавішу Enter, перебуваючи в одному з полів форми.

```
const form = document.getElementById('exampleForm');
```

```
// Цей обробник НЕ спрацює при виклику form.submit()
form.addEventListener('submit', function(event) {
    console.log('Форма надсилається через подію submit');

    // Тут могла б бути валідація
});

const submitButton = document.getElementById('submit');
const programSubmitButton =
document.getElementById('programSubmit');

// Цей спосіб ВИКЛИЧЕ подію submit
submitButton.addEventListener('click', function() {
    // Браузер автоматично згенерує подію submit
});

// А цей спосіб НЕ викличе подію submit
programSubmitButton.addEventListener('click',
function() {
    form.submit(); // Пряме надсилення без події
});
```

Ця особливість має важливі практичні наслідки. Якщо вам потрібно, щоб при програмному надсиленні також виконувалася вся логіка валідації та обробки, яка зазвичай відбувається в обробнику події submit, то варто або винести цю логіку в окрему функцію і викликати її перед form.submit(), або програмно створити та відправити подію submit.

```
// Правильний підхід – винесення логіки в окрему
функцію
function processFormSubmission() {
    console.log('Виконуємо валідацію...');
```

```

    // Логіка валідації
    return true; // або false, якщо валідація не
    пройшла
}

// Обробник натискання кнопки submit
form.addEventListener('submit', function(event) {
    if (!processFormSubmission()) {
        event.preventDefault(); // Зупиняємо
        надсилання, якщо є помилки
    }
});

// Програмне надсилання з тією самою логікою
function submitProgrammatically() {
    if (processFormSubmission()) {
        form.submit();
    }
}

```

### 3.4.2 Навігація: форми та елементи

Навігація по формах та їхніх елементах є критично важливою для створення доступних та зручних користувацьких інтерфейсів. JavaScript надає декілька способів отримання доступу до форм та їхніх елементів, кожен з яких має свої переваги залежно від конкретної ситуації.

#### Доступ до форм в документі

Документ надає кілька способів отримання доступу до форм. Колекція `document.forms` містить всі форми документа і дозволяє отримувати доступ до них за індексом, ім'ям або ідентифікатором.

```

// Отримання всіх форм документа

```

```
console.log('Кількість форм у документі:',  
document.forms.length);  
  
// Доступ до форми за індексом  
const firstForm = document.forms[0];  
  
// Доступ до форми за атрибутом name  
const loginForm = document.forms['loginForm'];  
// або  
const loginForm2 = document.forms.loginForm;  
  
// Доступ до форми за id (якщо немає атрибуту name)  
const registrationForm =  
document.forms['registrationForm'];
```

### **Доступ до елементів форми**

Після отримання посилання на форму можна отримати доступ до її елементів кількома способами. Найбільш універсальним є використання колекції `elements`, яка містить всі елементи управління форми.

```
const form = document.forms.userForm;  
  
// Доступ до елементів через колекцію elements  
const usernameField = form.elements['username'];  
const emailField = form.elements.email;  
const passwordField = form.elements[2]; // за індексом  
  
// Альтернативний спосіб - прямий доступ через ім'я  
const usernameField2 = form.username;  
const emailField2 = form.email;
```

```
console.log('Поточне значення імені користувача:',
usernameField.value);
```

Особливо корисною є можливість роботи з групами елементів, які мають однакове ім'я, наприклад, радіо-кнопки або прапорці в групі.

```
// HTML: <input type="radio" name="gender"
value="male">
//      <input type="radio" name="gender"
value="female">
const form = document.forms.profileForm;
const genderRadios = form.elements.gender;

// genderRadios буде NodeList з усіма радіо-кнопками з
name="gender"
for (let radio of genderRadios) {
    if (radio.checked) {
        console.log('Обрана стать:', radio.value);
        break;
    }
}
```

### Навігація між елементами

Для покращення користувацького досвіду важливо забезпечити логічну навігацію між полями форми за допомогою клавіші Tab. JavaScript дозволяє керувати цим процесом програмно.

```
const form = document.forms.orderForm;

// Встановлення фокуса на першому полі при завантаженні
сторінки
window.addEventListener('load', function() {
    form.elements[0].focus();
});
```

```

// Автоматичний перехід до наступного поля після
заповнення
form.elements.cardNumber.addEventListener('input',
function(event) {
    if (event.target.value.length === 16) {
        // Переходимо до поля з датою закінчення
терміну дії картки
        form.elements.expiryDate.focus();
    }
});

// Обробка натискання клавіш для навігації
form.addEventListener('keydown', function(event) {
    if (event.key === 'Enter' && event.target.type !==
'textarea') {
        event.preventDefault();
        // Знаходимо наступний елемент для фокуса
        const currentIndex =
Array.from(form.elements).indexOf(event.target);
        const nextElement = form.elements[currentIndex
+ 1];
        if (nextElement) {
            nextElement.focus();
        }
    }
});

```



### 3.4.3 Зворотне посилання: `element.form`

Кожен елемент форми має властивість `form`, яка містить посилання на батьківську форму. Ця властивість є надзвичайно корисною, оскільки дозволяє отримати доступ до форми та інших її елементів з будь-якого елемента, не знаючи структуру DOM заздалегідь.

Зворотне посилання особливо корисне при створенні обробників подій, коли потрібно отримати доступ до інших елементів форми або до самої форми для виконання валідації чи інших операцій.

```
// Припустимо, у нас є поле для вводу електронної пошти
const emailInput = document.getElementById('email');

emailInput.addEventListener('blur', function() {
    // Отримуємо посилання на форму через властивість
    form
    const form = this.form;
    const confirmEmailInput =
form.elements.confirmEmail;

    if (confirmEmailInput.value && this.value !==
confirmEmailInput.value) {
        showError('Електронні адреси не співпадають');
        // Можемо також заблокувати кнопку надсилання
        form.elements.submitButton.disabled = true;
    } else {
        clearError();
        form.elements.submitButton.disabled = false;
    }
});
```

```
function showError(message) {
    const errorDiv = document.getElementById('error-
message');
    errorDiv.textContent = message;
    errorDiv.style.display = 'block';
}

function clearError() {
    const errorDiv = document.getElementById('error-
message');
    errorDiv.style.display = 'none';
}
```

Зворотнє посилання дозволяє створювати універсальні функції валідації, які можуть працювати з різними формами та елементами.

```
function validateField(element) {
    const form = element.form;
    const fieldName = element.name;
    let isValid = true;
    let errorMessage = '';

    // Загальні правила валідації
    if (element.required && !element.value.trim()) {
        isValid = false;
        errorMessage = `Поле "${fieldName}" є
обов'язковим для заповнення`;
    }

    // Специфічна валідація для різних типів полів
    switch (element.type) {
        case 'email':
```

```
        if (element.value &&
!isValidEmail(element.value)) {
            isValid = false;
            errorMessage = 'Введіть коректну
електронну адресу';
        }
        break;
    case 'password':
        if (element.value.length > 0 &&
element.value.length < 8) {
            isValid = false;
            errorMessage = 'Пароль повинен містити
мінімум 8 символів';
        }
        // Перевіряємо підтвердження пароля, якщо
воно є
        const confirmPassword =
form.elements.confirmPassword;
        if (confirmPassword &&
confirmPassword.value !== element.value) {
            isValid = false;
            errorMessage = 'Паролі не співпадають';
        }
        break;
    }

    // Відображаємо результат валідації
    updateFieldValidation(element, isValid,
errorMessage);
    return isValid;
}
```

```

}

function updateFieldValidation(element, isValid,
errorMessage) {
    const errorSpan =
element.parentNode.querySelector('.error-message');
    if (errorSpan) {
        errorSpan.textContent = isValid ? '' :
errorMessage;
    }
    element.style.borderColor = isValid ? '#ccc' :
'#ff0000';
}

function isValidEmail(email) {
    const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
    return emailRegex.test(email);
}

// Додаємо обробники до всіх полів форми
document.addEventListener('DOMContentLoaded',
function() {
    const forms = document.forms;
    for (let form of forms) {
        for (let element of form.elements) {
            if (element.type !== 'submit' &&
element.type !== 'button') {
                element.addEventListener('blur',
function() {
                    validateField(this);

```

```

    } ) ;

    }

    }

    } ) ;

```

### 3.4.4 Елементи форми

Коли ми говоримо про елементи форм у веб-розробці, то маємо на увазі різноманітні компоненти, через які користувачі можуть взаємодіяти з нашим додатком та передавати інформацію. Кожен тип елемента має свої унікальні особливості, властивості та методи, які дозволяють нам ефективно керувати поведінкою форми та забезпечувати зручний користувацький досвід.

#### Розуміння елементів форми

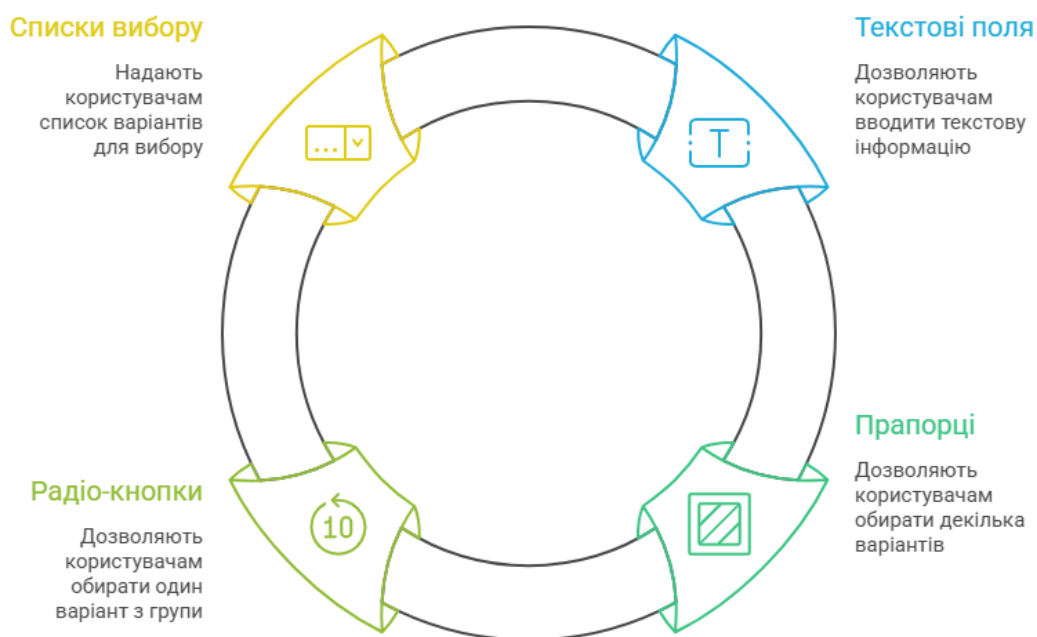


Рисунок 3.1 — Елементи форми

#### Текстові поля - основа взаємодії з користувачем

Текстові поля є найпоширенішими елементами форм і включають в себе звичайні текстові поля (`input type="text"`), поля для електронної пошти (`input type="email"`), поля для паролів (`input type="password"`) та багаторядкові

текстові області (textarea). Ці елементи надають нам потужні можливості для роботи з текстовою інформацією.

Основною властивістю текстових полів є value, яка дозволяє нам отримувати поточне значення поля або встановлювати нове значення програмно. Наприклад, якщо ми маємо поле для введення імені користувача, то можемо легко отримати введену інформацію:

```
const nameInput = document.getElementById('userName');  
console.log('Користувач ввів:', nameInput.value);  
nameInput.value = 'Іван Петренко'; // Встановлюємо  
значення програмно
```

Особливо цікавими є властивості selectionStart та selectionEnd, які показують позицію початку та кінця виділеного тексту в полі. Це дозволяє нам створювати розумні текстові редактори прямо у браузері. Припустімо, ми хочемо вставити текст у позицію курсору:

```
const textArea = document.getElementById('comment');  
const cursorPosition = textArea.selectionStart;  
const textBefore = textArea.value.substring(0,  
cursorPosition);  
const textAfter =  
textArea.value.substring(cursorPosition);  
textArea.value = textBefore + '[вставлений текст]' +  
textAfter;
```

### **Прапорці та радіо-кнопки - елементи вибору**

Прапорці (checkbox) та радіо-кнопки (radio) представляють різні типи вибору для користувача. Прапорці дозволяють обирати декілька варіантів одночасно, тоді як радіо-кнопки працюють у групах, де можна обрати лише один варіант з усієї групи.

Головною властивістю цих елементів є checked, яка повертає true або false залежно від того, чи обрано елемент. Для прапорців ми можемо легко перевіряти стан кожного окремо:

```
const subscribeCheckbox =
document.getElementById('newsletter');
if (subscribeCheckbox.checked) {
    console.log('Користувач хоче отримувати розсилку');
}
```

З радіо-кнопками ситуація трохи складніша, оскільки вони об'єднані в групи за атрибутом name. Щоб знайти, яка кнопка обрана в групі, нам потрібно перебрати всі кнопки з однаковим ім'ям:

```
const genderRadios =
document.querySelectorAll('input[name="gender"]');
for (let radio of genderRadios) {
    if (radio.checked) {
        console.log('Обрана стать:', radio.value);
        break;
    }
}
```

### Списки для вибору - елементи select

Елементи select надають користувачеві можливість обирати з попередньо визначеного списку варіантів. Ці елементи мають багато корисних властивостей, серед яких selectedIndex (індекс обраного елемента), options (колекція всіх варіантів) та value (значення обраного варіанта).

Простий приклад роботи зі списком виглядає так:

```
const citySelect = document.getElementById('city');
console.log('Обране місто:', citySelect.value);
console.log('Індекс обраного варіанта:',
citySelect.selectedIndex);
```

Особливо корисними є каскадні списки, коли вибір в одному списку впливає на варіанти в іншому. Наприклад, спочатку користувач обирає країну, а потім йому показуються міста цієї країни:

```

const countrySelect =
document.getElementById('country');
const citySelect = document.getElementById('city');

countrySelect.addEventListener('change', function() {
    // Очищаємо список міст
    citySelect.innerHTML = '<option>Оберіть
місто</option>';

    // Додаємо міста відповідно до обраної країни
    if (this.value === 'ukraine') {
        citySelect.innerHTML += '<option
value="kyiv">Київ</option>';
        citySelect.innerHTML += '<option
value="lviv">Львів</option>';
    }
});

```

Списки також можуть підтримувати множинний вибір, коли властивість `multiple` встановлена в `true`. У такому випадку користувач може обрати декілька варіантів одночасно, утримуючи клавішу `Ctrl` або `Cmd`:

```

const skillsSelect = document.getElementById('skills');
skillsSelect.multiple = true;

// Отримуємо всі обрані навички
function getSelectedSkills() {
    const selected = [];
    for (let option of skillsSelect.options) {
        if (option.selected) {
            selected.push(option.value);
        }
    }
}

```



```

    }
    return selected;
}

```

### **Практичні аспекти роботи з елементами**

У реальних проектах часто доводиться працювати з комбінаціями різних типів елементів, створюючи складні інтерактивні форми. Наприклад, форма реєстрації може містити текстові поля для імені та електронної пошти, радіо-кнопки для вибору статі, прапорці для згоди з умовами користування та список для вибору країни.

Важливо розуміти, що кожен тип елемента має свої особливості валідації та обробки даних. Текстові поля потребують перевірки на порожність та відповідність певним шаблонам, прапорці та радіо-кнопки - перевірки стану вибору, а списки - перевірки того, чи обрано коректний варіант.

Також слід пам'ятати про доступність форм - всі елементи повинні мати відповідні мітки (label), бути доступними для навігації за допомогою клавіатури та надавати зрозумілі повідомлення про помилки валідації.

Розуміння специфіки кожного типу елементів форми дозволяє створювати інтуїтивно зрозумілі та функціональні інтерфейси, які забезпечують позитивний досвід взаємодії користувача з веб-додатком.

#### **3.4.5 Події форм - реагування на дії користувача в реальному часі**

Робота з подіями форм є надзвичайно важливою частиною створення сучасних інтерактивних веб-додатків, оскільки саме події дозволяють нам миттєво реагувати на дії користувача та забезпечувати зручний, responsive користувацький досвід. Кожна подія має свої особливості спрацьовування та призначення, що робить їх незамінними інструментами для різних сценаріїв взаємодії.

## Події форм

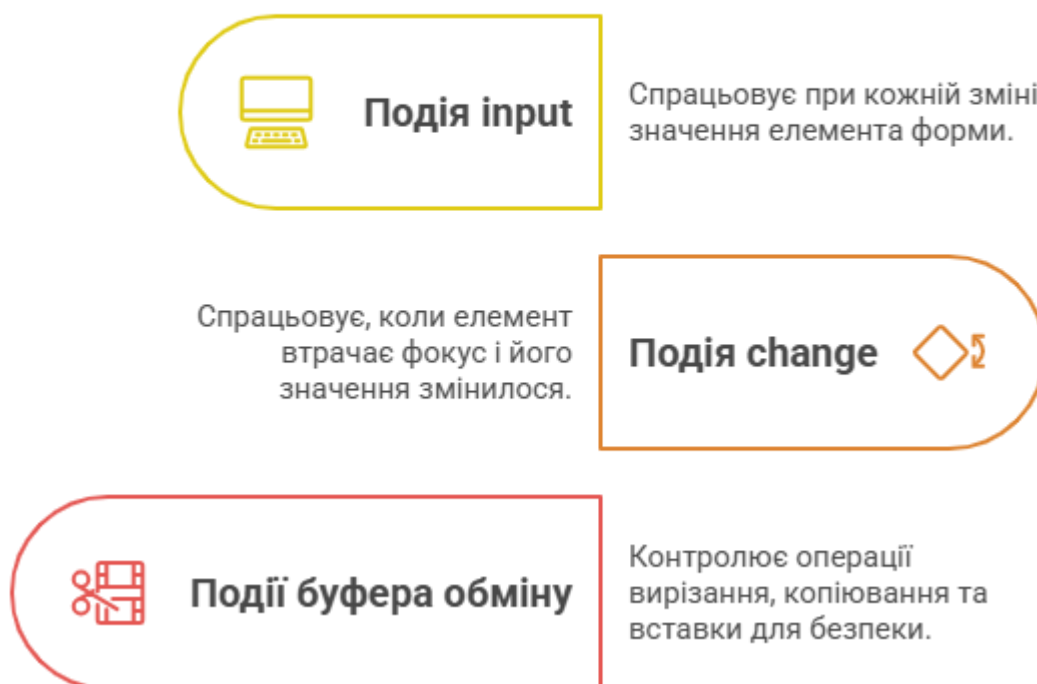


Рисунок 3.2 — Події форм

### Подія input - миттєве відстеження змін

Подія input є однією з найбільш універсальних та корисних подій для роботи з формами, оскільки вона спрацьовує щоразу, коли користувач вносить будь-які зміни до значення елемента форми. На відміну від інших подій, input реагує миттєво на кожен введений символ, що робить її ідеальною для реалізації функцій, які потребують негайного відгуку.

Типовим прикладом використання події input є створення системи пошуку з автодоповненням або перевірка доступності імені користувача під час реєстрації:

```
const searchInput = document.getElementById('search');
searchInput.addEventListener('input', function() {
  const query = this.value;
  if (query.length > 2) {
    console.log('Шукаємо:', query);
    // Тут буде логіка пошуку
```

```
    }  
  });
```

Особливо корисною є подія `input` для створення індикаторів складності пароля, які оновлюються в режимі реального часу під час введення:

```
const passwordInput =  
document.getElementById('password');  
const strengthIndicator =  
document.getElementById('strength');  
  
passwordInput.addEventListener('input', function() {  
  const password = this.value;  
  let strength = 0;  
  
  if (password.length >= 8) strength++;  
  if (/[A-Z]/.test(password)) strength++;  
  if (/[0-9]/.test(password)) strength++;  
  
  strengthIndicator.textContent = strength > 2 ?  
  'Сильний' : 'Слабкий';  
});
```

Подія `input` також чудово підходить для реалізації лічильників символів у текстових полях, що особливо актуально для соціальних мереж або форм з обмеженням довжини тексту:

```
const commentInput =  
document.getElementById('comment');  
const counterDisplay =  
document.getElementById('counter');  
const maxLength = 280;  
  
commentInput.addEventListener('input', function() {
```

```

    const remaining = maxLength - this.value.length;
    counterDisplay.textContent = `Залишилось символів:
    ${remaining}`;
    counterDisplay.style.color = remaining < 20 ? 'red'
    : 'black';
  });

```

### Подія change - фіксування завершених змін

Подія change має інший характер спрацьовування порівняно з input - вона активується тільки тоді, коли елемент втрачає фокус і його значення відрізняється від того, що було на момент отримання фокуса. Це робить подію change ідеальною для операцій, які не потрібно виконувати при кожній зміні символу, а лише після завершення введення користувачем.

Класичним прикладом використання події change є валідація електронної пошти після того, як користувач закінчив її вводити:

```

const emailInput = document.getElementById('email');
emailInput.addEventListener('change', function() {
    const email = this.value;
    const isValid = email.includes('@') &&
    email.includes('.');

    if (isValid) {
        this.style.borderColor = 'green';
        console.log('Email коректний');
    } else {
        this.style.borderColor = 'red';
        console.log('Email некоректний');
    }
});

```

Подія change також активно використовується для елементів select, radio-кнопок та checkbox, де вона спрацьовує відразу після зміни стану:

```

const countrySelect =
document.getElementById('country');
countrySelect.addEventListener('change', function() {
    console.log('Обрано країну:', this.value);
    // Завантажуємо список міст для обраної країни
});

const subscribeCheckbox =
document.getElementById('subscribe');
subscribeCheckbox.addEventListener('change', function()
{
    if (this.checked) {
        console.log('Користувач підписався на
розсилку');
    } else {
        console.log('Користувач відмовився від
розсилки');
    }
});

```

Важливою особливістю події `change` є те, що вона дозволяє виконувати "важкі" операції, такі як форматування даних або запити до сервера, не перевантажуючи систему частими викликами:

```

const phoneInput = document.getElementById('phone');
phoneInput.addEventListener('change', function() {
    // Форматуємо номер телефону після завершення
введення
    let phone = this.value.replace(/\D/g, ''); //
Видаляємо всі нецифрові символи
    if (phone.startsWith('0') && phone.length === 10) {

```

```

        this.value = `+38 (${phone.substr(1, 2)})
        ${phone.substr(3, 3)}-${phone.substr(6, 4)}`;
    }
});

```

### Події cut, copy, paste - контроль операцій з буфером обміну

Події cut, copy та paste надають можливість контролювати операції з буфером обміну, що може бути критично важливим для безпеки додатка або для забезпечення цілісності даних. Ці події спрацьовують відповідно при вирізанні, копіюванні та вставці тексту користувачем.

Подія copy часто використовується для запобігання копіюванню конфіденційної інформації:

```

const cardNumberInput =
document.getElementById('cardNumber');
cardNumberInput.addEventListener('copy',
function(event) {
    event.preventDefault(); // Забороняємо копіювання
    alert('Копіювання номера картки заборонено з
міркувань безпеки');
});

```

Аналогічно працює подія cut, яка дозволяє заборонити вирізання важливої інформації:

```

const passwordInput =
document.getElementById('password');
passwordInput.addEventListener('cut', function(event) {
    event.preventDefault();
    console.log('Спроба вирізання пароля заблокована');
});

```

Особливо цікавою є подія paste, яка дозволяє не тільки контролювати процес вставки, але й модифікувати вставлені дані перед їх розміщенням у полі:

```

const textInput = document.getElementById('cleanText');
textInput.addEventListener('paste', function(event) {
    event.preventDefault(); // Забороняємо стандартну
вставку

    // Отримуємо текст з буфера обміну
    const pastedText =
event.clipboardData.getData('text');

    // Очищаємо текст від HTML тегів та зайвих символів
    const cleanText = pastedText.replace(/<[^>]*>/g,
'').trim();

    // Вставляємо очищений текст
    this.value = cleanText;
    console.log('Вставлено очищений текст:',
cleanText);
});

```

Подія `paste` також може використовуватися для додаткової валідації вставленого контенту:

```

const emailInput = document.getElementById('email');
emailInput.addEventListener('paste', function(event) {
    const pastedData =
event.clipboardData.getData('text');

    // Перевіряємо, чи схожі вставлені дані на email
    if (!pastedData.includes('@')) {
        event.preventDefault();
        alert('Вставлені дані не схожі на електронну
адресу');
    }
}

```

```

        return;
    }

    console.log('Вставлено email:', pastedData);
});

```

### Практичне поєднання подій

У реальних проектах події часто використовуються в комбінації для створення складних інтерактивних сценаріїв. Наприклад, можна створити поле для введення тексту, яке одночасно відстежує зміни в режимі реального часу через `input`, виконує валідацію після завершення введення через `change`, та контролює операції з буфером обміну:

```

const smartInput =
document.getElementById('smartInput');
const statusDiv = document.getElementById('status');

// Реальний час - підрахунок символів
smartInput.addEventListener('input', function() {
    statusDiv.textContent = `Введено символів:
${this.value.length}`;
});

// Після завершення введення - валідація
smartInput.addEventListener('change', function() {
    if (this.value.length < 5) {
        statusDiv.textContent = 'Текст занадто
короткий';
        statusDiv.style.color = 'red';
    } else {
        statusDiv.textContent = 'Текст прийнято';
        statusDiv.style.color = 'green';
    }
});

```



```
    }  
  });  
  
  // Контроль вставки  
  smartInput.addEventListener('paste', function(event) {  
    const pastedText =  
event.clipboardData.getData('text');  
    console.log('Користувач намагається вставити:',  
pastedText.substring(0, 20) + '...');  
  });
```

Розуміння особливостей кожної з цих подій та їх правильне застосування дозволяє створювати по-справжньому інтерактивні та зручні форми, які реагують на дії користувача саме тоді, коли це потрібно, забезпечуючи оптимальний баланс між функціональністю та продуктивністю.

### 3.4.6 Форми: подія та метод submit



Рисунок 3.3 — Цикл надсилення форми

Процес надсилення форми є кульмінаційним моментом у взаємодії користувача з веб-додатком, оскільки саме в цей момент відбувається передача зібраної інформації на сервер для подальшої обробки. Подія submit та одноіменний метод надають розробникам потужні інструменти для контролю цього критично важливого процесу, дозволяючи виконувати фінальну валідацію, модифікувати дані та керувати поведінкою форми перед її фактичним надсиленням.

#### Подія submit - останній рубіж контролю

Подія submit спрацьовує безпосередньо перед тим, як браузер готується надіслати дані форми на сервер, що робить її ідеальним місцем для виконання

останньої перевірки та підготовки даних. Важливою особливістю цієї події є те, що вона може бути скасована за допомогою методу `preventDefault()`, що дозволяє розробникам повністю контролювати процес надсилання.

Базовий приклад обробки події `submit` виглядає наступним чином:

```
const loginForm = document.getElementById('loginForm');

loginForm.addEventListener('submit', function(event) {
    event.preventDefault(); // Зупиняємо стандартне
    надсилання

    console.log('Форма готова до надсилання');

    const username = this.elements.username.value;
    const password = this.elements.password.value;

    console.log('Дані для входу:', username, password);
});
```

Одним з найважливіших застосувань події `submit` є виконання комплексної валідації всіх даних форми перед їх надсиланням на сервер, що дозволяє заощадити ресурси сервера та забезпечити кращий користувацький досвід:

```
const registrationForm =
document.getElementById('registrationForm');

registrationForm.addEventListener('submit',
function(event) {
    event.preventDefault();

    const email = this.elements.email.value;
    const password = this.elements.password.value;
```

```

    const confirmPassword =
this.elements.confirmPassword.value;

    // Перевіряємо, чи всі поля заповнені
    if (!email || !password || !confirmPassword) {
        alert('Заповніть всі обов\'язкові поля');
        return;
    }

    // Перевіряємо коректність email
    if (!email.includes('@')) {
        alert('Введіть коректну електронну адресу');
        return;
    }

    // Перевіряємо збіг паролів
    if (password !== confirmPassword) {
        alert('Паролі не співпадають');
        return;
    }

    console.log('Всі перевірки пройдено, можна
надсилати форму');
    // Тут буде логіка надсилання на сервер
});

```

### **Індикатори завантаження та користувацький досвід**

Важливим аспектом обробки події submit є забезпечення зворотного зв'язку з користувачем під час обробки форми, оскільки надсилання даних на сервер може тривати деякий час, і користувач повинен розуміти, що його запит обробляється:

```
const contactForm =
document.getElementById('contactForm');
const submitButton =
contactForm.querySelector('button[type="submit"]');

contactForm.addEventListener('submit', function(event)
{
    event.preventDefault();

    // Показуємо, що форма обробляється
    submitButton.disabled = true;
    submitButton.textContent = 'Надсилання...';

    // Імітуємо надсилання на сервер
    setTimeout(() => {
        alert('Повідомлення надіслано успішно!');

        // Повертаємо кнопку до початкового стану
        submitButton.disabled = false;
        submitButton.textContent = 'Надіслати';

        // Очищаємо форму
        this.reset();
    }, 2000);
});
```

### **Робота з FormData та сучасні підходи**

Сучасні веб-додатки часто використовують AJAX для надсилання форм без перезавантаження сторінки, що забезпечує більш плавний користувацький досвід. Об'єкт FormData значно спрощує процес збирання та надсилання даних форми:

```

const uploadForm =
document.getElementById('uploadForm');

uploadForm.addEventListener('submit', async
function(event) {
    event.preventDefault();

    // Створюємо об'єкт FormData з усіма даними форми
    const formData = new FormData(this);

    // Можемо додати додаткові дані
    formData.append('timestamp', new
Date().toISOString());

    try {
        const response = await fetch('/api/upload', {
            method: 'POST',
            body: formData
        });

        if (response.ok) {
            console.log('Файл завантажено успішно');
        } else {
            console.log('Помилка завантаження');
        }
    } catch (error) {
        console.log('Помилка з\'єднання:', error);
    }
});

```

**Метод submit() - програмне надсилення форми**

Окрім події `submit`, JavaScript надає метод `submit()`, який дозволяє програмно ініціювати надсилення форми. Важливою особливістю цього методу є те, що він не викликає подію `submit`, тому всі обробники цієї події будуть проігноровані:

```
const form = document.getElementById('autoForm');
const autoSubmitButton =
document.getElementById('autoSubmit');

autoSubmitButton.addEventListener('click', function() {
    // Заповнюємо форму програмно
    form.elements.username.value = 'testuser';
    form.elements.email.value = 'test@example.com';

    // Надсилаємо форму без виклику події submit
    console.log('Автоматичне надсилення форми');
    form.submit();
});
```

Це означає, що якщо вам потрібно виконати валідацію перед програмним надсиленням, її треба робити вручну:

```
function validateAndSubmit(form) {
    const email = form.elements.email.value;

    if (!email) {
        alert('Введіть електронну адресу');
        return false;
    }

    if (!email.includes('@')) {
        alert('Електронна адреса некоректна');
        return false;
    }
}
```

```

    }

    // Якщо валідація пройшла успішно, надсилаємо форму
    form.submit();
    return true;
}

const quickSubmitButton =
document.getElementById('quickSubmit');
quickSubmitButton.addEventListener('click', function()
{
    const form = document.getElementById('quickForm');
    validateAndSubmit(form);
});

```

### **Обробка помилок та відновлення після невдач**

Професійна обробка події `submit` обов'язково включає механізми обробки помилок та відновлення інтерфейсу у випадку невдачі:

```

const orderForm = document.getElementById('orderForm');
const errorContainer =
document.getElementById('errorContainer');

orderForm.addEventListener('submit', async
function(event) {
    event.preventDefault();

    // Приховуємо попередні помилки
    errorContainer.style.display = 'none';

    const formData = new FormData(this);

```



```

    try {
        const response = await fetch('/api/orders', {
            method: 'POST',
            body: formData
        });

        if (!response.ok) {
            throw new Error(`Помилка сервера:
${response.status}`);
        }

        const result = await response.json();
        alert(`Замовлення створено! Номер:
${result.orderNumber}`);
        this.reset();

    } catch (error) {
        console.error('Помилка при надсиланні:',
error);

        errorContainer.textContent = 'Виникла помилка.
Спробуйте ще раз.';
        errorContainer.style.display = 'block';

        // Прокручуємо до помилки
        errorContainer.scrollIntoView({ behavior:
'smooth' });
    }
});

```

**Спеціальні сценарії та розширена функціональність**

У складних додатках подія submit може використовуватися для реалізації розширених сценаріїв, таких як збереження чернетки, багатосторінкові форми або умовне надсилання:

```
const draftForm = document.getElementById('draftForm');
let isDraft = false;

// Автоматичне збереження чернетки
setInterval(() => {
    if (hasUnsavedChanges()) {
        saveDraft();
    }
}, 30000); // Кожні 30 секунд

draftForm.addEventListener('submit', function(event) {
    event.preventDefault();

    if (isDraft) {
        // Зберігаємо як чернетку
        saveDraft();
        alert('Чернетку збережено');
    } else {
        // Фінальне надсилання
        if (confirm('Ви впевнені, що хочете надіслати форму?')) {
            submitFinalForm();
        }
    }
});

function saveDraft() {
```

```

    const data = new FormData(draftForm);
    localStorage.setItem('draft',
JSON.stringify(Object.fromEntries(data)));
    console.log('Чернетку збережено локально');
}

function hasUnsavedChanges() {
    return draftForm.querySelector('input, textarea,
select').value.length > 0;
}

```

Розуміння всіх нюансів роботи з подією submit та методом submit() є критично важливим для створення надійних та зручних веб-додатків, оскільки саме на цьому етапі відбувається фінальна взаємодія з користувачем та передача важливих даних на сервер для обробки.

## Висновки

У цьому розділі ми розглянули ключові принципи роботи з мовою JavaScript та її інтеграцією у веб-середовище. Було підкреслено, що якісна розробка починається з правильної організації коду: структуризації, логічного групування функцій і використання зрозумілих імен змінних та методів. Це підвищує не лише читабельність програм, а й спрощує подальшу підтримку та командну роботу.

Ми з'ясували, що DOM є основою для взаємодії JavaScript з HTML-документом. Його ієрархічна структура забезпечує зручний доступ до будь-яких елементів, їхніх властивостей та подій. Важливо вміти ефективно переміщатися по DOM-дереву, змінювати вміст і керувати станом елементів.

Особливу увагу було приділено різниці між атрибутами та властивостями. Розуміння цієї відмінності дозволяє правильно працювати з даними, забезпечувати доступність та оптимізувати швидкодію веб-застосунків.

Ми також розглянули механізми підключення скриптів, роботу з зовнішніми бібліотеками та роль кешування у пришвидшенні завантаження сайтів. Наведені приклади показали, що грамотна організація скриптів суттєво впливає на продуктивність і масштабованість проєктів.

Загалом матеріал розділу демонструє, що технології інтернет-програмування поєднують у собі як технічні аспекти (синтаксис і функціональність JavaScript), так і концептуальні принципи — структурність, логічність і безпеку. Опановуючи їх, отримують фундамент для створення динамічних, надійних і зручних для користувача веб-додатків.

### **Питання для самоперевірки**

- 1) Які особливості відрізняють JavaScript від інших мов програмування?
- 2) У чому полягають переваги використання консолі розробника в браузері?
- 3) Яка різниця між вбудованим та зовнішнім скриптом? У яких випадках доцільно застосовувати кожний підхід?
- 4) Що роблять атрибути `defer` та `async` у тегу `<script>`?
- 5) Чим відрізняється область видимості змінних, оголошених за допомогою `var`, `let` і `const`?
- 6) Які примітивні типи даних існують у JavaScript? Наведіть приклади їх використання.
- 7) Що таке складні типи даних? Як відрізняється їх зберігання від примітивних?
- 8) У чому різниця між властивостями та атрибутами HTML-елементів?
- 9) Для чого використовуються функції `alert`, `prompt` і `confirm`? Які їхні обмеження?
- 10) Які можливості надає об'єкт `Math` у JavaScript? Наведіть приклади.
- 11) Що таке функція? Які способи її оголошення ви знаєте?

- 12) У чому відмінність між звичайним оголошенням функції та функціональним виразом?
- 13) Що таке замикання і як воно використовується на практиці?
- 14) Які переваги мають стрілкові функції?
- 15) Що таке рекурсія? Наведіть приклад її застосування.
- 16) У чому полягає різниця між глобальною та локальною областю видимості змінних?
- 17) Як правильно організовувати структуру коду в JavaScript-проєкті?
- 18) Які практики сприяють покращенню читабельності та підтримуваності коду?
- 19) Які особливості виконання JavaScript у браузері слід враховувати під час розробки?
- 20) Чому правильне використання подій є важливим для динаміки вебсторінок?

## РОЗДІЛ 4. ГРАФІЧНИЙ ДИЗАЙН І UX-ПРОЄКТУВАННЯ ВЕБСАЙТІВ

### 4.1 Графічний дизайн сайту

Графічний дизайн сайту являє собою одну з найважливіших складових процесу створення веб-ресурсів, що визначає не лише зовнішній вигляд, але й функціональність, зручність використання та загальне сприйняття користувачами інтернет-сайту. Сучасний веб-дизайн об'єднує в собі мистецтво візуального представлення інформації з технічними можливостями веб-технологій, створюючи гармонійний синтез естетики та функціональності. У цьому розділі ми детально розглянемо ключові аспекти створення ефективного графічного дизайну сайту, починаючи від генерації початкових ідей і закінчуючи створенням цілісного та структурованого дизайн-макету.

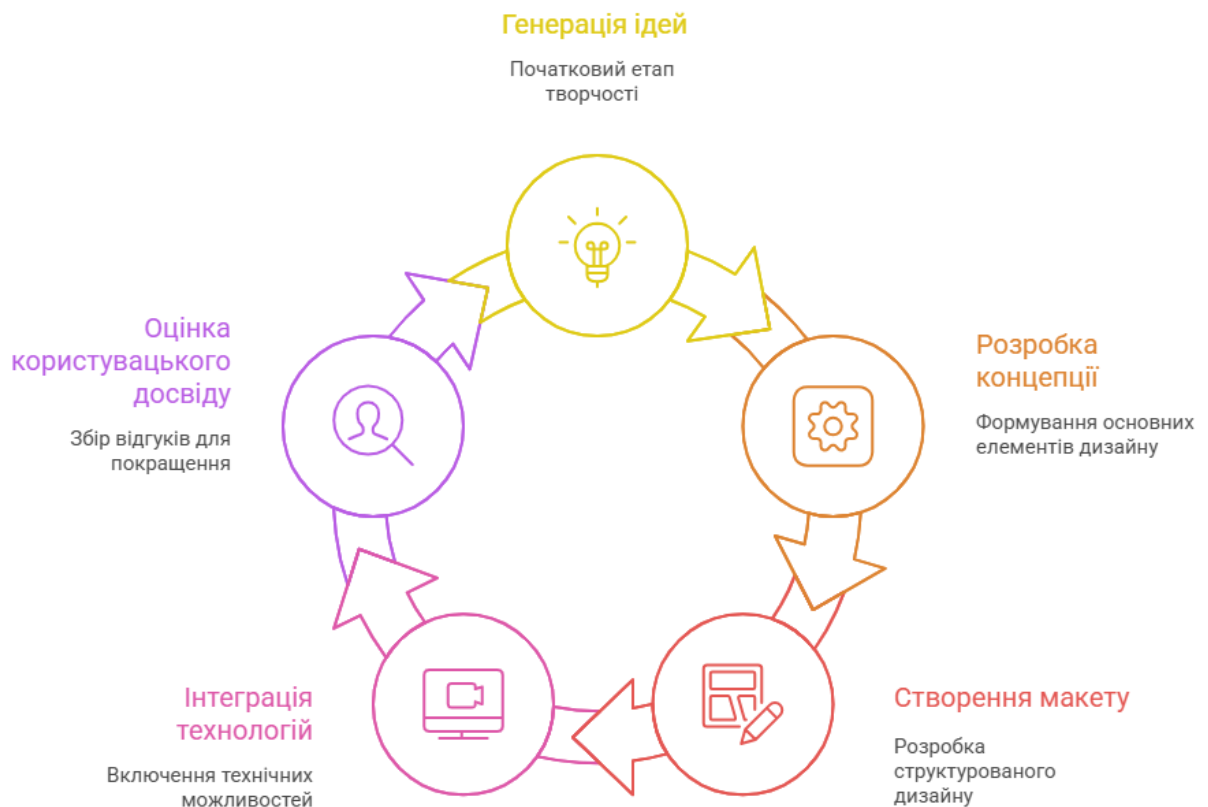


Рисунок 4.1 — Цикл графічного дизайну веб-сайту

#### 4.1.1 Генерація ідей дизайну

Процес генерації ідей для дизайну веб-сайту є творчим та багатоетапним процесом, який потребує глибокого розуміння цільової аудиторії, специфіки бізнесу чи організації, а також сучасних тенденцій у веб-дизайні. Початковим етапом генерації ідей завжди має стати ретельний аналіз потреб та очікувань майбутніх користувачів сайту, адже дизайн повинен насамперед слугувати їхнім інтересам та забезпечувати комфортне та інтуїтивно зрозуміле користування ресурсом.

Одним із найефективніших методів генерації ідей є створення мудбордів (mood boards) – візуальних колажів, які об'єднують кольорові схеми, типографічні рішення, фотографії, ілюстрації та інші візуальні елементи, що відображають бажаний настрій та атмосферу майбутнього сайту. Мудборди допомагають дизайнерам та замовникам візуалізувати концептуальні ідеї ще до початку роботи над конкретними макетами, створюючи спільне розуміння стилістичного напрямку проекту.

Дослідження конкурентів та аналіз існуючих рішень у відповідній галузі також відіграють важливу роль у процесі генерації ідей, проте важливо пам'ятати, що мета такого аналізу полягає не в копіюванні успішних рішень, а в розумінні сучасних стандартів та очікувань користувачів, що дозволяє створити унікальний та відмінний від конкурентів дизайн. Браузінгові галереї веб-дизайну, такі як Dribbble, Behance, Awwwards та інші подібні ресурси, можуть слугувати джерелом натхнення та демонструвати найкращі практики сучасного веб-дизайну.

Техніка скетчування залишається одним із найцінніших інструментів для швидкої візуалізації ідей, оскільки дозволяє дизайнерам експериментувати з різними варіантами компоновання елементів на сторінці без витрат часу на детальне опрацювання в графічних редакторах. Створення декількох альтернативних ескізів допомагає дослідити різні підходи до розташування контенту, навігації та ключових функціональних елементів сайту.



Рисунок 4.2 — Воронка генерації ідей веб-дизайну

**Схема перегляду сторінки** - це шлях, за яким погляд рухається по сторінці; шлях, який допоможе привернути увагу до важливого контенту на сайті.

Згідно цьому, погляд рухається зліва направо і зверху вниз.

Відповідно, значна кількість відвідувачів зупинить свій погляд на кнопці, що розташована в лівому верхньому кутку сайту, і це може привести до зростання числа кліків. В цьому місці варто розміщувати найважливіший контент.

Зазвичай, елементи сторінки розташовують наступним чином: логотип зліва вгорі, меню справа вгорі, інформаційні блоки, картинки зліва внизу, кнопка із закликом до дії справа внизу.



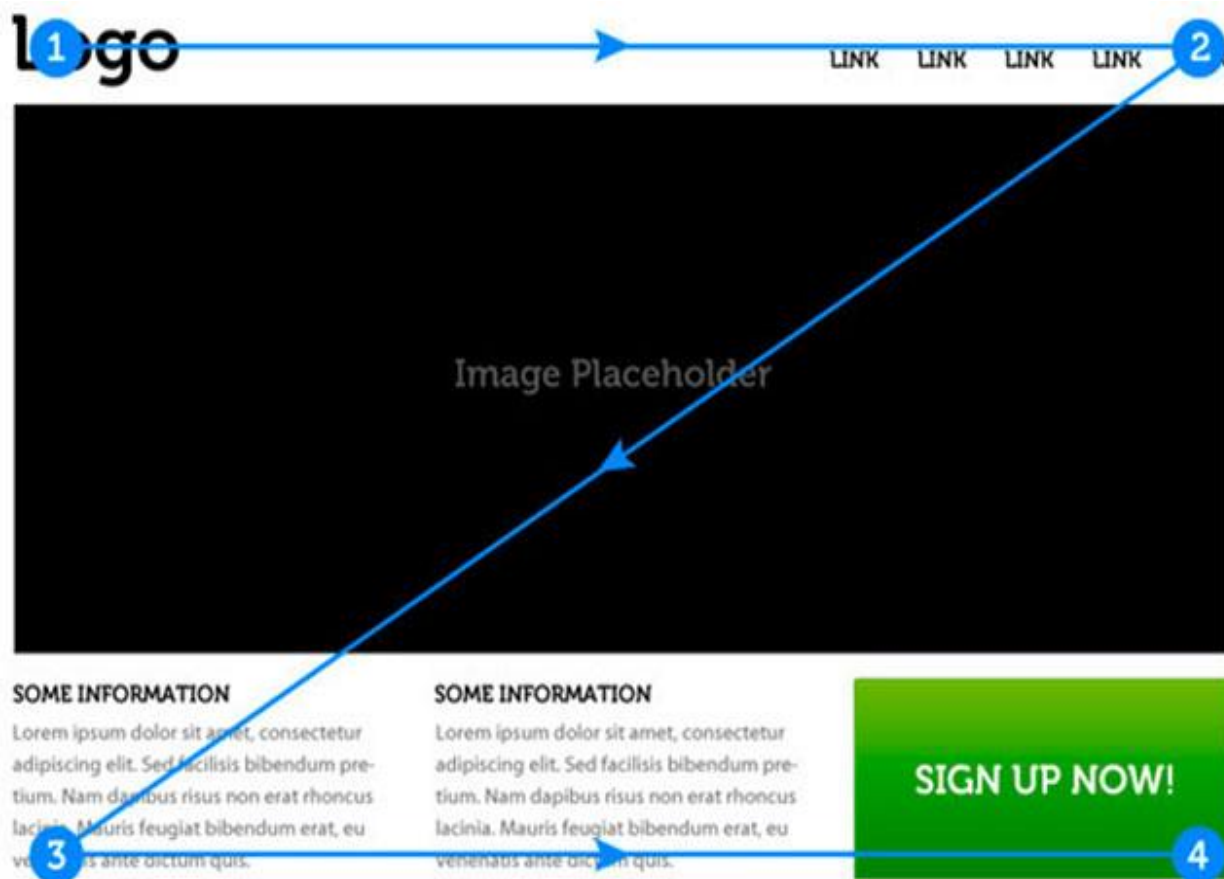


Рисунок 4.3 — Схема перегляду сторінки

**Візуальними напрямками** називають декоративні елементи сторінки, які скеровують погляд користувача на ті чи інші елементи дизайну, форми, кнопки та ін. Як візуальні напрямки можуть бути стрілки, напрям погляду людини на зображення, напрям вказівного пальця, в загальному все, що може вказувати в певний бік.



Рисунок 4.4 — Візуальні напрямки

### Елементи для спонукання здійснити дію

Спонукання до дії (Call to Action) відноситься до інтерактивних елементів сайту: кнопки, банери та інше.

Дані елементи оформлюються таким чином, щоб користувач захотів неодмінно на них натиснути.

Наприклад, це може бути кнопка із закликом до дії (Натисніть, Дізнайтеся тощо), яскравий банер з привабливою пропозицією чи яскравою картинкою.

Дане поняття добре вписується в принцип AIDA (Attention Interest Desire Action), який часто застосовується при дизайні головних сторінок, сторінок акцій, де необхідно підштовхнути користувача до тієї або іншої дії: підписка, покупка тощо.



Рисунок 4.5 — AIDA

Таким чином стає зрозумілим принцип побудови дизайну, що спирається на дане поняття: яскрава картинка чи банер повинні привернути увагу користувача, супутній підпис в тексті повинен викликати в нього інтерес і бажання, а завершальним акордом повинна стати, наприклад, кнопка із закликом до дії.

Але цей принцип не працює сам без деяких інших, наприклад, схеми перегляду сторінки чи візуальні напрямків.

#### 4.1.2 Розроблення макету головної сторінки сайту

Головна сторінка сайту є найважливішим елементом всього веб-ресурсу, оскільки саме вона формує перше враження відвідувачів та визначає їхнє подальше ставлення до сайту загалом. Розроблення макету головної сторінки потребує особливої уваги до балансу між інформативністю та простотою сприйняття, адже занадто перевантажена інформацією сторінка може відлякати користувачів, тоді як надмірно спрощений дизайн може не надати достатньо інформації для прийняття рішення про подальше знайомство з сайтом.

Структура головної сторінки зазвичай включає кілька ключових зон: шапку сайту з логотипом та основною навігацією, героблок або банерну зону з основним повідомленням чи пропозицією, секції з ключовим контентом, що розкривають основні переваги чи послуги, та підвал з додатковою інформацією та посиланнями. Кожна з цих зон має свої особливості

розміщення та оформлення, що вимагають ретельного планування та тестування з точки зору зручності використання.

Героблок, розташований зазвичай у верхній частині сторінки одразу після шапки, повинен миттєво повідомляти відвідувачеві про призначення сайту та основну цінність, яку він може отримати. Цей блок часто містить привабливий заголовок, короткий опис основної пропозиції та заклик до дії, оформлені на тлі якісного зображення чи відео, що підкреслює настрій та атмосферу бренду чи організації.

Планування контентних секцій головної сторінки має ґрунтуватися на принципі перевернутої піраміди, коли найважливіша інформація розміщується у верхній частині сторінки, а додаткові деталі та другорядна інформація – нижче. Такий підхід відповідає природній поведінці користувачів, які зазвичай сканують веб-сторінки згори вниз, приділяючи більше уваги верхній частині екрану.

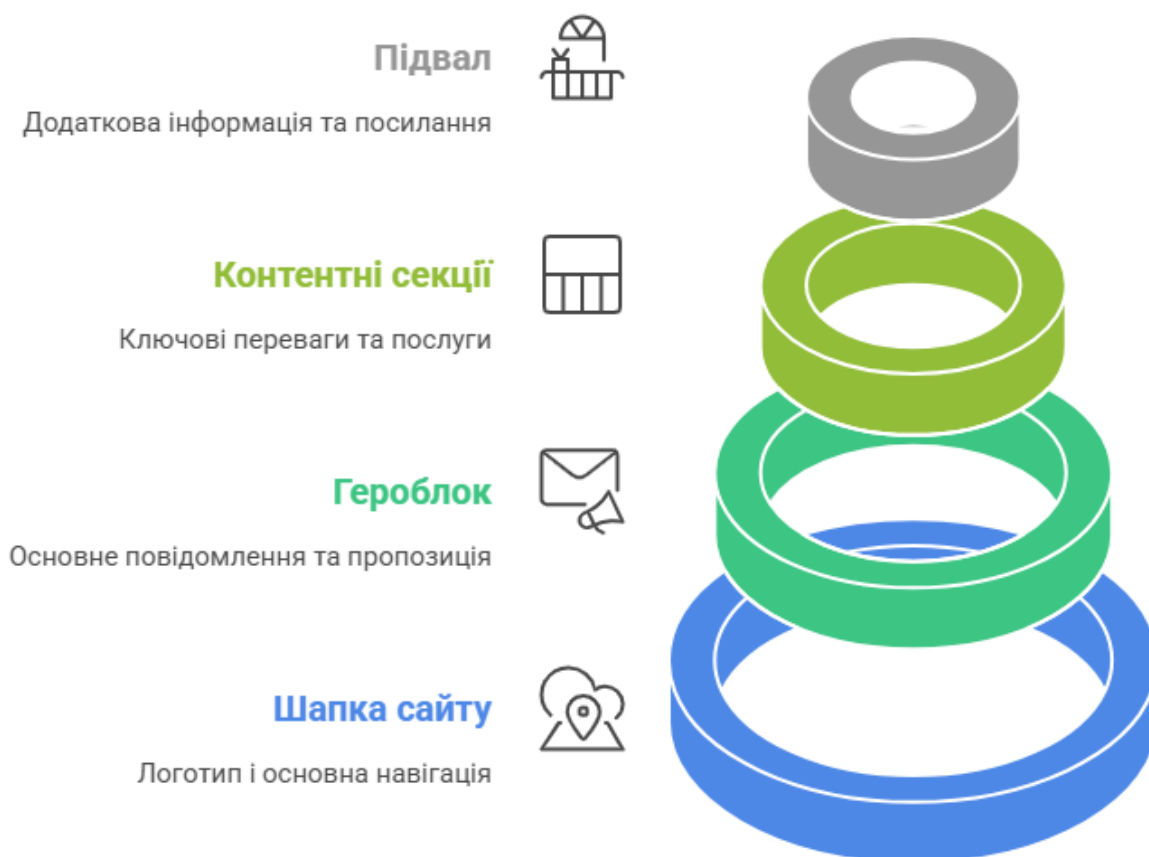


Рисунок 4.3 — Дизайн головної сторінки

Структура головної сторінки повинна бути простою і не містити зайвих елементів. Не слід зловживати занадто великою кількістю інформації і картинок. Контент сайту подається в акуратному і добре продуманому вигляді. В верхній частині сторінки розміщується сама важлива інформація: логотип, назва сайту та його напрямок або вид послуг, що надаються.

Доречно застосовані зображення можуть краще представити проект ніж багато рядків тексту. Слід використовувати зображення високої якості, які викличуть у користувачів бажання продовжити вивчення сайту.

Для підкреслення бренду компанії або тематики сайту слід використовувати колірну схему і фон, які добре працюють разом. Варто уникати використання надмірної кількості різних кольорів і не використовувати складний фон, що відвертає увагу з основного тексту та зображень.

Кнопки із закликом до дії ведуть на інші сторінки, сайти, промо-акції, каталоги і т.д. Їх слід зробити привабливими, щоб переконати користувачів натиснути на кнопку і зробити перехід. Текст на кнопці повинен бути коротким і чітким – одне-два слова.

Складно створити ідеальну головну сторінку, не використовуючи текст. Текст головної сторінки має коротко пояснити суть сайту. Не варто надавати великі секції з текстом, на головній сторінці обмежуються 1-2 абзацами, більш розлога інформація подається на відповідних сторінках сайту.

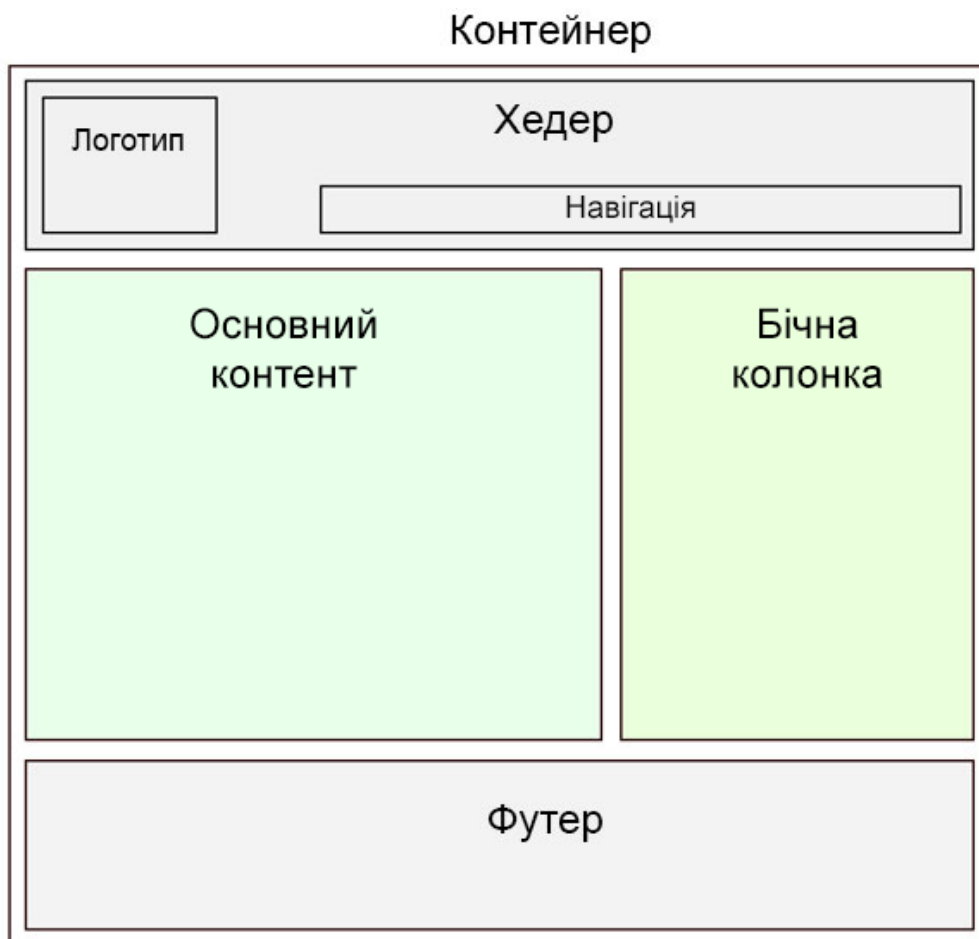


Рисунок 4.4 — Основні елементи сторінки

#### 4.1.3 Фіксований та еластичний макет

Вибір між фіксованим та еластичним макетом є одним із фундаментальних рішень у процесі створення веб-сайту, що суттєво впливає на сприйняття контенту користувачами з різними розмірами екранів та пристроями. Фіксований макет характеризується жорстко заданими розмірами елементів, зазвичай вказаними в пікселях, що забезпечує повний контроль дизайнера над зовнішнім виглядом сторінки та гарантує однакове відображення на всіх пристроях з подібними характеристиками екрану.

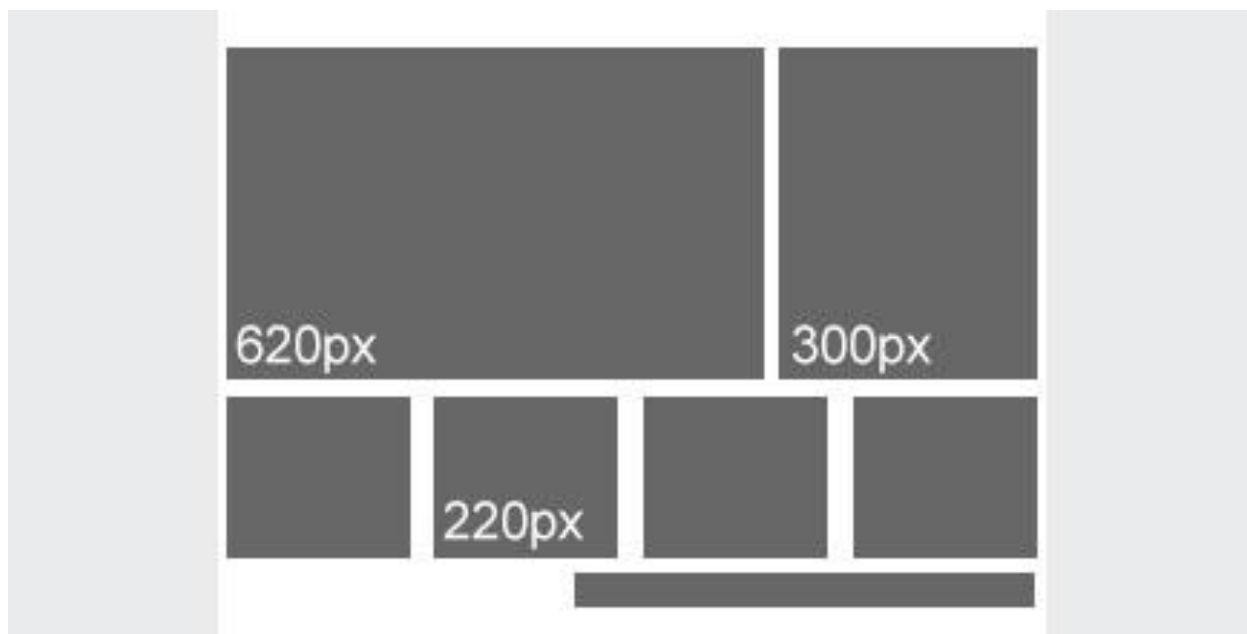


Рисунок 4.5 — Фіксований макет

Перевагами фіксованого макету є передбачуваність відображення дизайну, простота створення та тестування, а також можливість точного позиціонування всіх елементів відповідно до дизайнерського задуму. Проте такий підхід має суттєві недоліки в епоху різноманіття екранних розмірів, оскільки фіксований макет може погано адаптуватися до екранів з нестандартними розмірами, створюючи незручності для користувачів мобільних пристроїв або моніторів з високою роздільністю.

Еластичний макет, навпаки, використовує відносні одиниці виміру, такі як відсотки, `em` або `rem`, що дозволяє елементам сторінки адаптуватися до розміру вікна браузера та екрану пристрою. Цей підхід забезпечує більшу гнучкість та кращий користувацький досвід на різноманітних пристроях, проте вимагає від дизайнера більше уваги до тестування та оптимізації для різних розмірів екрану.

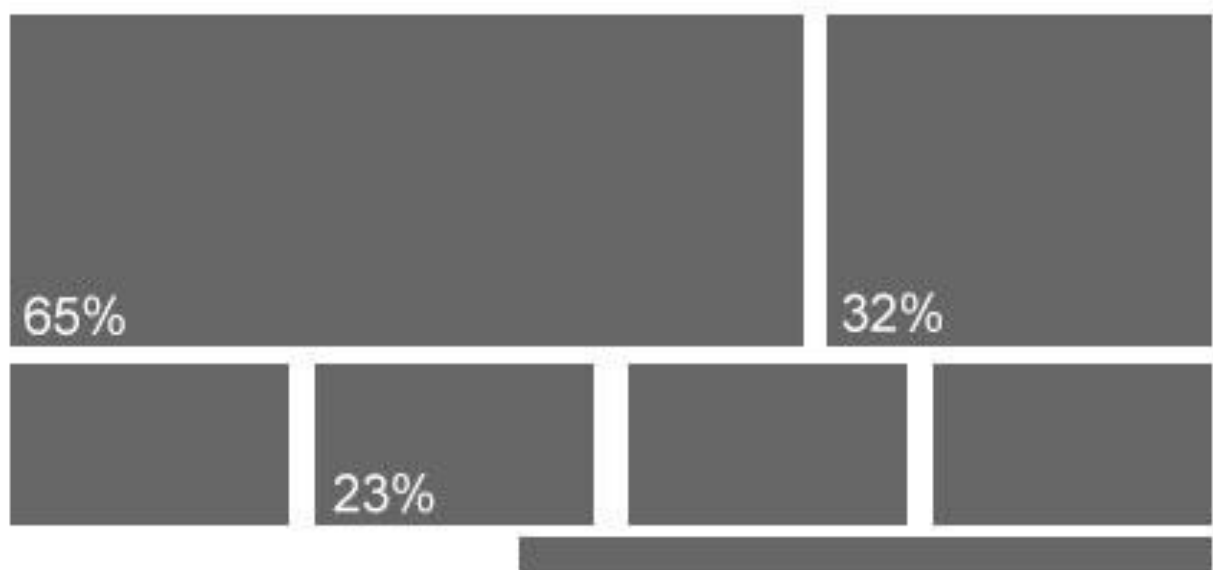


Рисунок 4.6 — Еластичний макет

Сучасні тенденції веб-дизайну все більше схиляються до використання гібридних підходів, що поєднують переваги обох методів. Такі рішення можуть включати фіксовані максимальні та мінімальні ширини для забезпечення оптимального відображення контенту, еластичні проміжки між елементами та адаптивні зображення, що змінюють свої розміри відповідно до доступного простору.



Рисунок 4.7 — Вибір макету для веб-сайту: баланс гнучкості та контролю

На сьогоднішній день введено такі поняття як чуйний веб-дизайн (Responsive Web Design) і адаптивний веб-дизайн (Adaptive Web Desing).

Перше поняття вкладається в концепцію «еластичного» і означає, що при зміні розміру екрану сайт підлаштовується під нього, друге поняття передбачає, що при розробці зазначаються основні розміри екрану, під які буде підлаштовуватися (адаптуватися) контент.



В обох випадках слід розробляти не один, а кілька макетів, які будуть відповідати різним розмірам екрану. Часто створюється 3 макета під розміри Phone, Tablet і Desktop.

При розробці макету мобільної версії сайту намагаються на перший план винести основний контент, тому, часто навігаційне меню перетворюється в «гамбургер-меню», великі банери та декоративні елементи приховуються, блоки контенту, зазвичай, розташовують один під одним.

На заздалегідь складеному макеті можна визначити, які елементи залишаються видимими на мобільному, а які будуть приховані.

#### **4.1.4 Дизайн під мобільні пристрої**

Швидкий розвиток мобільних пристроїв потребує сучасного підходу до розробки веб-сайтів. Інтернет-користувачі переглядають сайти на різних пристроях з екранами різних розмірів. Розміри екранів постійно змінюються, тому, важливо, щоб сайт адаптувався до будь-якого з них.

Мобільний дизайн перестав бути додатковою опцією і став обов'язковою вимогою для будь-якого сучасного веб-сайту, оскільки частка користувачів мобільних пристроїв продовжує стабільно зростати та в багатьох галузях уже перевищує частку користувачів настільних комп'ютерів. Підхід "mobile-first" передбачає, що дизайн спочатку створюється для найменших екранів, а потім поступово адаптується для більших розмірів, що забезпечує оптимальний користувацький досвід на всіх пристроях.

Особливості дизайну для мобільних пристроїв включають необхідність створення більших та легкодоступних інтерактивних елементів, оскільки користувачі керують інтерфейсом за допомогою дотиків пальців, що менш точні, ніж клацання мишею. Рекомендований мінімальний розмір інтерактивних елементів для мобільних пристроїв становить 44×44 пікселі, що забезпечує комфортне натискання без випадкових активацій сусідніх елементів.

Навігація на мобільних пристроях потребує особливого підходу, оскільки обмежений простір екрану не дозволяє розмістити всі навігаційні елементи у видимій області. Популярними рішеннями є використання "гамбургер-меню", що ховає навігаційні посилання під спеціальною іконкою з трьома горизонтальними лініями, або застосування нижньої навігації з іконками основних розділів сайту.

Оптимізація контенту для мобільних пристроїв також включає скорочення текстових блоків, використання більших шрифтів для кращої читабельності, та приділення особливої уваги швидкості завантаження сторінок, оскільки мобільні користувачі часто використовують повільніші інтернет-з'єднання та мають менше терпіння до тривалого очікування.

Існує два основні підходи створення сайтів, що легко адаптуються для різних типів пристроїв:

- Responsive Design (Responsive Web Design, RWD) - чуйний дизайн - проектування сайту з певними значеннями властивостей, наприклад, гнучка сітка макета, які дозволяють одному макету працювати на різних пристроях;
- Adaptive Design (Adaptive Web Design, AWD) - адаптивний дизайн - проектування сайту, що базується на кількох макетах фіксованої ширини.

Чуйний і адаптивний веб-дизайн тісно пов'язані і вирішують одну задачу, але в різний спосіб. Основна відмінність між цими прийомами - чуйний дизайн - один макет для всіх пристроїв, адаптивний дизайн – різні макети для різних пристроїв. Їх поєднання забезпечує ідеальну формулу для функціональних сайтів.

### **Чуйний дизайн**

Чуйний дизайн об'єднує три методики - гнучкий макет на основі сітки(fluid grid), гнучкі зображення (fluid images) і медіазапити. Пропорції та розміри елементів задаються у відсотках. При зменшенні ширини сторінки

весь вміст плавно стискається, структурні елементи зменшуються відносно один одного. Так, наприклад, якщо веб-сайт мав 3-х колонкову структуру, то на вузькому екрані він буде мати дві або одну колонки контенту.

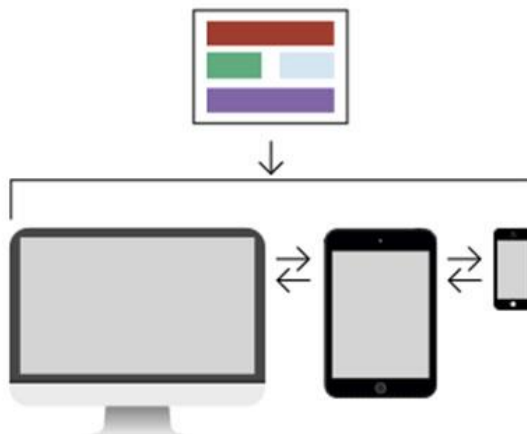


Рисунок 4.8 — Макет для чуйного дизайну

Для створення чуйної версії веб-сайту використовуються медіа запити (media queries) - блоки коду із вказуванням параметрів екрану. Медіа-запити дозволяють застосовувати певні правила (стилі) для виведення контент-блоків у різний порядок та пропорції залежно від ширини екрана та можливостей пристрою, на якому відбувається перегляд сайту.

### Адаптивний дизайн

На відміну від чуйного дизайну, адаптивний дизайн орієнтується на розміри пристроїв. Він використовує кілька статичних макетів для різних типів пристроїв, базуючись на контрольних (переломних) точках. Макети завантажуються при певних розмірах вікна браузера пристрою, а переходи між макетами відбуваються стрибкоподібно, а не плавно.

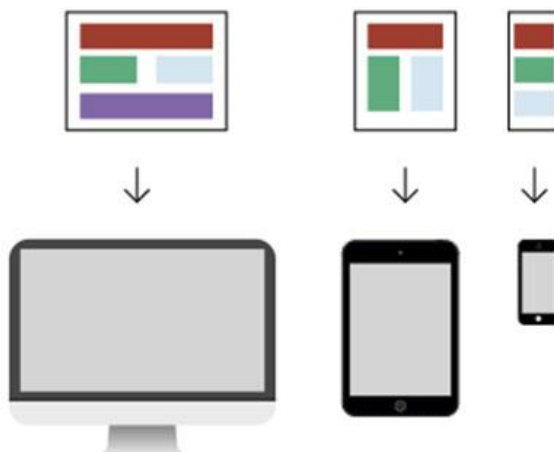


Рисунок 4.9 — Макети для адаптивного дизайну

В адаптивному дизайні серверні скрипти спочатку визначають тип пристрою, за допомогою якого користувач намагається отримати доступ до сайту (настільний ПК, телефон або планшет), потім завантажує саме ту версію сторінки, яка найбільш оптимізована для нього. Для елементів сітки задаються фіксовані розміри.

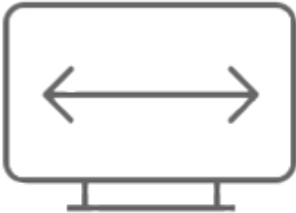
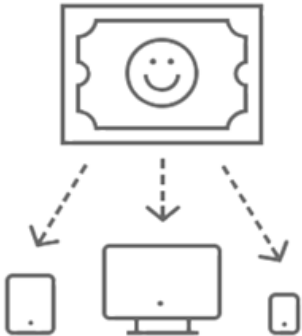
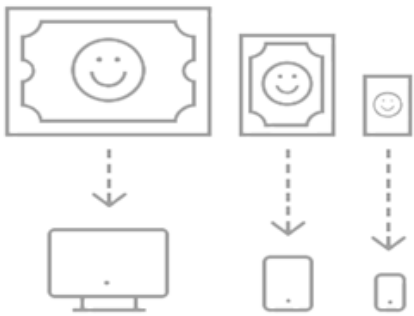
### Порівняння чуйного та адаптивного дизайну

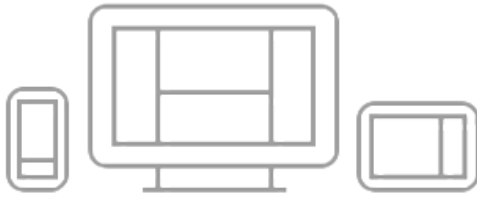
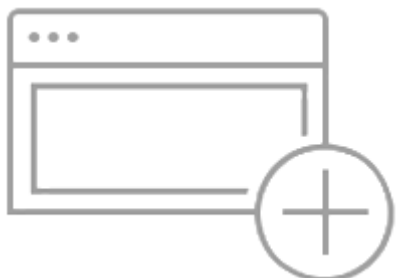
Чуйний дизайн більше підходить тоді, коли немає різниці між переглядом і використанням веб-сайту на мобільному і десктоп пристрої, а також коли бюджет і термін розробки обмежені.

Адаптивні сайти найбільш затребувані тоді, коли важливою є швидкість завантаження сторінок і взаємодія користувача з мобільною версією сильно відрізняється від використання десктоп версії.

Кожен випадок створення мобільної версії сайту потрібно розглядати окремо. Для простоти прийняття рішення нижче наведено порівняльну таблицю використання чуйного і адаптивного дизайну.

Таблиця 4.1 — Порівняння чуйного та адаптивного дизайну

| Характеристики       | Responsive Design                                                                                                                                                                                                                        | Adaptive Design                                                                                                                                                                                                                     |
|----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Визначення пристрою  |  <p>Використовуються медіа запити, що формують структуру сторінки щодо розмірів екрану пристрою, для всіх пристроїв використовується один шаблон.</p> |  <p>Використовується серверне або браузерні визначення пристрою, після чого застосовується конкретний шаблон саме для цього типу пристрою.</p> |
| Оптимізація контенту |                                                                                                                                                       |                                                                                                                                                |

|                                           |                                                                                                                                                                                                                                                               |                                                                                                                                                                                                                                                                                                                                                                             |
|-------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                           | Завантажується весь контент, для всіх типів пристроїв, незалежно від того, використовується він чи ні                                                                                                                                                         | Завантажується підготовлений, і оптимізований контент під конкретний пристрій                                                                                                                                                                                                                                                                                               |
| Адаптація під пристрій і його особливості |  <p>Використовується один шаблон для всіх пристроїв. Використовуються однакові блоки, модулі, медіа файли, як для мобільного, так і для десктопного перегляду.</p>           |  <p>Створюються різні унікальні шаблони, що призначені для різних пристроїв. Поведінка сайту змінюється в залежності від типу пристрою.</p>                                                                                                                                              |
| Швидкість завантаження                    |  <p>Швидкість завантаження нижче, оскільки користувачу потрібно чекати, поки завантажиться весь контент і всі структурні елементи для всіх розмірів екрану</p>              |  <p>Швидкість завантаження вище, оскільки завантажиться оптимізований контент, створений безпосередньо для певного пристрою і розміру його екрану</p>                                                                                                                                   |
| Розробка і запуск                         |  <p>Потрібна переробка основного сайту. При проектуванні і розробці сайту, потрібно враховувати і оптимізувати взаємодію користувача відразу під всі можливі пристрої.</p> |  <p>Більш затратне, але без змін існуючого сайту Адаптивні шаблони можуть розроблятися на базі існуючого сайту, без його зміни. Потрібно більш висока кваліфікація розробників, етап проектування і дизайну триває довше, відповідно, створення такого сайту буде більш витратним.</p> |

### 4.1.5 Модульна сітка

Модульна сітка є структурною основою веб-дизайну, що забезпечує впорядкованість та візуальну гармонію розташування елементів на сторінці. Система сітки базується на поділі доступного простору на рівномірні колонки з визначеними проміжками між ними, що дозволяє дизайнерам створювати збалансовані та професійно виглядаючі макети з чіткою ієрархією контенту.

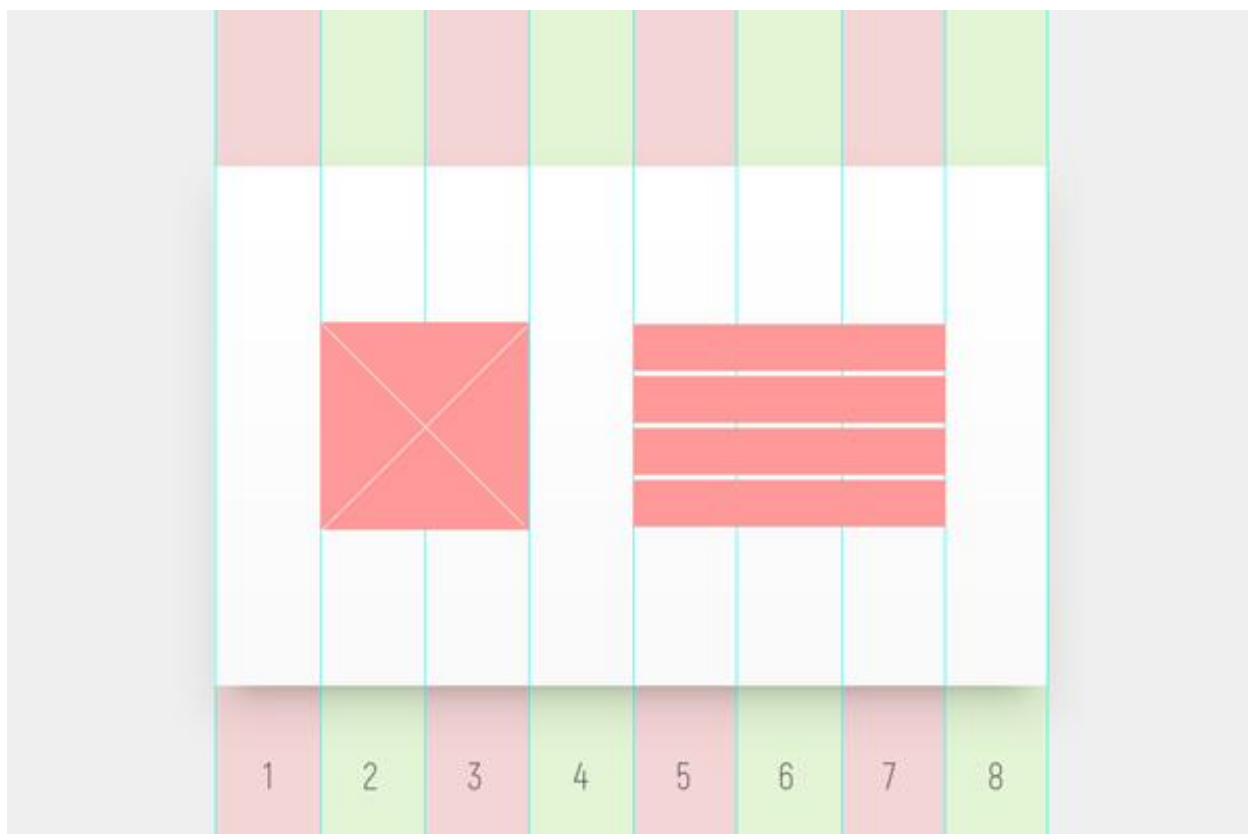


Рисунок 4.10 — Модульна сітка

Найпопулярнішою є дванадцятиколонкова сітка, що забезпечує максимальну гнучкість у розташуванні елементів, оскільки число 12 легко ділиться на 1, 2, 3, 4, 6 та 12, дозволяючи створювати різноманітні комбінації ширини блоків контенту. Така сітка може легко адаптуватися для різних розмірів екрану: наприклад, блок, що займає 4 колонки на настільному комп'ютері, може розширитися до 6 колонок на планшеті та до повної ширини на мобільному телефоні.

|    |   |   |   |   |   |   |   |   |   |   |   |
|----|---|---|---|---|---|---|---|---|---|---|---|
| 1  | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2  |   | 2 |   | 2 |   | 2 |   | 2 |   | 2 |   |
| 3  |   |   | 3 |   |   | 3 |   |   | 3 |   |   |
| 4  |   |   |   | 4 |   |   |   | 4 |   |   |   |
| 6  |   |   |   |   |   | 6 |   |   |   |   |   |
| 12 |   |   |   |   |   |   |   |   |   |   |   |

Рисунок 4.11 — Модульна сітка

Використання модульної сітки суттєво спрощує процес створення респонсивного дизайну, оскільки надає дизайнерам та розробникам єдину систему координат для всіх елементів інтерфейсу. Сучасні CSS-фреймворки, такі як Bootstrap, Foundation та інші, надають готові рішення для реалізації модульних сіток з вбудованими можливостями адаптивності.

Важливим аспектом роботи з модульною сіткою є розуміння принципу "gutters" – проміжків між колонками, що забезпечують візуальне розділення контенту та покращують читабельність. Розміри цих проміжків мають бути пропорційними до загального дизайну та забезпечувати достатній простір для комфортного сприйняття інформації користувачами.

#### 4.1.6 Навігаційне меню

Навігаційне меню є критично важливим елементом будь-якого веб-сайту, що забезпечує користувачам можливість швидкого та зручного переміщення між різними розділами та сторінками ресурсу. Ефективна навігація має бути інтуїтивно зрозумілою, легкодоступною та послідовною в усіх розділах сайту, щоб користувачі могли без труднощів знайти потрібну інформацію та виконати необхідні дії.

Основними типами навігаційних меню є горизонтальна навігація, розташована зазвичай у верхній частині сторінки, вертикальна бічна навігація, що може знаходитися зліва чи справа від основного контенту, та навігація

підвалу, що містить додаткові посилання та службову інформацію. Кожен тип навігації має свої переваги та найкраще підходить для певних типів сайтів та обсягів контенту.

Створення ефективної структури навігації починається з аналізу контенту сайту та групування споріднених сторінок у логічні категорії. Принцип "правила семи" рекомендує обмежувати кількість пунктів головного меню до семи елементів, оскільки більша кількість може перевантажувати користувачів та ускладнювати прийняття рішень. Для сайтів з великим обсягом контенту рекомендується використовувати багаторівневу навігацію з підменю та допоміжними категоріями.

Візуальне оформлення навігації має підкреслювати поточне місцезнаходження користувача на сайті через виділення активного пункту меню та надання зворотного зв'язку при взаємодії з навігаційними елементами. Breadcrumb-навігація (хлібні крихти) додатково допомагає користувачам орієнтуватися в структурі сайту та швидко повертатися до попередніх розділів.





Рисунок 4.12 — Створення ефективної навігації сайту

Горизонтальне меню підходить для сайтів з невеликою кількістю розділів, а також для односторінкових сайтів. Зазвичай, розташовується під або над шапкою сторінки.

Опції меню виконують у вигляді тексту, іконок, вкладок. Назви опції мають бути в 1-2 слова. Хорошим тоном є виділення поточної опції кольором, підкресленням або в інший спосіб. Якщо сайт достатньо довгий, горизонтальне меню можна зафіксувати, щоб воно не прогорталось.

Слабке місце горизонтальних меню полягає в обмеженні кількості опцій. Тому, сайти зі складною структурою використовують випадні меню, які допомагають приховати підрозділи і не захащувати простір сайту. Випадні

меню підходять для сайтів з розгалуженою структурою: магазинів з широким асортиментом і великою кількістю груп товарів.

Часто на потужних ресурсах одночасно використовують горизонтальне і вертикальне меню.

Вертикальне меню може містити значно більшу кількість опцій, що складаються з більшої кількості слів. Таке меню є зручним для магазинів і каталогів, де кількість товарів може збільшуватися. Широко вживаються групування опцій, розгортання підменю, іконки.

Меню-гамбургер — популярний тип меню, що на сьогодні використовують не лише на мобільних, а й на десктопних версіях сайтів. У ньому може ховатися кілька пунктів або повний набір розділів і підрозділів. Недолік такого типу меню - непомітність, тому варто подбати про те, щоб користувачеві було зрозуміло, що перед ним саме меню.



Рисунок 4.13 — Гамбургер-меню

Рекомендації при розробці навігаційного меню:

1. Логотип компанії завжди повинен вести на головну сторінку сайту. Згідно досліджень 36% людей схильні використовувати логотип як засіб переходу до головної сторінки.
2. Горизонтальне меню має містити якомога менше його опцій (від 3 до 7). Більша кількість заплутує відвідувачів. Назви опцій повинні бути лаконічними, але разом з тим і ємними.
3. Меню повинні бути оптимальними за розміром для екранів різних пристроїв: текст буде простіше читати, а на кнопки - натискати.
4. Ефективним є виділення опцій кольором або hover-ефектом, щоб мати можливість вказати відвідувачам, в якій області сайту вони знаходяться в даний момент.

5. Фіксовані меню, що залишаються на екрані при скролінгу, значно допомагають відвідувачам. Це відноситься до будь-якого стилю навігації: мобільного, десктопного, горизонтального і вертикального.
6. Для користувачів мобільних і десктопних пристроїв пропонуються різні варіанти меню, що пов'язано з розмірами екрану.
7. Меню для мобільної версії сайту варто робити більш виразним і краще оточити рамкою, щоб надати вигляд кнопки.
8. Застосовані іконки мають бути легко впізнаваними за змістом.
9. Глибокі, багаторівневі меню часто приводять до розгублення відвідувачів. Як альтернативу можна використовувати мега-меню і навігаційні ланцюжки (breadcrumbs), щоб полегшити подальшу навігацію.
10. Креативний варіант меню краще розміщати в очікуваному для відвідувачів місці. Багато дизайнерів експериментують з рорир-формами, оскільки несподіваний рух на екрані має добре захоплювати увагу відвідувачів. Але спливаючі вікна не повинні бути зведені до вікон, що оповіщають про маркетингові пропозиції та інше, вони повинні використовуватися суто для навігації.

#### 4.1.7 Типографіка

Типографіка у веб-дизайні відіграє ключову роль у забезпеченні читабельності контенту та формуванні загальної візуальної ідентичності сайту. Правильний вибір шрифтів, їх розмірів, міжрядкових інтервалів та кольорів суттєво впливає на сприйняття інформації користувачами та загальне враження від сайту. Сучасний веб надає дизайнерам широкі можливості для використання різноманітних шрифтових рішень завдяки веб-шрифтам та сервісам типу Google Fonts.

Основним принципом веб-типографіки є забезпечення максимальної читабельності тексту на різних пристроях та в різних умовах освітлення. Це

вимагає ретельного підбору контрасту між кольором тексту та тла, використання достатніх розмірів шрифтів (зазвичай не менше 16 пікселів для основного тексту) та оптимальних міжрядкових інтервалів, що становлять приблизно 1.4-1.6 від розміру шрифту.

Типографічна ієрархія створюється за допомогою різних розмірів, насиченості та кольорів шрифтів для заголовків різних рівнів, основного тексту, підписів та додаткової інформації. Правильно побудована ієрархія допомагає користувачам швидко сканувати контент та знаходити потрібну інформацію, а також створює візуальний ритм та структуру сторінки.

Вибір шрифтової пари – поєднання основного шрифту для тексту та акцентного шрифту для заголовків – є важливим дизайнерським рішенням, що впливає на характер та настрій сайту. Класичним підходом є поєднання serif та sans-serif шрифтів, що створює контраст та візуальний інтерес, проте сучасні тенденції також включають використання єдиного шрифтового сімейства з різними варіантами насиченості та стилю.



Рисунок 4.13 — Компоненти ефективної веб-типографіки

#### 4.1.8 Порядок у вихідному макеті дизайну

Створення впорядкованого та структурованого дизайн-макету є заключним етапом проектування, що забезпечує ефективну передачу проекту до етапу розробки та підтримки в майбутньому. Правильно організований макет має включати чіткі вказівки щодо розмірів, відступів, кольорових кодів, використаних шрифтів та інших технічних специфікацій, необхідних для точної реалізації дизайну в коді.

Система найменування та організації слоїв у графічних редакторах відіграє критичну роль у створенні професійного макету, що може бути легко зрозумілий та модифікований іншими дизайнерами чи розробниками. Рекомендується використовувати логічні назви для всіх елементів, групувати споріднені слої у папки та підтримувати послідовну структуру в усіх макетах проекту.

Документування дизайн-системи включає створення стайлгайду або дизайн-системи, що містить всі використані кольори з їх точними кодами, специфікації шрифтів з розмірами та міжрядковими інтервалами, розміри та стилі кнопок, форм та інших інтерактивних елементів, а також правила використання логотипів, зображень та інших брендингових елементів.

Підготовка асетів для розробки включає експорт всіх зображень, іконок та графічних елементів у відповідних форматах та розмірах, створення файлів для різних роздільностей екрану (1x, 2x, 3x для підтримки високощільності пікселів) та організацію всіх файлів у логічну структуру папок з зрозумілими назвами.

Усі етапи розробки макету веб-сторінки виконуються у спеціалізованих графічних програмах. Найбільш поширеним інструментом, яким користуються дизайнери, є Adobe Photoshop.

1. Грамотно підготовлений макет слід зберігати у форматі \*.psd із чітко впорядкованою структурою шарів і груп.

2. Перед передачею файлу видаляють непотрібні елементи, усі шари отримують осмислені назви, а для текстових елементів створюють растрові копії. Частини, що належать одному блоку або компоненту, об'єднують у відповідні папки. Непотрібні шари видаляються, щоб уникнути плутанини та зменшити вагу файлу.
3. У макеті мають бути присутні напрямні, побудовані за модульною сіткою. Це допомагає швидко знаходити потрібні елементи, замінювати або редагувати їх.
4. Якщо у дизайні використано нестандартні шрифти, їх додають до пакета з макетами, оскільки PSD-файли не зберігають шрифтових даних. Для подальшої верстки через @font-face ці шрифти інтегрують у сторінку, що економить час верстальника.
5. Додатково макет зберігають у поширених растрових форматах (.jpg або .png) для попереднього перегляду та погодження з замовником.
6. До пакета передаються й текстові файли з поясненнями для верстальника, який буде створювати шаблони сторінок. Через можливі відмінності у кодуванні різних операційних систем рекомендується використовувати латинські символи у назвах файлів.
7. Дизайнеру бажано володіти базовими знаннями HTML та CSS, а також розуміти основні принципи верстки і використовувані фреймворки. Це дає змогу створювати елементи, які легко реалізуються кодом, без надмірної залежності від растрових зображень.

## 4.2 Принципи композиції сайтів

Композиція сайтів є одним з найважливіших аспектів веб-дизайну, який визначає, як користувачі сприймають та взаємодіють з веб-ресурсом. Правильно побудована композиція не лише привертає увагу відвідувачів, але й

направляє їх погляд у потрібному напрямку, створює зручну навігацію та забезпечує позитивний користувацький досвід.

#### **4.2.1 Принципи веб дизайну**

Сучасний веб-дизайн є поєднанням естетики та функціональності, де важливо не лише створити красивий інтерфейс, а й забезпечити зручність користувача, логічність подання інформації та швидке досягнення цілей взаємодії з веб-ресурсом. Принципи композиції сайтів охоплюють сукупність правил і рекомендацій, що дозволяють розробникам створювати візуально гармонійні та ефективні інтерфейси.

Принципи веб дизайну:

1. Баланс
2. Контраст
3. Акцент
4. Рух
5. Ритм
6. Ієрархія
7. «Білий простір»
8. Послідовність

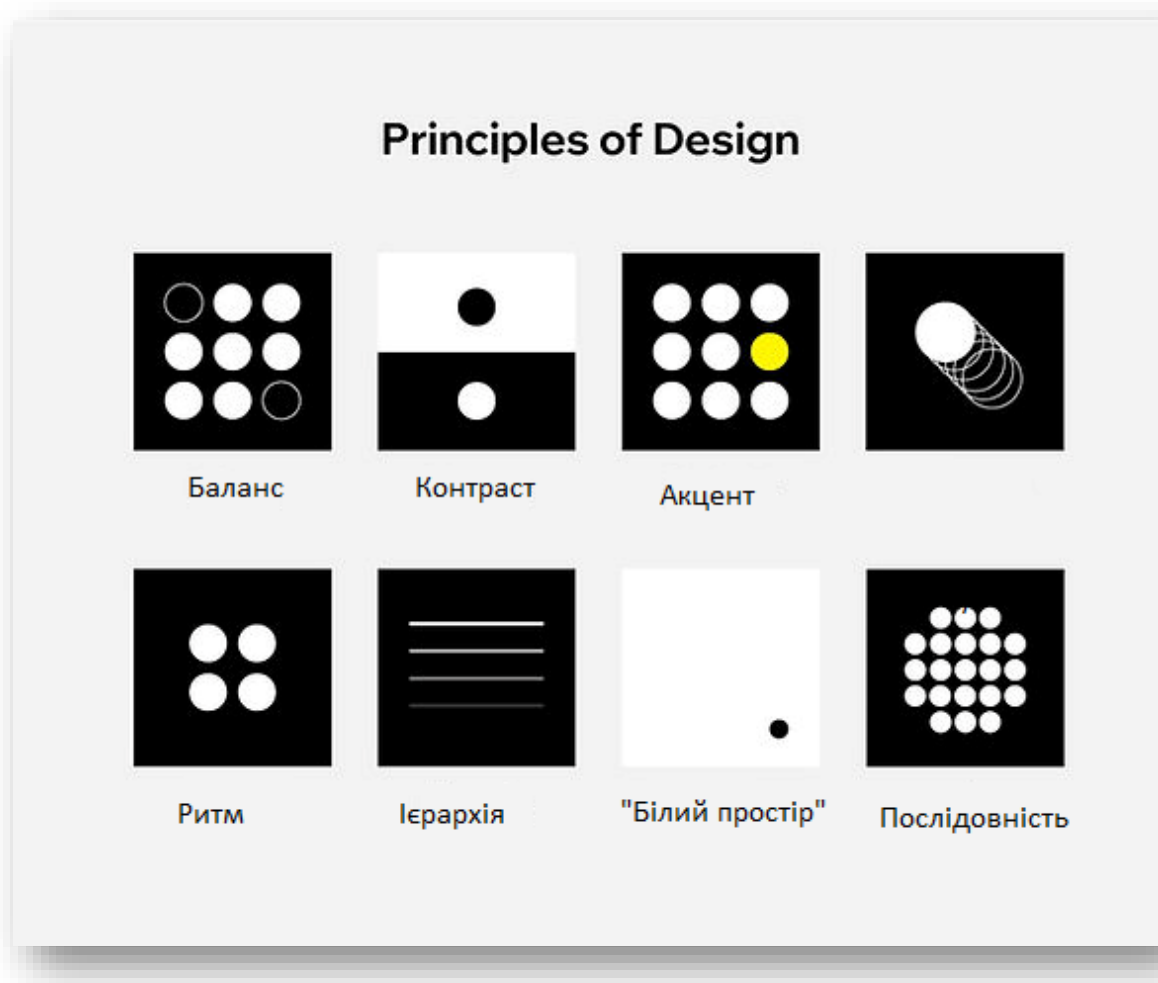


Рисунок 4.14 — Принципи веб дизайну

Візуальний баланс означає, що жоден з елементів у межах будь-якої композиції не затьмарюватиме інші. Щоб досягти цього, прокресліть уявну лінію посередині екрана і переконайтеся, що елементи на правій і лівій стороні виглядають рівноцінно. Існує два способи досягти балансу у веб-дизайні:

- Симетричний баланс — це дзеркальне відображення елементів на сторінці, де право і ліворуч практично ідентичні. Сторінки із симетричним балансом можуть викликати у користувача почуття надійності, гармонії та краси.
- Асиметричний баланс – це ситуація, коли елементи на правій та стороні сайту відрізняються,



але сходяться за загальною візуальною вагою (наприклад, один великий графічний елемент на одній частині і кілька маленьких на іншій). Такий баланс вважається динамічнішим і охоче використовується в сучасному веб-дизайні.

На малюнку нижче - приклад симетричного балансу (ліворуч) та асиметричного балансу (праворуч).



Рисунок 4.15 — Баланс

Контраст – це розташування елементів таким чином, щоб підкреслити їх відмінності та виділити на фоні один одного. Наприклад, темне та світле, легке та важке, велике та маленьке. З контрастом веб дизайн виходить цікавішим і захоплюючим, а значить, з більшою ймовірністю утримує увагу користувачів на сторінці.

На малюнку нижче - приклад хорошого розмаїття (ліворуч) і поганого розмаїття (праворуч). Як видно з прикладу справа, неконтрастний сірий повністю втрачається і натомість контрастних монохромних кольорів.

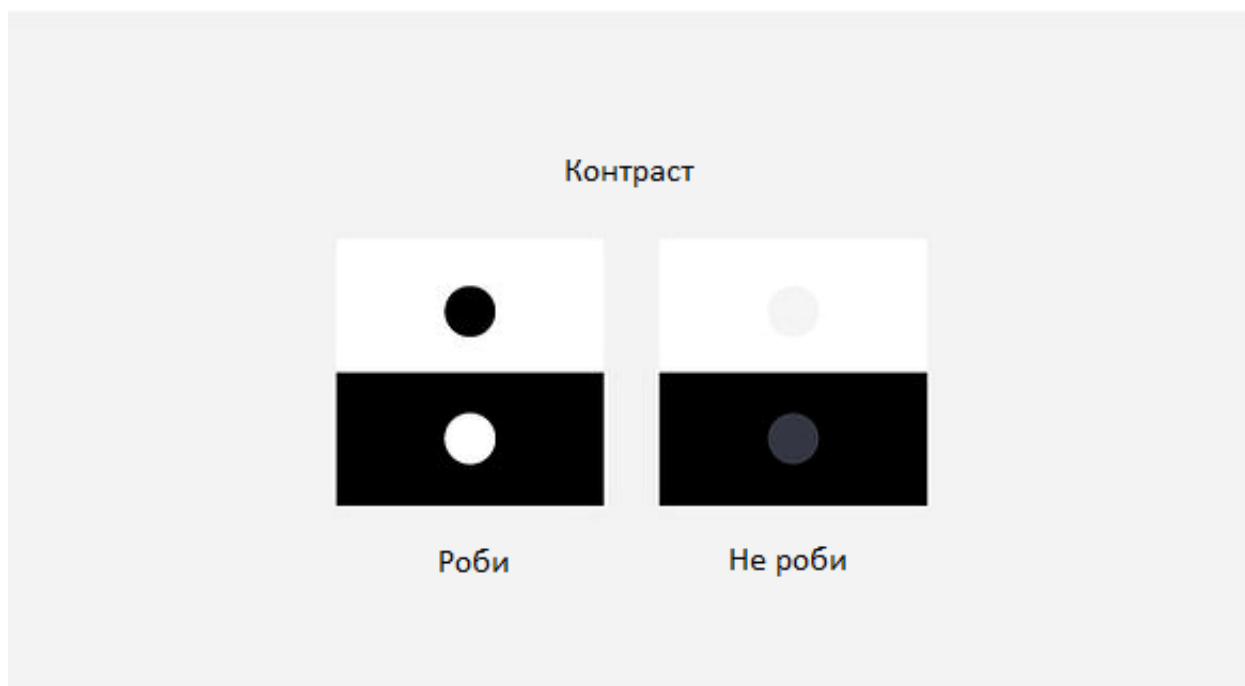


Рисунок 4.16 — Контраст

Не всі елементи на сайті мають бути однаково важливими. Деякі повинні впадати у вічі з першої секунди на сайті: логотип компанії, заклик до дії або кнопка. Виділивши важливий елемент, ви не тільки привернете до нього увагу користувача, але й зробите всю композицію сайту динамічнішою.

На малюнку нижче - приклад структури з акцентом (ліворуч) та структури без акценту (праворуч).

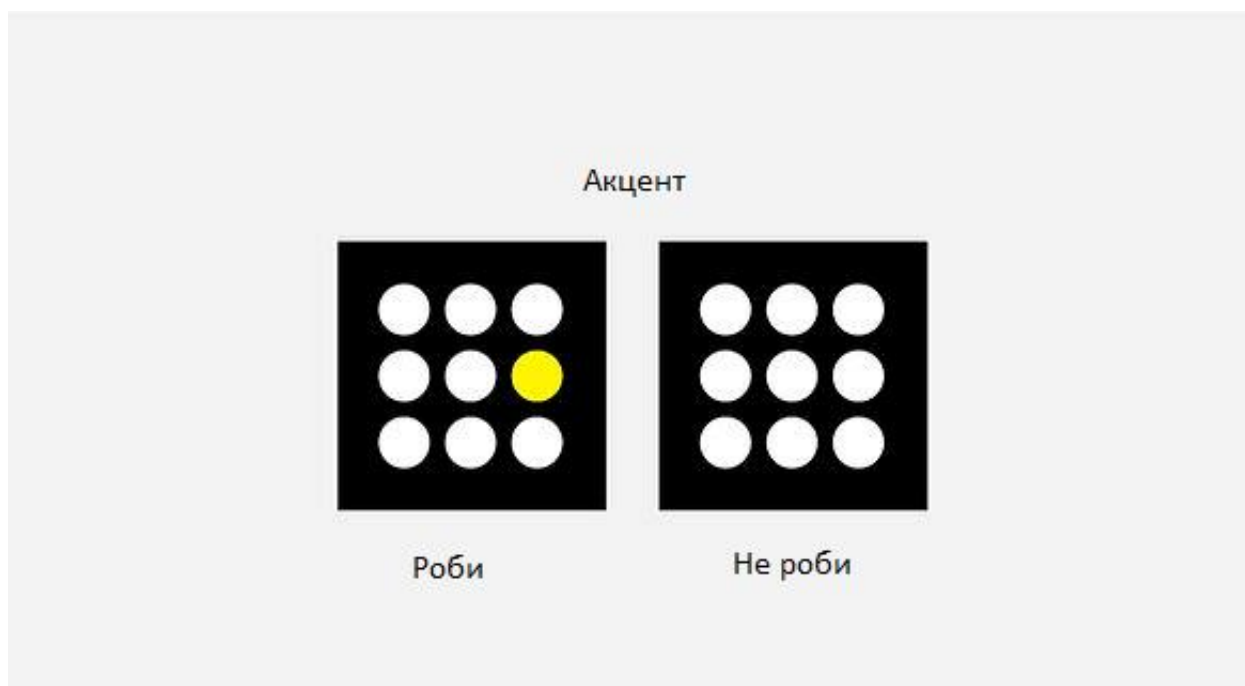


Рисунок 4.17 — Акцент

Змінюючи розмір і порядок елементів на сторінці, ви можете контролювати напрямок, яким рухається погляд читача. А значить – привернути його увагу до найбільш значущих частин сайту.

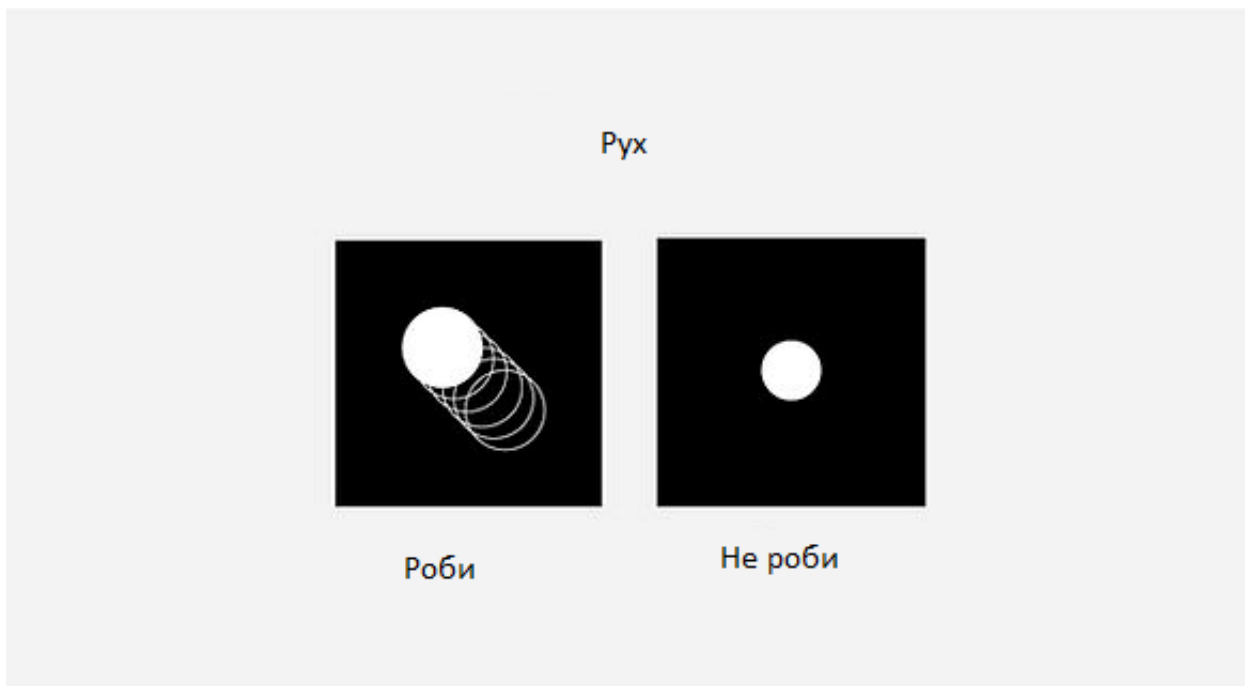


Рисунок 4.18 — Рух

Елементи на сторінці повинні "рифмуватися" один з одним - тобто, мати схожі характеристики на кшталт брендових кольорів і фірмового шрифту. Це допоможе зміцнити ідентичність бренду та підвищити онлайн присутність. На малюнку нижче приклад хорошого ритму (ліворуч) і поганого ритму (праворуч).

На прикладі справа зібрані елементи з різними характеристиками, які не поєднуються між собою та руйнують цілісність структури.

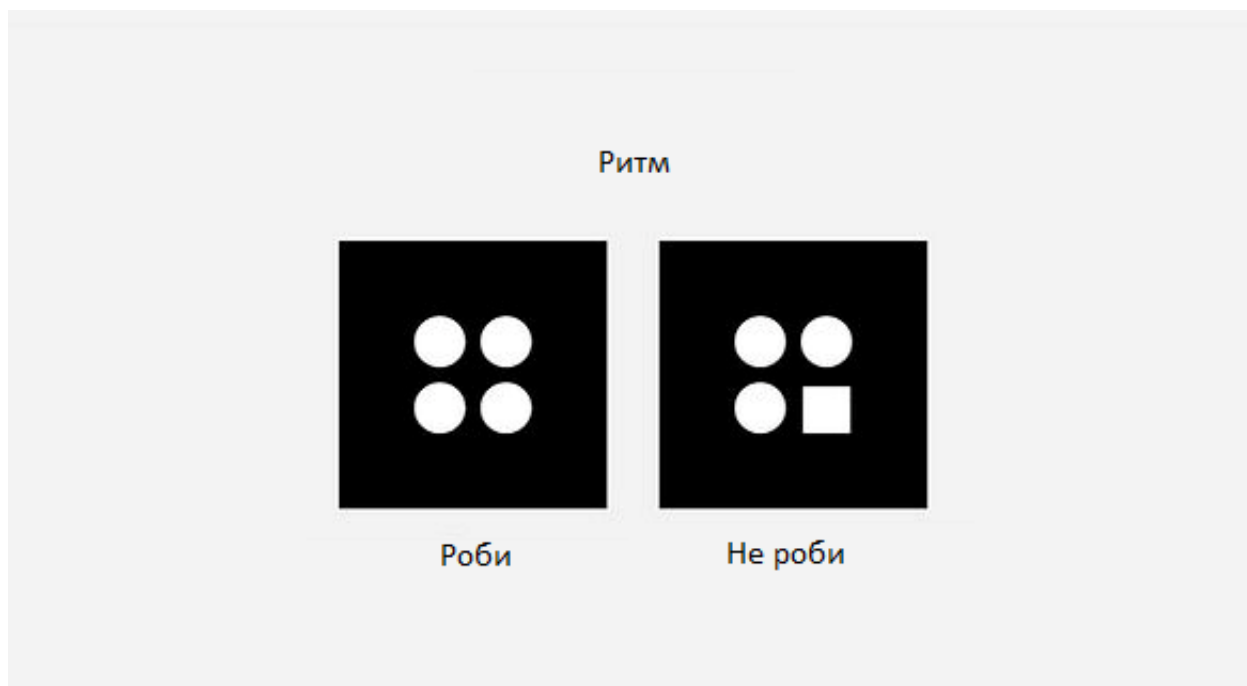


Рисунок 4.19 — Ритм

Одна з найбільш грубих помилок у веб-дизайні – розташування важливої інформації в нижній частині сторінки. Таким чином її помітять лише ті користувачі, які не залишили сайт у першу хвилину і вирішили прокрутити вниз – і таких людей буде небагато. Тому запам'ятайте просте правило: чим важливіша інформація, тим вище на сайті потрібно її розташувати. На малюнку нижче - приклад хорошої ієрархії (ліворуч) та поганої ієрархії (праворуч). На прикладі ліворуч елементи розташовані в порядку зменшення важливості, що допомагає читачеві швидко розставити пріоритети.

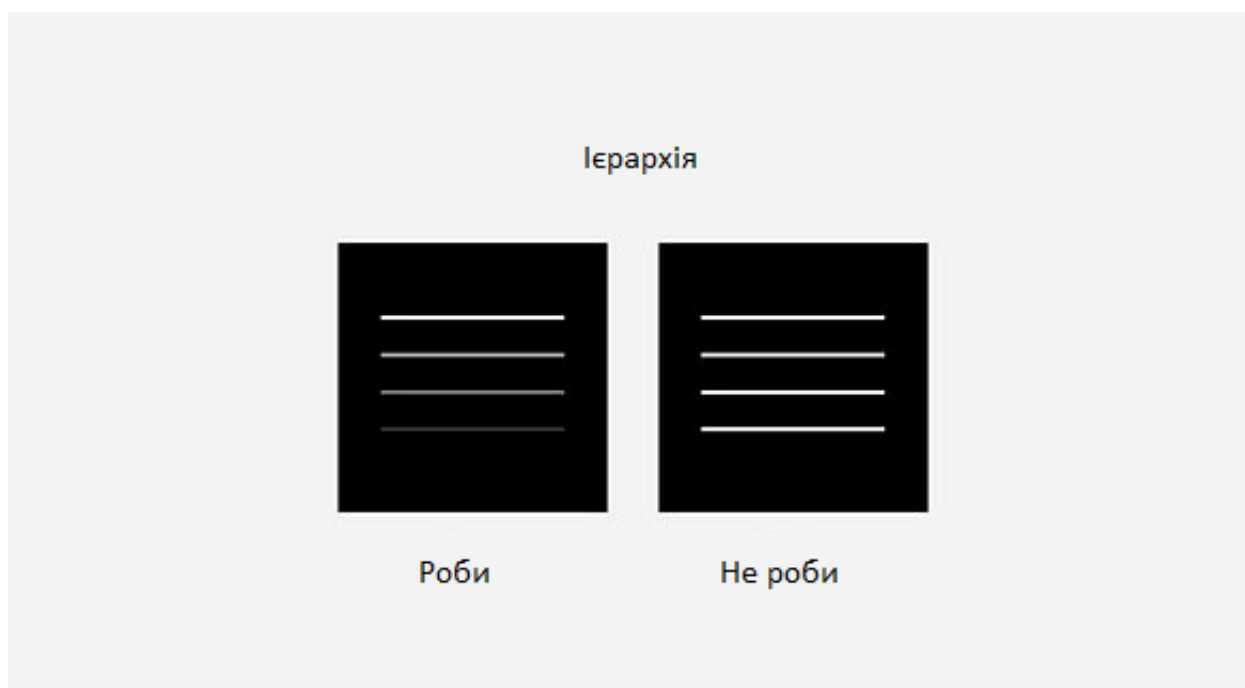


Рисунок 4.20 — Ієрархія

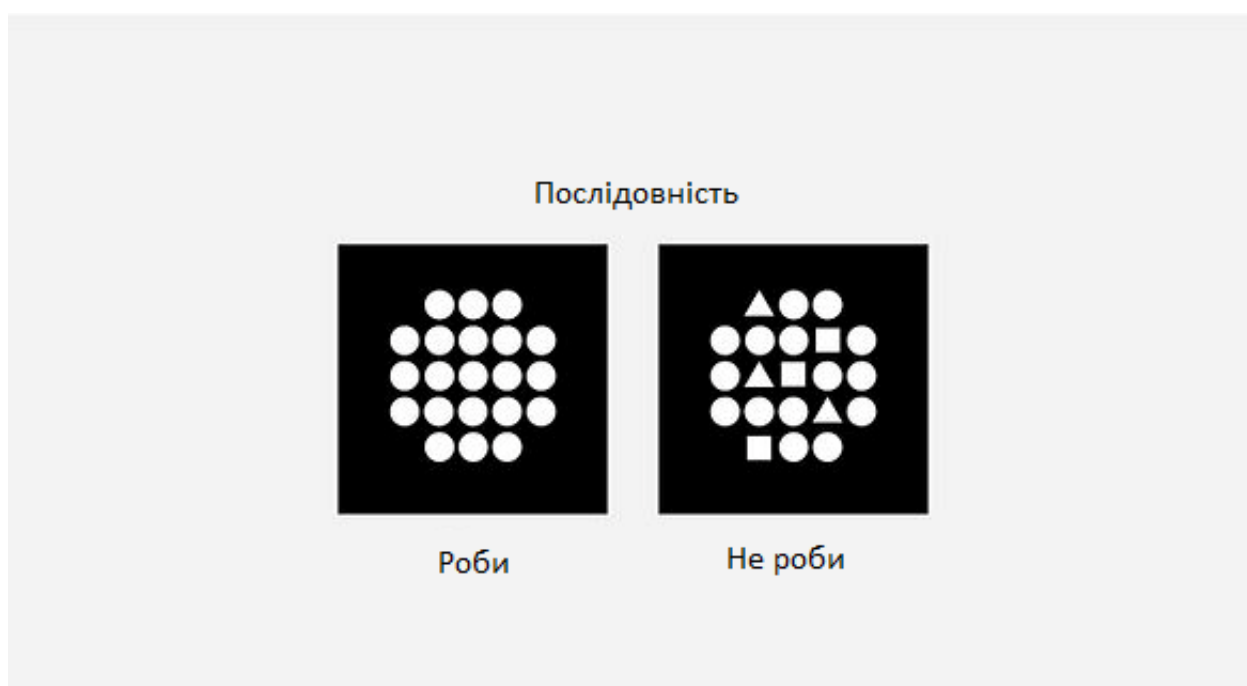
"Білий простір" або негативний простір - це ділянки сайту, вільні від будь-яких візуальних елементів. Тобто відступи, проміжки між розділами, тло і так далі. Велика кількість «повітря» потрібна будь-якому сайту — негативний простір розвантажує сайт і робить легшим для сприйняття.

На малюнку нижче - приклад правильного використання "повітря" (ліворуч) та неправильного використання "повітря" (праворуч). На прикладі праворуч елементи розташовуються дуже близько один до одного, що створює візуальний шум і виглядає занадто громіздко.



Рисунок 4.21 — «Повітря»

Послідовність веб-дизайну досягається, коли інші базові принципи дотримані і перебувають у гармонії. Коли елементи сайту збалансовані, правильно розташовані, контрастують один з одним і оточені достатньою кількістю «повітря», відвідувач з більшою ймовірністю затримається на сайті та здійснить потрібну вам дію. У послідовному веб-дизайні елементи гармонійно поєднуються між собою, створюючи єдину візуальну картину як на прикладі зліва.



## Рисунок 4.22 — Послідовність

**Функціональність і зручність використання**

Перший і найголовніший принцип веб-дизайну полягає в тому, що сайт повинен бути функціональним та зручним для користувача. Красивий дизайн, який не виконує своїх основних завдань, є марним. Функціональність передбачає, що всі елементи сайту працюють коректно, швидко завантажуються та надають користувачеві можливість легко знайти потрібну інформацію або виконати необхідні дії.

Зручність використання (юзабіліті) включає в себе інтуїтивну навігацію, зрозумілу структуру контенту, швидкість завантаження сторінок та адаптивність до різних пристроїв. Наприклад, кнопка "Купити" повинна бути помітною, мати контрастний колір відносно фону та розміщуватися в логічному місці на сторінці товару.

**Принцип простоти та мінімалізму**

Простота в веб-дизайні не означає примітивність або відсутність креативності. Навпаки, це мистецтво створення максимально ефективного інтерфейсу з мінімальною кількістю елементів. Принцип "менше – це більше" допомагає користувачам сконцентруватися на найважливішому контенті, не відволікаючись на зайві деталі.

```
/* Приклад мінімалістичного стилю для заголовка */  
.main-heading {  
    font-family: 'Arial', sans-serif;  
    font-size: 2.5rem;  
    font-weight: 300;  
    color: #333;  
    margin: 2rem 0;  
    text-align: center;  
    line-height: 1.4;  
}
```

```

/* Простий стиль для кнопки з мінімальними деталями */
.simple-button {
  background-color: #007bff;
  color: white;
  border: none;
  padding: 12px 24px;
  border-radius: 4px;
  font-size: 1rem;
  cursor: pointer;
  transition: background-color 0.3s ease;
}

.simple-button:hover {
  background-color: #0056b3;
}

```

### **Консистентність та єдність стилю**

Консистентність означає послідовне використання однакових елементів дизайну по всьому сайту. Це стосується типографіки, кольорової палітри, стилю кнопок, іконок, відступів та інших візуальних елементів. Коли користувач переходить з однієї сторінки на іншу, він повинен відчувати, що знаходиться на тому ж сайті, а не на абсолютно новому ресурсі.

### **Ієрархія та структура інформації**

Візуальна ієрархія допомагає користувачам зрозуміти, яка інформація є найважливішою, а яка – другорядною. Це досягається за допомогою різних розмірів шрифтів, кольорів, контрасту, розміщення елементів та використання білого простору. Наприклад, заголовки повинні бути більшими та помітнішими за основний текст, а найважливіші кнопки – мати більший розмір та контрастний колір.

```

/* Приклад створення візуальної ієрархії */
h1 {

```



```
    font-size: 2.5rem;
    font-weight: 700;
    color: #1a1a1a;
    margin-bottom: 1rem;
}

h2 {
    font-size: 2rem;
    font-weight: 600;
    color: #333;
    margin-bottom: 0.8rem;
}

h3 {
    font-size: 1.5rem;
    font-weight: 500;
    color: #555;
    margin-bottom: 0.6rem;
}

p {
    font-size: 1rem;
    line-height: 1.6;
    color: #666;
    margin-bottom: 1rem;
}
```

#### 4.2.2 Веб дизайн з чого почати

Перш ніж розпочинати роботу над дизайном, необхідно чітко визначити, хто є цільовою аудиторією сайту та які цілі він повинен досягати. Дизайн сайту

для молодії аудиторії буде кардинально відрізнятися від корпоративного ресурсу для ділових партнерів. Важливо зрозуміти потреби, вподобання та поведінкові особливості майбутніх користувачів.

Wireframe – це схематичне зображення майбутнього сайту, яке показує розміщення основних елементів без деталізації дизайну. Це дозволяє зосередитися на структурі та функціональності, не відволікаючись на візуальні деталі. Mockup – це більш детальне зображення з кольорами, шрифтами та реальним контентом.

```
<!-- Приклад HTML-структури для wireframe головної
сторінки -->
<!DOCTYPE html>
<html lang="uk">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <title>Wireframe - Головна сторінка</title>
  <style>
    .wireframe {
      font-family: Arial, sans-serif;
      max-width: 1200px;
      margin: 0 auto;
      padding: 20px;
    }

    .header, .nav, .hero, .content, .sidebar,
.footer {
      border: 2px solid #ccc;
      margin: 10px 0;
      padding: 20px;
```

```

        background-color: #f9f9f9;
    }

    .container {
        display: flex;
        gap: 20px;
    }

    .content {
        flex: 2;
    }

    .sidebar {
        flex: 1;
    }
</style>
</head>
<body>
    <div class="wireframe">
        <header class="header">HEADER - Логотип і
контактна інформація</header>
        <nav class="nav">NAVIGATION - Головне
меню</nav>
        <section class="hero">HERO SECTION - Головний
банер з call-to-action</section>
        <div class="container">
            <main class="content">MAIN CONTENT -
Основний контент сторінки</main>
            <aside class="sidebar">SIDEBAR - Додаткова
інформація</aside>
    </div>
</body>
</html>

```

```

        </div>
        <footer class="footer">FOOTER - Копірайт і
додаткові посилання</footer>
    </div>
</body>
</html>

```

Сучасний веб-дизайн неможливий без правильного вибору технологій. HTML5 надає семантичну структуру, CSS3 забезпечує стилізацію та анімації, JavaScript додає інтерактивність. Важливо також врахувати необхідність адаптивного дизайну, оптимізації для пошукових систем та швидкості завантаження.

Більшість дизайнерів спочатку використовують графічні редактори, а потім віддають свої проекти кодеру на верстку. Photoshop – це інструмент, що найчастіше використовується. Але дизайнери також використовують такі програми, як Sketch, GIMP, Inkscape та Illustrator. Однак, якщо ви хочете створювати саме сайти, то намагайтеся відразу ж робити це в їхньому природному середовищі - в браузері. В ідеалі – робіть це у кількох браузерах – тому що люди не переглядають сайти у Photoshop і не бачать їх так, як ви через інтерфейс графічного редактора.

Робочий процес на основі браузера має низку інших переваг:

- 1) Ви бачите, що отримуєте.

Навіть професійні мокапи не передають інтерактивні чи анімовані частини сайтів. Створюючи дизайн на основі браузера, ви зможете точно бачити, як він працює.

Це особливо актуально, коли йдеться про адаптивний дизайн. (Для непосвячених: чуйний або адаптивний дизайн – це той, який коректно відображається на різних екранах та пристроях, чи то мобільний телефон, планшет, або ПК).

- 2) Дизайн на основі браузера може зробити вас найкращим дизайнером. Коли ви самі знаєте весь процес, ви зможете зрозуміти, що і як працює, і

застерегти себе від багатьох помилок. Кошмар будь-якого дизайнера (або верстальника, якому це доведеться робити), що працює в, скажімо, Photoshop, це подальше його «натягування» на живий сайт.

Дизайн у браузері може багато в чому змінити ваш підхід до роботи і підштовхнути вас довідатися про щось нове про HTML і CSS. В принципі, чим більше ви повозитеся з брудними з кодом, тим краще ви зрозумієте, як працюють сайти. Це не зробить вас чудовим дизайнером; але це чудовий початок.

Теорія кольору тісно пов'язана із типографікою. Наприклад, якщо колір тексту занадто близький до кольору обраного фону, то його буде важко читати і відвідувачі, швидше за все, просто пройдуть повз цю частину сайту, або ж зовсім покинуть його.

Можна підібрати чудові шрифти, відмінну колірну схему, а потім банально напороти з правильним розташуванням блоків, ієрархією та пропорціями і отримати «найогиднішу страву з відмінних продуктів».

Оскільки тенденції в веб-дизайні постійно змінюються, варто стежити за сайтами для веб-дизайнерів, ось кілька хороших прикладів:

- [abduzeedo.com](http://abduzeedo.com)
- [behance.net](http://behance.net)
- [noupe.com](http://noupe.com)
- [webdesignerwall.com](http://webdesignerwall.com)
- [flickr.com](http://flickr.com)
- [awwwards.com](http://awwwards.com)

#### **4.2.3 Теорія кольору та колірні схеми**

Колір є одним із найпотужніших інструментів веб-дизайнера, який може викликати емоції, направляти увагу та створювати певний настрій. Теорія кольору базується на кольоровому колі, яке містить первинні (червоний, синій, жовтий), вторинні (зелений, помаранчевий, фіолетовий) та третинні кольори.

Основні характеристики кольору включають:

- відтінок (Hue) – власне колір на кольоровому колі;
- насиченість (Saturation) – інтенсивність кольору;
- яскравість (Brightness/Value) – кількість світла в кольорі.

Багато кольорів не люблять один одного. Звичайно, кількість кольорів практично безмежна, але лише деякі з них є дружніми. Крім того, хороших комбінацій кольорів досить мало в порівнянні з величезною кількістю відтінків.

Щоб створювати тільки гідні комбінації кольорів і використовувати поєднання кольорів в дизайні сайту, важливо зрозуміти природу та особливості кожного кольору.

Червоний, синій та жовтий називаються основними кольорами, тому що з них утворюється більшість комбінацій. Практично їх можна назвати «батьками» квітів. Вторинні кольори створюються шляхом поєднання основних кольорів, у рівних частинах:

Синій + Жовтий = Зелений

Синій + Червоний = Фіолетовий

Червоний + Жовтий = Помаранчевий

І нарешті, є кольори третього рівня, які утворюються шляхом комбінації первинно-вторинних чи вторинно-вторинних кольорів. Безперечно, це відбувається в рівних частинах. Результати виходять такі:

Синій + Зелений = Аквамарин

Синій + Фіолетовий = Ліловий

Червоний + Оранжевий = червоний

Червоний + Фіолетовий = Пурпуровий

Жовтий + Помаранчевий = Золотистий

Жовтий + Зелений = Жовто-зелений

Ще один факт, який завдає дизайнерам клопоту, це значення кольорів у різних країнах та культурах. Можливо, це прозвучить трохи дивно, але одним

людям колір може приносити радість, а в іншій частині світу може розглядатися як скорботний. Незважаючи на ці відмінності, кожен колір має свої особливі значення.

Різні кольори викликають різні емоційні реакції та асоціації:

1. Червоний асоціюється з енергією, пристрастю, терміновістю. Часто використовується для кнопок призову до дії.

Коли людина замислюється над значенням червоного кольору, у його голові відразу ж виникають асоціації з любов'ю та Днем Святого Валентина, бо більшість подарунків у це свято червоного кольору. Також червоний провокує хвилювання – тому багато кнопок виклику та дій пофарбовані в червоний колір; крім того, він підвищує кров'яний тиск. А ще червоний означає небезпеку. Вас колись цікавило, чому є дорожні знаки червоного кольору? Ось вам пояснення.

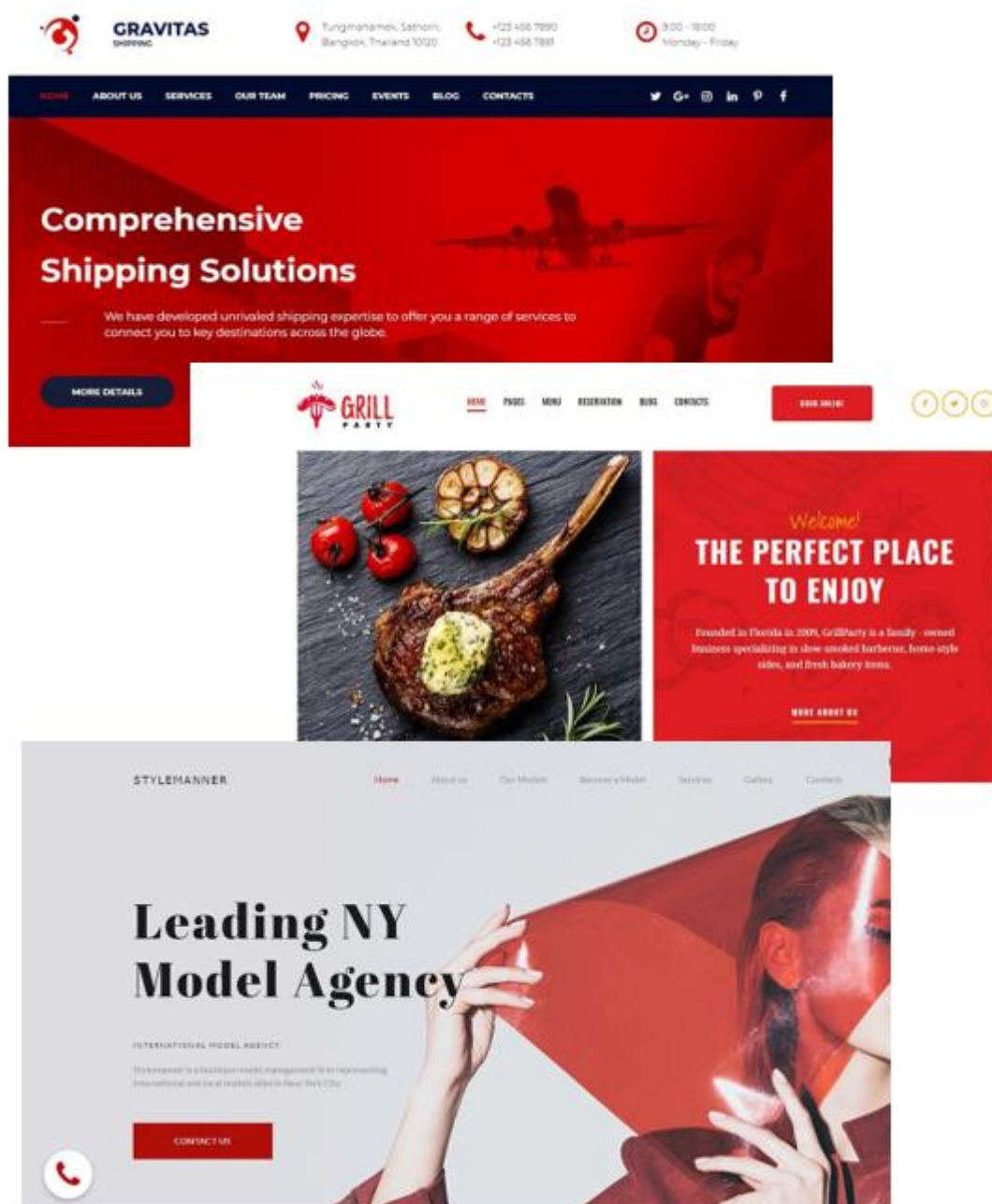


Рисунок 4.23 — Червоний дизайн

2. Синій передає довіру, професіоналізм, стабільність. Популярний у корпоративних сайтах.

Більшість дизайнерів погодиться: якщо хтось не впевнений, який колір краще використовувати, зазвичай його вибір зупиняється на синьому кольорі. Синій – це колір води та неба. Він часто використовується в дизайні та дає особливе відчуття комфорту, спокою та стабільності. Це і є головна причина, чому він чому він так поширений в палітрі кольорів фінансових веб-сайтах.



Синій колір асоціюється з довірою, правдою, і крім того, що він має особливий заспокійливий ефект, дизайнери рекомендують використовувати його як фон.

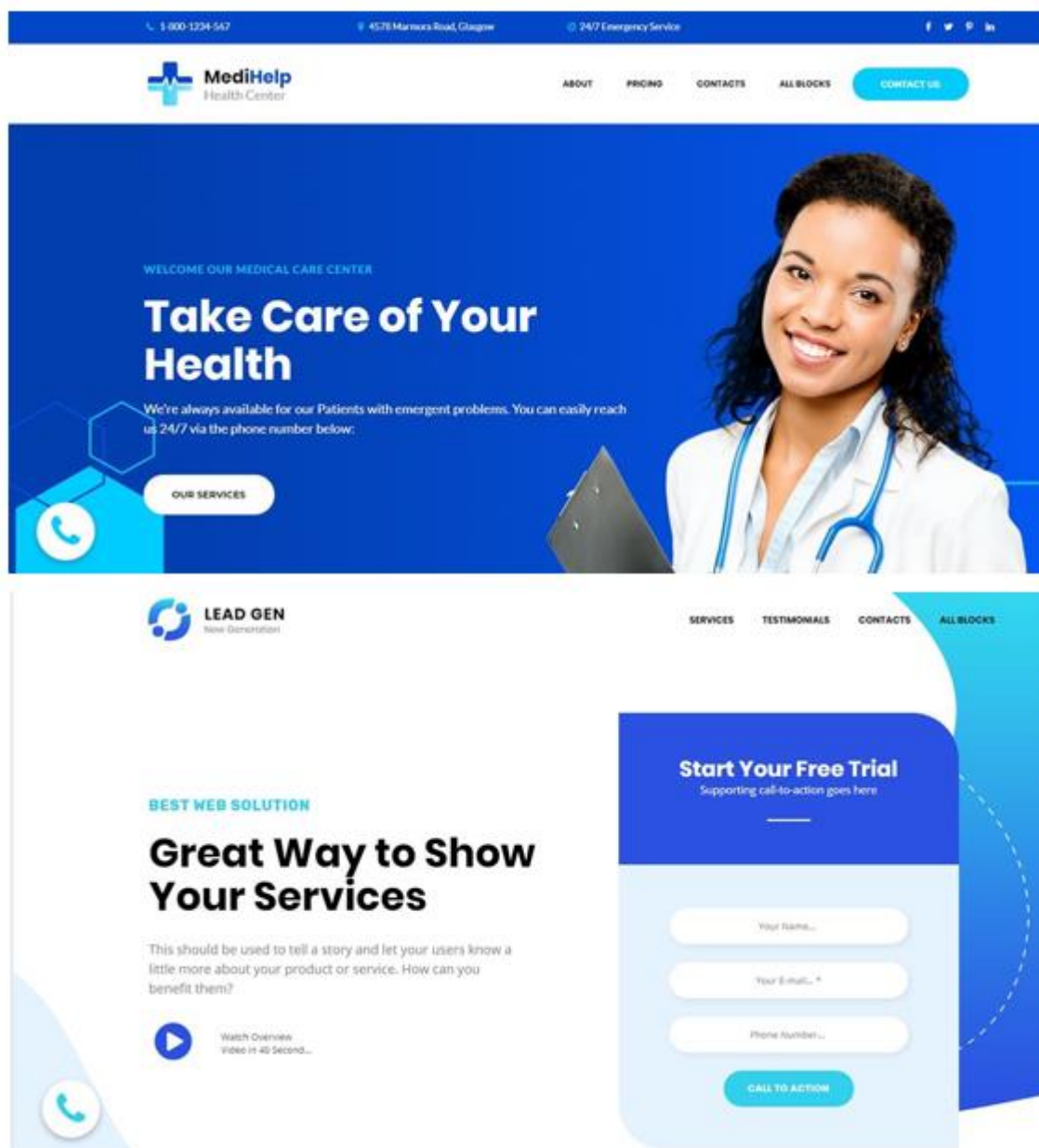


Рисунок 4.24 — Синій дизайн

3. Зелений символізує природу, зростання, успіх. Ідеальний для екологічних проектів.

В наш час дуже популярні екологічно чисті акції та продукти, які не забруднюють довкілля. І, звичайно ж, попит на веб-сайти, пов'язані з цим явищем, дуже високий. Результат дуже простий: зелений колір використовується для позначення довкілля. З іншого боку, зелений колір асоціюється зі свіжістю, молодістю і грошима (адже долари зелені).

Зверніть увагу на дивовижне використання зеленого кольору у веб-дизайні.



Рисунок 4.25 — Зелений дизайн

4. Жовтий асоціюється з радістю, оптимізмом. Привертає увагу, але може втомлювати.

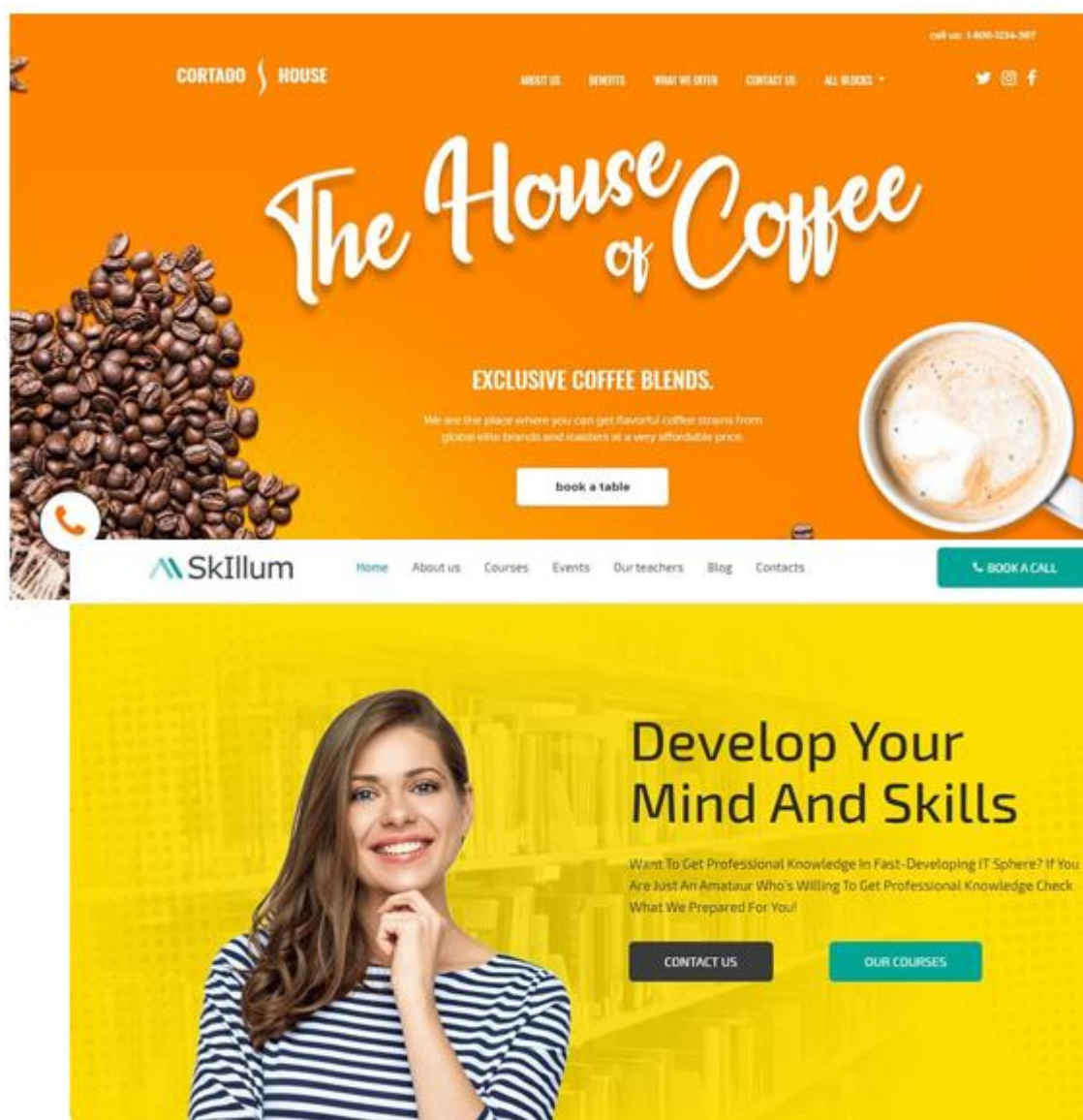


Рисунок 4.26 — Жовтий дизайн

5. Чорний означає елегантність, владу, професіоналізм. Часто використовується в преміум-брендах.

### Типи колірних схем

Фактично, палітра кольорів – це графічне зображення кольорів у формі кола. Цей факт дуже важливий, оскільки з використанням спектра кольорів, можна комбінувати відтінки між собою.

Давайте заглянемо глибше, щоб побачити, як правильно варто створювати комбінації кольорів.

Монохроматична схема використовує різні відтінки одного кольору. Така схема створює гармонійний та спокійний вигляд, але може виглядати одноманітно без достатнього контрасту.



Рисунок 4.27 — Монохроматична схема

```
/* Приклад монохроматичної синьої схеми */
:root {
  --primary-blue: #0066cc;
  --light-blue: #3385d6;
  --lighter-blue: #66a3e0;
  --dark-blue: #004499;
  --darker-blue: #002266;
}

.header {
  background-color: var(--primary-blue);
  color: white;
```

```
}  
  
.sidebar {  
  background-color: var(--lighter-blue);  
  color: var(--darker-blue);  
}  
  
.button {  
  background-color: var(--dark-blue);  
  border: 2px solid var(--darker-blue);  
}
```

Аналогова (або аналогічна) колірна схема - це поєднання кольорів, які розташовані поруч один з одним на кольоровому колі. Зазвичай використовують 3-5 сусідніх кольорів.

Чому вона виглядає природно? В природі аналогові кольори зустрічаються разом:

- Захід сонця: від жовтого через помаранчевий до червоного
- Океан: від синього через синьо-зелений до зеленого
- Осіннє листя: від зеленого через жовто-зелений до жовтого

Психологічні причини:

- Наше око звикло бачити плавні переходи кольорів
- Немає різкого контрасту, що заспокоює зір
- Створює відчуття гармонії та єдності





Рисунок 4.28 — Аналогова схема

Комплементарна (або контрастна) колірна схема використовує два кольори, які знаходяться точно навпроти один одного на кольоровому колі. Ці кольори називаються "комплементарними" або "доповнюючими".

Основні комплементарні пари:

- Червоний ↔ Зелений
- Синій ↔ Помаранчевий
- Жовтий ↔ Фіолетовий

Чому створюється високий контраст?

Фізичні причини:

- Комплементарні кольори мають максимальну різницю у довжині хвиль світла
- Коли вони поруч, око сприймає їх як особливо яскраві
- Один колір "підсилює" інший через оптичний ефект

Психологічні ефекти:

- Створюють відчуття динаміки та енергії
- Привертають максимальну увагу
- Можуть викликати напругу або збудження



Рисунок 4.29 — Комплементарна схема

Тріадна колірна схема використовує три кольори, які розташовані на рівній відстані один від одного на кольоровому колі. Якщо уявити кольорове коло як годинник, то ці кольори знаходяться на відстані  $120^\circ$  (4 години) один від одного.

Основні тріадні комбінації:

- Первинні кольори: Червоний — Синій — Жовтий
- Вторинні кольори: Помаранчевий — Зелений — Фіолетовий
- Третинні кольори: Червоно-помаранчевий — Синьо-зелений — Жовто-зелений

Чому це працює?

Математична гармонія:

- $360^\circ \div 3 = 120^\circ$  — ідеальний баланс
- Створює рівносторонній трикутник на кольоровому колі
- Жоден колір не домінує візуально

Психологічний ефект:

- Достатньо контрасту для енергії
- Достатньо гармонії для комфорту
- Створює живий, але збалансований вигляд



Рисунок 4.30 — Тріадна схема

Тетрадічна комбінація складається із чотирьох кольорів. Тут вибираються два сусідні кольори, і на додаток до них – два протилежні у кольоровому колі.





Рисунок 4.31 — Тетрадічна схема

#### 4.2.4 Як використовувати поєднання кольорів у дизайні сайту

##### 1. Правило 60-30-10

Це класичне правило дизайну стверджує, що в колірній схемі повинен домінувати один колір (60%), другий колір використовуватися як додатковий (30%), а третій – як акцентний (10%). Таке співвідношення створює збалансований та гармонійний дизайн.

```
/* Приклад використання правила 60-30-10 */
body {
    background-color: #f8f9fa; /* 60% - домінуючий
світло-сірий */
    color: #333; /* основний текст */
}

.sidebar, .card {
```

```

    background-color: #e9ecf; /* 30% - додатковий
сірий */
}

.button-primary, .link-active {
    background-color: #007bff; /* 10% - акцентний синій
*/
    color: white;
}

.highlight, .notification {
    background-color: #ffc107; /* додатковий акцентний
жовтий */
}

```

## 2. Створення контрасту для читабельності

Контраст між текстом та фоном є критично важливим для читабельності. Згідно з рекомендаціями WCAG, коефіцієнт контрасту для звичайного тексту повинен бути не менше 4.5:1, а для великого тексту – 3:1.

```

/* Приклади правильного контрасту */
.good-contrast {
    background-color: #ffffff;
    color: #333333; /* Коефіцієнт контрасту: 12.63:1 */
}

.acceptable-contrast {
    background-color: #0066cc;
    color: #ffffff; /* Коефіцієнт контрасту: 4.56:1 */
}

/* Приклад поганого контрасту (уникати!) */

```

```
.bad-contrast {
    background-color: #cccccc;
    color: #999999; /* Коефіцієнт контрасту: 2.93:1 –
занадто низький */
}
```

### 3. Використання градієнтів

Градiente можуть додати глибину та сучасність дизайну, але їх слід використовувати обережно. Тонкі градієнти виглядають більш професійно, ніж яскраві та контрастні.

```
/* Приклади ефективного використання градієнтів */
.subtle-gradient {
    background: linear-gradient(135deg, #667eea 0%,
#764ba2 100%);
    color: white;
    padding: 40px;
    text-align: center;
}

.card-gradient {
    background: linear-gradient(to bottom, #ffffff 0%,
#f8f9fa 100%);
    border: 1px solid #dee2e6;
    border-radius: 8px;
    padding: 20px;
    box-shadow: 0 2px 4px rgba(0,0,0,0.1);
}

.button-gradient {
    background: linear-gradient(to bottom, #28a745 0%,
#1e7e34 100%);
```

```

border: none;
color: white;
padding: 10px 20px;
border-radius: 4px;
transition: all 0.3s ease;
}

.button-gradient:hover {
    background: linear-gradient(to bottom, #34ce57 0%,
#28a745 100%);
}

```

В даний час в Інтернеті є чимало ресурсів, завдяки яким можна поліпшити свої навички в будь-якій галузі веб-дизайну, в тому числі і теорії кольору. Крім того, деякі веб-додатки є відмінними інструментами, які роблять процес вибору колірних комбінацій легшим. Ось тільки деякі з них, які можуть виявитися вам корисними. Ви можете знайти й інші, на власний смак.

#### #1 – Canva Color Wheel

Це прекрасний додаток зі зручним інтерфейсом. Він простий у використанні та інтуїтивно зрозумілий.

#### #2 – Adobe Color Wheel

Ця програма, як і попередня, має багато функцій. Отже, якщо вам не сподобалося Canva, це стане дуже надійною альтернативою. Додаток дозволяє створювати різні колірні комбінації та побачити, як кольори відповідають один одному.

#### #3 – Colors Co

Це ще одна не менш вражаюча програма, та його головною особливістю є швидкий підбір колірних схем.

## 4.2.5 Основи композиції сайтів

Щодо композиції у дизайні є кілька теорій, з якими ми познайомимосся.

## Єдиний візуальний образ

В основі цього методу композиції лежить одна привертає увагу картинка, навколо якої будується дизайн. Найчастіше це можна побачити у поліграфії, рідше – у веб-дизайні.

Єдиний візуальний образ – це найпростіший спосіб досягти потрібного результату. Зоровий образ виконуватиме більшу частину роботи. Тут дуже важливо, щоб решта елементів дизайну підтримувала картинку, а не конфліктувала з нею.

Ось чудові приклади цього методу.



Рисунок 4.32 — Єдиний візуальний образ

## Золотий перетин

Золотий перетин, також відомий як Божественна пропорція, золота спіраль тощо, дорівнює приблизно 1:1.618. Це співвідношення є у всіх аспектах життя, від будови кісток людини до спірального розташування насіння соняшника та завитків раковин молюсків, вона лежить в основі всіх біологічних структур і здається геометричною схемою життя.

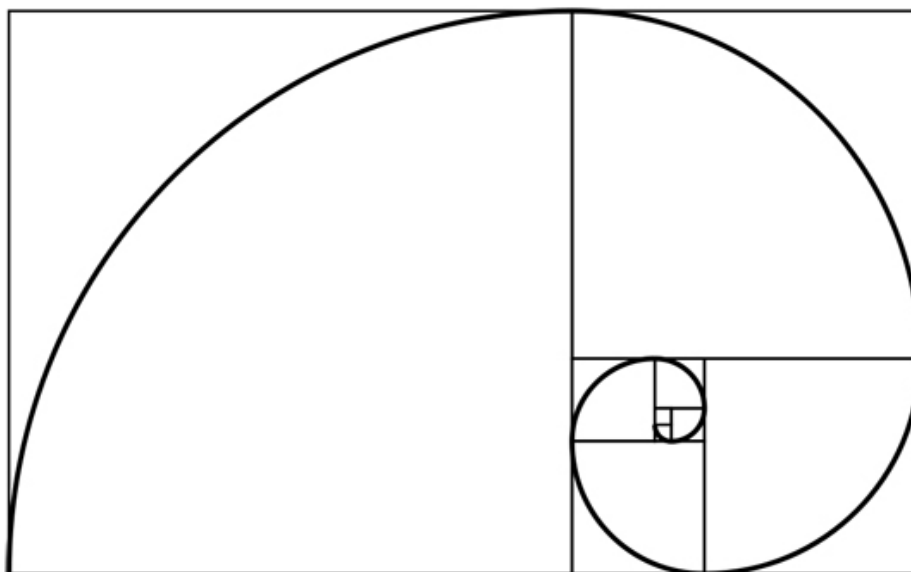


Рисунок 4.33 — Золотий перетин

```

/* Використання золотого перерізу в пропорціях */
.golden-ratio-layout {
    display: grid;
    grid-template-columns: 1fr 1.618fr; /* Золотий
переріз */
    gap: 30px;
    min-height: 100vh;
}

.sidebar {
    /* Менша частина - 1 */
    background-color: #f8f9fa;
    padding: 40px 30px;
}

.main-content {
    /* Більша частина - 1.618 */
    padding: 40px;
}

```

```
/* Золотий переріз у розмірах елементів */
.hero-section {
    height: 61.8vh; /* 61.8% від висоти екрану */
}

.content-section {
    height: 38.2vh; /* 38.2% від висоти екрану */
}
```

### **Правило третьої**

Правило третій іноді плутають із золотою спіраллю, але це не одне й те саме. Співвідношення тут 1:1.667, і, в цілому, це просто спрощений варіант золотого перерізу, що використовується фотографами.

Під час визначення зорових центрів кадр ділиться лініями, паралельними його сторонам, у пропорціях 3:5, 2:3 чи 1:2.

На перетині цих ліній повинні розташовуватися об'єкти композиції.

### **Фокальна точка**

Фокальна точка дає глядачам щось, що можна дивитися. Грубо кажучи, це напрямна сила в дизайні, хоча може виконувати роль громовідводу. Кожен дизайн повинен містити в собі такий об'єкт(и) - це може бути типографіка, кнопки, картинки або що-небудь інше. Ця точка повинна заодно виконувати цілі дизайну. Наприклад, у інтернет-магазину фокальною точкою може бути кнопка "купити".

### **Сітка**

Сітка – один із найвідоміших прийомів дизайнерів, який дозволяє створювати рівні та пропорційні макети. Існує безліч видів сіток і ресурсів, присвячених їм.

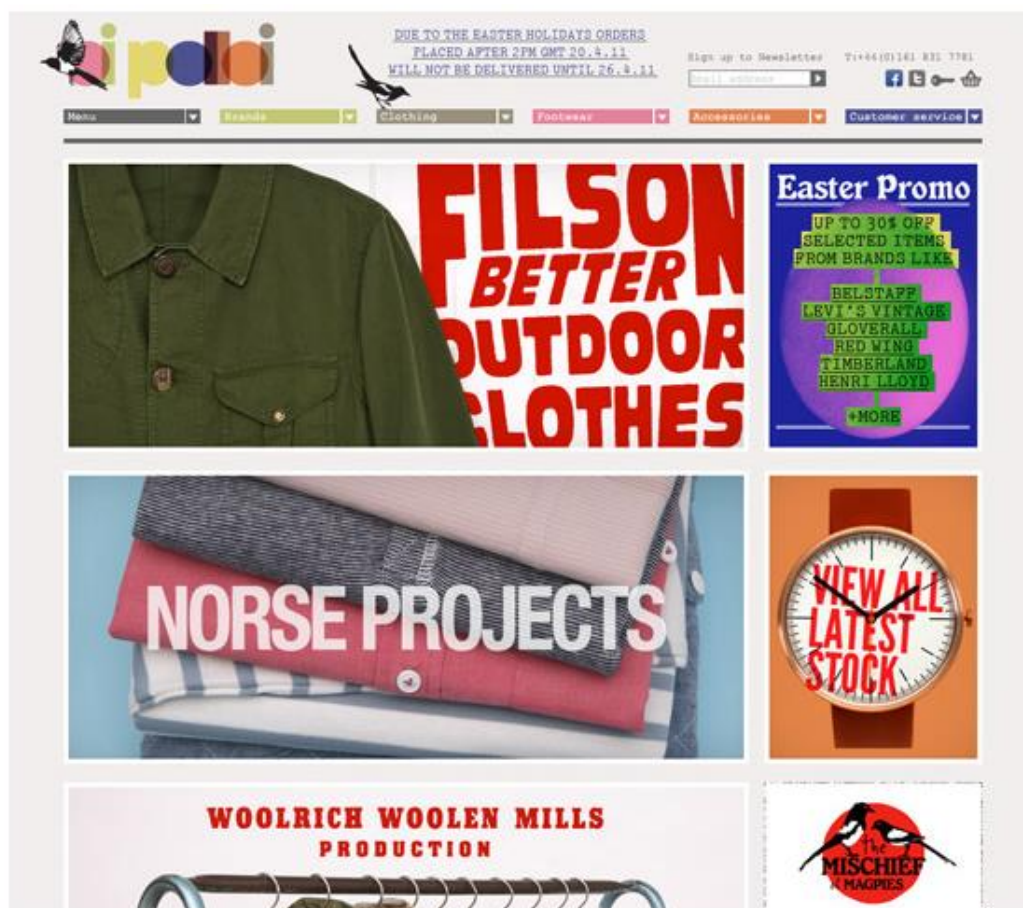


Рисунок 4.34 — Сітка

Сітка є основою організації контенту на веб-сторінці. Вона забезпечує структуру, вирівнювання та консистентність. Найпоширенішою є 12-колонкова сітка, яка дозволяє гнучко організовувати контент для різних розмірів екранів.

```
/* Приклад створення сітки CSS Grid */
.grid-container {
  display: grid;
  grid-template-columns: repeat(12, 1fr);
  grid-gap: 20px;
  max-width: 1200px;
  margin: 0 auto;
  padding: 0 20px;
}
```



```

.col-1 { grid-column: span 1; }
.col-2 { grid-column: span 2; }
.col-3 { grid-column: span 3; }
.col-4 { grid-column: span 4; }
.col-6 { grid-column: span 6; }
.col-8 { grid-column: span 8; }
.col-12 { grid-column: span 12; }

/* Адаптивність сітки */
@media (max-width: 768px) {
    .grid-container {
        grid-template-columns: repeat(6, 1fr);
        grid-gap: 15px;
    }

    .col-8 { grid-column: span 6; }
    .col-4 { grid-column: span 6; }
}

@media (max-width: 480px) {
    .grid-container {
        grid-template-columns: 1fr;
        grid-gap: 10px;
    }

    .col-1, .col-2, .col-3, .col-4, .col-6, .col-8,
.col-12 {
        grid-column: span 1;
    }
}

```

Білий простір – це не просто "порожнє місце" на сторінці, а активний елемент дизайну, який допомагає організувати контент, створити фокус на важливих елементах та покращити читабельність. Правильне використання білого простору робить дизайн більш елегантним та професійним.

```
/* Приклад ефективного використання білого простору */
.content-section {
    padding: 80px 0; /* Великі відступи між секціями */
    margin-bottom: 40px;
}

.article-text {
    line-height: 1.8; /* Достатні міжрядкові відступи */
    margin-bottom: 1.5rem; /* Відступи між абзацами */
    max-width: 65ch; /* Обмеження ширини для кращої
читабельності */
}

.card-list {
    display: grid;
    grid-template-columns: repeat(auto-fit,
minmax(300px, 1fr));
    gap: 30px; /* Простір між картками */
    padding: 40px 0;
}

.card {
    padding: 30px; /* Внутрішній простір карток */
    background: white;
    border-radius: 8px;
```

```
box-shadow: 0 2px 10px rgba(0,0,0,0.1);
}
```

Дослідження показують, що користувачі сканують веб-сторінки у вигляді літери F: спочатку горизонтально вгорі сторінки, потім коротше горизонтально трохи нижче, а далі вертикально вліво. Розуміння цього патерну допомагає розміщувати найважливішу інформацію в правильних місцях.

```
<!-- HTML-структура, оптимізована під F-подібне
сканування -->
<article class="article-layout">
  <header class="article-header">
    <h1>Важливий заголовок статті</h1> <!-- Перша
горизонтальна лінія -->
    <p class="article-meta">Дата публікації • Автор
• Час читання</p>
  </header>

  <div class="article-content">
    <p class="lead">
      <strong>Важливе перше речення</strong> з
ключовою інформацією... <!-- Друга горизонтальна лінія
-->
    </p>

    <p>Перший абзац з важливими словами на початку
речень...</p>

    <h2>Підзаголовок</h2> <!-- Третя точка фокусу -
->
    <p>Продовження контенту...</p>
```

```

        <ul class="key-points"> <!-- Ліва колонка для
сканування -->
            <li><strong>Ключовий пункт 1:</strong>
опис</li>
            <li><strong>Ключовий пункт 2:</strong>
опис</li>
            <li><strong>Ключовий пункт 3:</strong>
опис</li>
        </ul>
    </div>
</article>

```

#### 4.2.6 Принципи гештальту

##### Принцип близькості (Proximity)

Елементи, розташовані близько один до одного, сприймаються як пов'язані між собою. Цей принцип допомагає групувати споріднену інформацію та створювати логічну структуру контенту.

```

/* Демонстрація принципу близькості */
.contact-group {
    margin-bottom: 40px; /* Відокремлення від інших
груп */
}

.contact-group h3 {
    margin-bottom: 15px; /* Близько до своєї групи */
}

.contact-group p,
.contact-group .contact-item {

```

```
        margin-bottom: 8px; /* Тісне групування споріднених
елементів */
    }

    .contact-item:last-child {
        margin-bottom: 0;
    }

/* Форма з групуванням за близькістю */
.form-group {
    margin-bottom: 25px; /* Відстань між групами полів
*/
}

.form-group label {
    display: block;
    margin-bottom: 5px; /* Близько до свого поля */
}

.form-group input {
    margin-bottom: 3px; /* Дуже близько до підказки */
}

.form-group .help-text {
    font-size: 0.9em;
    color: #666;
    margin-bottom: 0; /* Частина тієї ж групи */
}
```

### **Принцип подібності (Similarity)**

Елементи з подібними візуальними характеристиками (колір, форма, розмір, текстура) сприймаються як пов'язані. Цей принцип дозволяє створювати візуальні зв'язки між елементами без їх фізичного групування.

```
/* Використання подібності для створення зв'язків */
.navigation-menu {
    display: flex;
    list-style: none;
    padding: 0;
}

.navigation-menu a {
    padding: 12px 20px;
    text-decoration: none;
    color: #666;
    border-radius: 4px;
    transition: all 0.3s ease;
    /* Всі елементи меню мають подібний стиль */
}

.navigation-menu a:hover,
.navigation-menu a.active {
    background-color: #007bff;
    color: white;
    /* Активні елементи мають подібний стиль між собою */
}

/* Подібність для категоризації карток */
.product-card {
    background: white;
```

```

border-radius: 8px;
padding: 20px;
box-shadow: 0 2px 8px rgba(0,0,0,0.1);
/* Базовий стиль для всіх карток товарів */
}

.product-card.premium {
    border-left: 4px solid #ffd700;
    /* Преміум товари мають золотий акцент */
}

.product-card.sale {
    border-left: 4px solid #dc3545;
    /* Товари зі знижкою мають червоний акцент */
}

.product-card.new {
    border-left: 4px solid #28a745;
    /* Нові товари мають зелений акцент */
}

```

### Принцип замкнутості (Closure)

Мозок має тенденцію "добудовувати" неповні форми до завершених образів. Цей принцип дозволяє створювати елегантні дизайни з мінімальними засобами, залишаючи простір для уяви користувача.

```

/* Використання принципу замкнутості */
.logo-minimal {
    position: relative;
    width: 60px;
    height: 60px;
}

```

```
/* Створення логотипу з неповних елементів */
.logo-minimal::before,
.logo-minimal::after {
    content: '';
    position: absolute;
    border: 3px solid #007bff;
}

.logo-minimal::before {
    top: 0;
    left: 0;
    width: 30px;
    height: 30px;
    border-bottom: none;
    border-right: none;
}

.logo-minimal::after {
    bottom: 0;
    right: 0;
    width: 30px;
    height: 30px;
    border-top: none;
    border-left: none;
}

/* Карточка з ефектом неповноти */
.card-open {
    background: white;
```



```

border-radius: 12px 12px 0 0;
padding: 30px;
position: relative;
/* Верх картки закритий, низ "відкритий" */
}

.card-open::after {
  content: '';
  position: absolute;
  bottom: -10px;
  left: 50%;
  transform: translateX(-50%);
  width: 0;
  height: 0;
  border-left: 15px solid transparent;
  border-right: 15px solid transparent;
  border-top: 10px solid white;
  /* Трикутник, що "завершує" форму картки */
}

```

### Принцип неперервності (Continuity)

Око має тенденцію слідувати плавним лініям та кривим. Цей принцип допомагає направляти погляд користувача по сторінці та створювати відчуття руху й динаміки.

```

/* Створення візуальних ліній для направлення погляду
*/

.hero-section {
  background: linear-gradient(135deg, #667eea 0%,
#764ba2 100%);
  position: relative;
  overflow: hidden;

```

```

}

/* Діагональні елементи для створення руху */
.hero-section::before {
    content: '';
    position: absolute;
    top: -50%;
    right: -10%;
    width: 200%;
    height: 200%;
    background: rgba(255,255,255,0.1);
    transform: rotate(15deg);
    /* Діагональна лінія, що веде погляд */
}

/* Плавний потік елементів */
.feature-flow {
    display: flex;
    align-items: center;
    position: relative;
}

.feature-item {
    flex: 1;
    text-align: center;
    position: relative;
}

.feature-item:not(:last-child)::after {
    content: '→';

```

```

position: absolute;
right: -20px;
top: 50%;
transform: translateY(-50%);
color: #007bff;
font-size: 1.5rem;
/* Стрілки створюють неперервну лінію між
елементами */
}

/* Плавна навігація хлібними крихтами */
.breadcrumb {
    display: flex;
    align-items: center;
    font-size: 0.9rem;
    color: #666;
}

.breadcrumb-item:not(:last-child)::after {
    content: '/';
    margin: 0 10px;
    color: #ccc;
    /* Слеші створюють візуальну неперервність */
}

```

### **Принцип фігури та фону (Figure-Ground)**

Принцип фігури та фону описує здатність людського ока розрізнити об'єкт (фігуру) та його оточення (фон). У веб-дизайні цей принцип є критично важливим для створення ієрархії та фокусування уваги користувача на найважливіших елементах.

```

/* Створення чіткого розрізнення фігури та фону */

```

```
.hero-content {
    background: rgba(0, 0, 0, 0.7);
    color: white;
    padding: 60px 40px;
    border-radius: 12px;
    position: relative;
    z-index: 2;
    /* Контентна область як фігура */
}

.hero-background {
    position: absolute;
    top: 0;
    left: 0;
    width: 100%;
    height: 100%;
    background-image: url('hero-bg.jpg');
    background-size: cover;
    background-position: center;
    filter: blur(2px);
    z-index: 1;
    /* Розмитий фон для контрасту */
}

/* Модальне вікно з чітким розрізненням */
.modal-overlay {
    position: fixed;
    top: 0;
    left: 0;
    width: 100vw;
```

```
height: 100vh;
background: rgba(0, 0, 0, 0.8);
z-index: 1000;
/* Темний фон затемнює все інше */
}

.modal-content {
  position: absolute;
  top: 50%;
  left: 50%;
  transform: translate(-50%, -50%);
  background: white;
  padding: 40px;
  border-radius: 16px;
  box-shadow: 0 20px 60px rgba(0, 0, 0, 0.3);
  z-index: 1001;
  /* Білий модал як чітка фігура */
}

/* Картки з ефектом підняття */
.product-card {
  background: white;
  border-radius: 12px;
  padding: 24px;
  transition: all 0.3s ease;
  box-shadow: 0 2px 12px rgba(0, 0, 0, 0.08);
}

.product-card:hover {
  transform: translateY(-8px);
```

```

    box-shadow: 0 12px 40px rgba(0, 0, 0, 0.15);
    /* При наведенні картка стає більш вираженою
    фігурою */
}

```

### Принцип спільної долі (Common Fate)

Елементи, що рухаються в одному напрямку або з однаковою швидкістю, сприймаються як група. У веб-дизайні цей принцип реалізується через анімації та інтерактивні ефекти.

```

/* Групування елементів через спільну анімацію */
.team-grid {
    display: grid;
    grid-template-columns: repeat(auto-fit,
minmax(250px, 1fr));
    gap: 30px;
}

.team-member {
    opacity: 0;
    transform: translateY(30px);
    transition: all 0.6s ease;
}

/* JavaScript додає клас при появі в області видимості
*/
.team-member.animate-in {
    opacity: 1;
    transform: translateY(0);
}

/* Затримка анімації для хвильового ефекту */

```

```

.team-member:nth-child(1) { transition-delay: 0.1s; }
.team-member:nth-child(2) { transition-delay: 0.2s; }
.team-member:nth-child(3) { transition-delay: 0.3s; }
.team-member:nth-child(4) { transition-delay: 0.4s; }

/* Спільний hover-ефект для споріднених елементів */
.feature-list:hover .feature-item:not(:hover) {
    opacity: 0.6;
    transform: scale(0.98);
    /* Всі елементи, крім активного, реагують однаково */
}

.feature-item {
    transition: all 0.3s ease;
}

/* Анімація завантаження для групи елементів */
@keyframes slideInFromLeft {
    from {
        transform: translateX(-100px);
        opacity: 0;
    }
    to {
        transform: translateX(0);
        opacity: 1;
    }
}

.navigation-menu li {

```

```

    animation: slideInFromLeft 0.6s ease forwards;
    opacity: 0;
}

.navigation-menu li:nth-child(1) { animation-delay:
0.1s; }
.navigation-menu li:nth-child(2) { animation-delay:
0.2s; }
.navigation-menu li:nth-child(3) { animation-delay:
0.3s; }
.navigation-menu li:nth-child(4) { animation-delay:
0.4s; }

```

#### 4.2.7 Практичне застосування принципів композиції

##### Створення ефективної головної сторінки

Головна сторінка є візитівкою сайту і повинна миттєво передавати його призначення та цінність. Розглянемо практичний приклад створення головної сторінки з використанням всіх описаних принципів.

```

<!DOCTYPE html>
<html lang="uk">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>TechSolutions - Інноваційні рішення для
бізнесу</title>
    <style>
        /* Основні змінні кольорів згідно з правилом
60-30-10 */
        :root {

```



```

        --primary-color: #2563eb; /* 10% -
акцентний синій */
        --secondary-color: #f8fafc; /* 60% -
домінантний світло-сірий */
        --accent-color: #1e293b; /* 30% -
додатковий темно-сірий */
        --text-primary: #1e293b;
        --text-secondary: #64748b;
        --white: #ffffff;
    }

    /* Основні стилі */
    * {
        margin: 0;
        padding: 0;
        box-sizing: border-box;
    }

    body {
        font-family: 'Inter', -apple-system,
BlinkMacSystemFont, sans-serif;
        line-height: 1.6;
        color: var(--text-primary);
        background-color: var(--white);
    }

    /* Сітка контейнера */
    .container {
        max-width: 1200px;
        margin: 0 auto;
    }

```

```
padding: 0 20px;
}

/* Заголовок сайту */
.header {
    background: var(--white);
    padding: 20px 0;
    box-shadow: 0 1px 3px rgba(0,0,0,0.1);
    position: sticky;
    top: 0;
    z-index: 100;
}

.header .container {
    display: flex;
    justify-content: space-between;
    align-items: center;
}

.logo {
    font-size: 1.5rem;
    font-weight: 700;
    color: var(--primary-color);
}

/* Навігація з принципом подібності */
.nav ul {
    display: flex;
    list-style: none;
    gap: 30px;
```

```
}

.nav a {
    text-decoration: none;
    color: var(--text-secondary);
    font-weight: 500;
    transition: color 0.3s ease;
    position: relative;
}

.nav a:hover,
.nav a.active {
    color: var(--primary-color);
}

/* Принцип неперервності в навігації */
.nav a::after {
    content: '';
    position: absolute;
    bottom: -5px;
    left: 0;
    width: 0;
    height: 2px;
    background: var(--primary-color);
    transition: width 0.3s ease;
}

.nav a:hover::after,
.nav a.active::after {
    width: 100%;
}
```

```

    }

    /* Hero-секція з золотим перерізом */
    .hero {
        background: linear-gradient(135deg, var(--
primary-color) 0%, #1e40af 100%);
        color: var(--white);
        padding: 120px 0;
        position: relative;
        overflow: hidden;
    }

    /* Принцип неперервності в hero-секції */
    .hero::before {
        content: '';
        position: absolute;
        top: -50%;
        right: -10%;
        width: 200%;
        height: 200%;
        background: rgba(255,255,255,0.05);
        transform: rotate(15deg);
    }

    .hero-content {
        display: grid;
        grid-template-columns: 1.618fr 1fr; /*
Золотий переріз */
        gap: 60px;
        align-items: center;
    }

```

```
        position: relative;
        z-index: 2;
    }

    .hero-text h1 {
        font-size: 3.5rem;
        font-weight: 800;
        line-height: 1.2;
        margin-bottom: 24px;
    }

    .hero-text p {
        font-size: 1.25rem;
        line-height: 1.8;
        margin-bottom: 40px;
        opacity: 0.9;
    }

    /* Кнопки з принципом подібності */
    .btn {
        display: inline-block;
        padding: 16px 32px;
        border-radius: 8px;
        text-decoration: none;
        font-weight: 600;
        transition: all 0.3s ease;
        border: 2px solid transparent;
    }

    .btn-primary {
```

```
        background: var(--white);
        color: var(--primary-color);
    }

    .btn-primary:hover {
        background: var(--secondary-color);
        transform: translateY(-2px);
        box-shadow: 0 8px 25px rgba(0,0,0,0.15);
    }

    .btn-secondary {
        background: transparent;
        color: var(--white);
        border-color: var(--white);
        margin-left: 16px;
    }

    .btn-secondary:hover {
        background: var(--white);
        color: var(--primary-color);
    }

    /* Секція переваг з принципом близькості */
    .features {
        padding: 120px 0;
        background: var(--secondary-color);
    }

    .section-title {
        text-align: center;
```

```
        margin-bottom: 80px;
    }

    .section-title h2 {
        font-size: 2.5rem;
        font-weight: 700;
        margin-bottom: 16px;
        color: var(--accent-color);
    }

    .section-title p {
        font-size: 1.125rem;
        color: var(--text-secondary);
        max-width: 600px;
        margin: 0 auto;
    }

    .features-grid {
        display: grid;
        grid-template-columns: repeat(auto-fit,
minmax(300px, 1fr));
        gap: 40px;
    }

    /* Картки з принципом фігури та фону */
    .feature-card {
        background: var(--white);
        padding: 40px 30px;
        border-radius: 16px;
        text-align: center;
```

```

        transition: all 0.3s ease;
        box-shadow: 0 4px 20px rgba(0,0,0,0.08);
        position: relative;
    }

    .feature-card:hover {
        transform: translateY(-8px);
        box-shadow: 0 20px 60px rgba(0,0,0,0.15);
    }

    .feature-icon {
        width: 80px;
        height: 80px;
        background: linear-gradient(135deg, var(--
primary-color), #3b82f6);
        border-radius: 50%;
        display: flex;
        align-items: center;
        justify-content: center;
        margin: 0 auto 24px;
        font-size: 2rem;
        color: var(--white);
    }

    .feature-card h3 {
        font-size: 1.5rem;
        font-weight: 600;
        margin-bottom: 16px;
        color: var(--accent-color);
    }

```



```
.feature-card p {
    color: var(--text-secondary);
    line-height: 1.8;
}

/* Адаптивный дизайн */
@media (max-width: 768px) {
    .hero-content {
        grid-template-columns: 1fr;
        text-align: center;
        gap: 40px;
    }

    .hero-text h1 {
        font-size: 2.5rem;
    }

    .nav ul {
        gap: 20px;
    }

    .btn-secondary {
        margin-left: 0;
        margin-top: 16px;
    }

    .features-grid {
        gap: 30px;
    }
}
```

```

    }
    </style>
</head>
<body>
    <!-- Заголовок з принципом консистентності -->
    <header class="header">
        <div class="container">
            <div class="logo">TechSolutions</div>
            <nav class="nav">
                <ul>
                    <li><a href="#"
class="active">Головна</a></li>
                    <li><a href="#">Послуги</a></li>
                    <li><a href="#">Проекти</a></li>
                    <li><a href="#">Контакти</a></li>
                </ul>
            </nav>
        </div>
    </header>

    <!-- Hero-секція з візуальною ієрархією -->
    <section class="hero">
        <div class="container">
            <div class="hero-content">
                <div class="hero-text">
                    <h1>Інноваційні технологічні
рішення для вашого бізнесу</h1>
                    <p>Ми допомагаємо компаніям
впроваджувати сучасні технології, автоматизувати
процеси та досягати нових висот в цифровому світі.</p>

```

```

        <a href="#" class="btn btn-
primary">Розпочати проект</a>
        <a href="#" class="btn btn-
secondary">Дізнатися більше</a>
    </div>
    <div class="hero-image">
        <!-- Тут може бути зображення або
додатковий контент -->
    </div>
</div>
</div>
</section>

<!-- Секція переваг з принципами гештальту -->
<section class="features">
    <div class="container">
        <div class="section-title">
            <h2>Чому обирають нас</h2>
            <p>Ми поєднуємо технічну експертизу з
індивідуальним підходом до кожного клієнта</p>
        </div>
        <div class="features-grid">
            <div class="feature-card">
                <div class="feature-icon">🚀</div>
                <h3>Швидкість</h3>
                <p>Реалізуємо проекти в стислі
терміни без компромісів щодо якості</p>
            </div>
            <div class="feature-card">
                <div class="feature-icon">🎯</div>

```

```

        <h3>Точність</h3>
        <p>Детально аналізуємо потреби
бізнесу та створюємо оптимальні рішення</p>
    </div>
    <div class="feature-card">
        <div class="feature-icon"><img alt="Shield icon" data-bbox="791 224 816 244"/></div>
        <h3>Надійність</h3>
        <p>Забезпечуємо стабільну роботу
систем та постійну технічну підтримку</p>
    </div>
</div>
</div>
</section>
</body>
</html>

```

### Тестування та оптимізація композиції

Створення ефективної композиції сайту не закінчується на етапі розробки. Важливо регулярно тестувати та оптимізувати дизайн на основі поведінки користувачів та аналітичних даних.

А/В тестування дозволяє порівнювати різні варіанти композиції та визначати, який з них краще виконує поставлені завдання. Можна тестувати розміщення кнопок призову до дії, колірні схеми, розміри елементів та структуру контенту.

Аналіз теплових карт показує, де саме користувачі найбільше фокусують увагу на сторінці. Це допомагає зрозуміти, чи працюють принципи композиції на практиці та чи потрапляють найважливіші елементи в зони максимальної уваги.

Тестування зручності використання виявляє проблеми в навігації та взаємодії користувачів з інтерфейсом. Спостереження за тим, як реальні

користувачі взаємодіють з сайтом, може розкрити неочевидні проблеми в композиції.

### **4.3 Figma - інструмент сучасного веб-дизайну**

У сучасному світі веб-технологій процес створення якісних користувацьких інтерфейсів неможливо уявити без використання спеціалізованих інструментів дизайну. Серед безлічі рішень, що існують на ринку, особливе місце займає Figma - інноваційна платформа для створення дизайн-проектів, яка за короткий час стала стандартом індустрії завдяки своїм унікальним можливостям та підходу до організації робочого процесу.

#### **4.3.1 Навіщо потрібна Фігма**

Figma представляє собою комплексне рішення для дизайнерів, розробників та продуктових команд, яке кардинально змінює підхід до створення цифрових продуктів. На відміну від традиційних графічних редакторів, які працюють як локальні застосунки, Figma функціонує як веб-платформа, що надає безпрецедентні можливості для колективної роботи та управління дизайн-проектами.

Основна потреба у Figma виникає з необхідності створення сучасних, адаптивних користувацьких інтерфейсів, які повинні працювати безперебійно на різноманітних пристроях та платформах. Традиційні графічні редактори, такі як Photoshop, були створені для обробки растрових зображень та друкованих матеріалів, тоді як Figma з самого початку розроблялася з урахуванням специфіки цифрового дизайну та веб-технологій.

Figma розв'язує критичну проблему синхронізації між різними учасниками проектної команди - дизайнерами, розробниками, менеджерами продукту та стейкхолдерами. Замість того, щоб передавати файли електронною поштою або через файлообмінники, всі учасники процесу можуть працювати з актуальною версією проекту в режимі реального часу, що

значно підвищує ефективність робочого процесу та зменшує кількість помилок.

Ще одним важливим аспектом є те, що Figma дозволяє створювати не просто статичні макети, а інтерактивні прототипи, які максимально наближені до фінального продукту. Це дає можливість тестувати користувацький досвід ще на етапі проектування, що економить значні ресурси на подальших стадіях розробки.

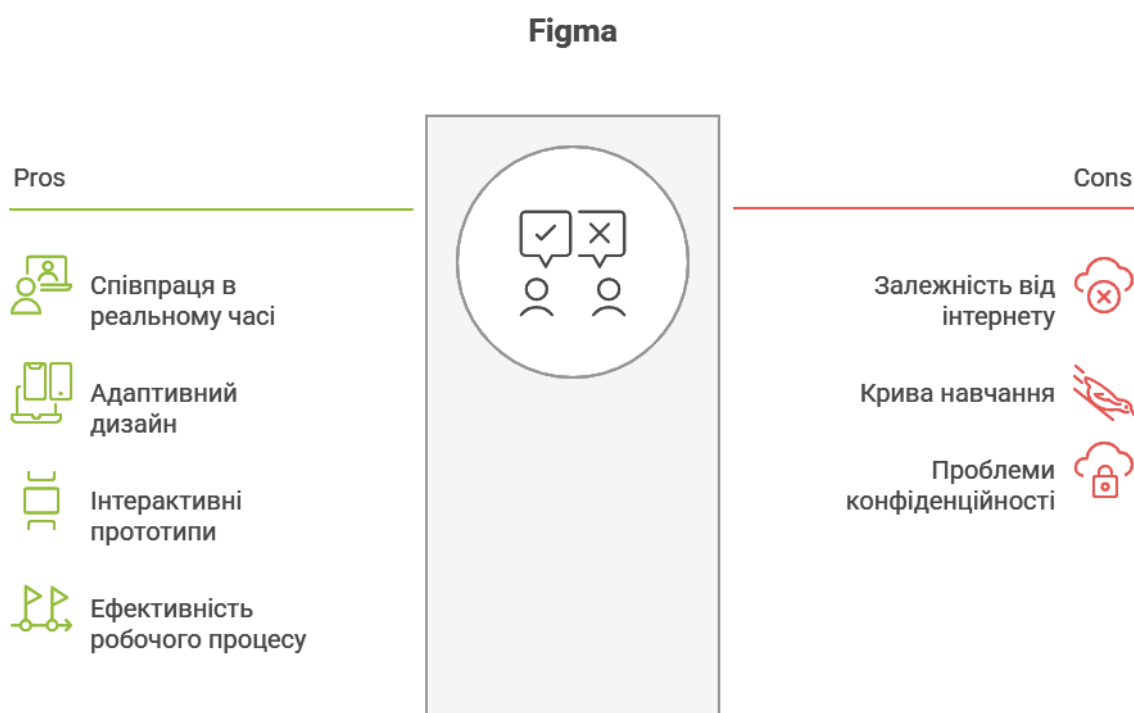


Рисунок 4.35 — Figma: переваги та недоліки

### 4.3.2 Які проекти можна створювати у Фігмі

Спектр проектів, які можна реалізувати за допомогою Figma, надзвичайно широкий та охоплює практично всі сфери цифрового дизайну. Найбільш поширеним застосуванням є створення веб-сайтів - від простих лендингів до складних корпоративних порталів та e-commerce платформ. Figma дозволяє проектувати адаптивні інтерфейси, які коректно відображаються на різних розширеннях екранів, що є критично важливим в епоху мобільного інтернету.

Мобільні додатки становлять ще одну велику категорію проектів, для яких Figma є ідеальним інструментом. Платформа надає готові шаблони та компоненти для iOS та Android, що значно прискорює процес створення мобільних інтерфейсів. Можливість створення інтерактивних прототипів дозволяє детально опрацювати користувацькі сценарії та переходи між екранами ще до початку розробки.

Десктопні додатки також можуть бути спроектовані у Figma, особливо якщо йдеться про крос-платформні рішення, що використовують веб-технології як основу. Електронні документи, презентації, інфографіка, банери для соціальних мереж - всі ці типи проектів можуть бути ефективно реалізовані за допомогою інструментів Figma.

Окремо варто відзначити можливості створення дизайн-систем - комплексних бібліотек компонентів, стилів та керівних принципів, які забезпечують консистентність дизайну в рамках великих проектів або цілих продуктових екосистем. Figma надає потужні інструменти для створення, підтримки та розповсюдження дизайн-систем між різними командами та проектами.

Прототипування складних користувацьких взаємодій, анімацій та мікроінтеракцій також є сильною стороною платформи. Можливість створення високоточних прототипів дозволяє передати розробникам не тільки візуальний дизайн, але й детальну інформацію про поведінку інтерфейсних елементів.

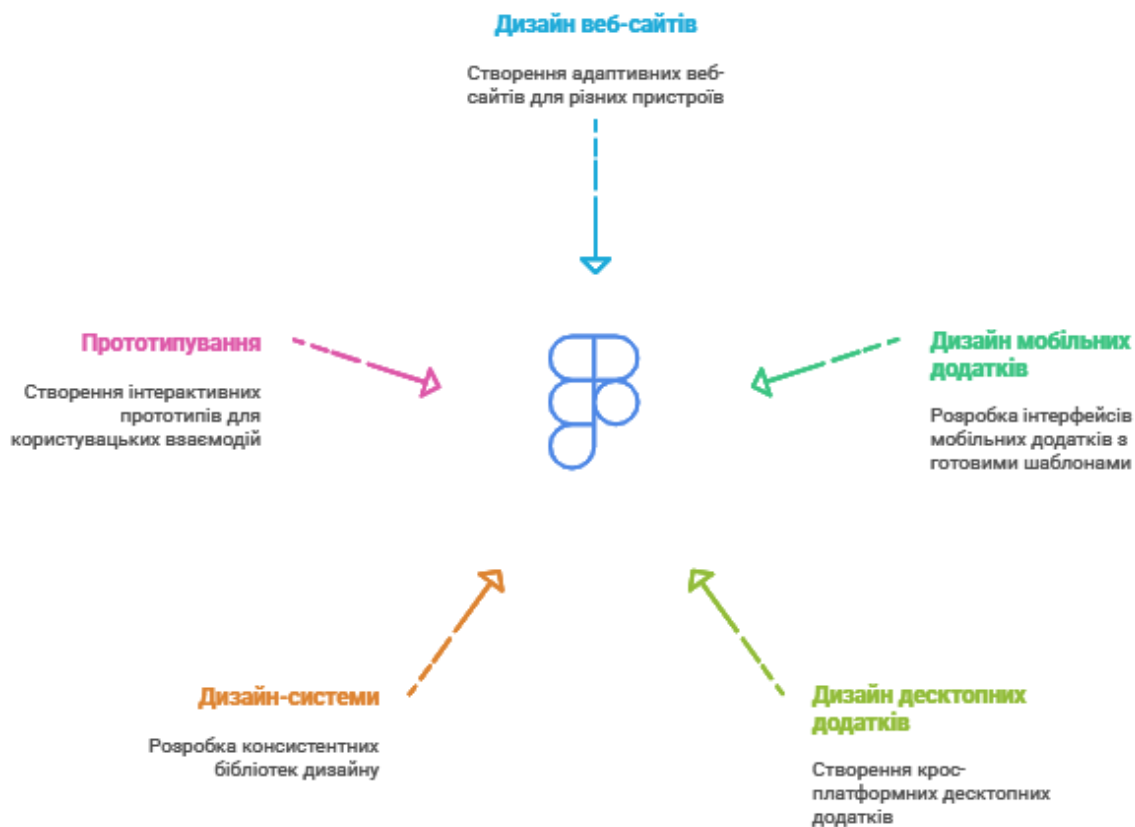


Рисунок 4.36 — Можливості Figma у цифровому дизайні

У Фігмі можна створити кнопки, іконки, форми зворотного зв'язку та налаштувати ефекти: зробити клікабельні кнопки, розкрити списки, створити анімацію для блоків та попапів.



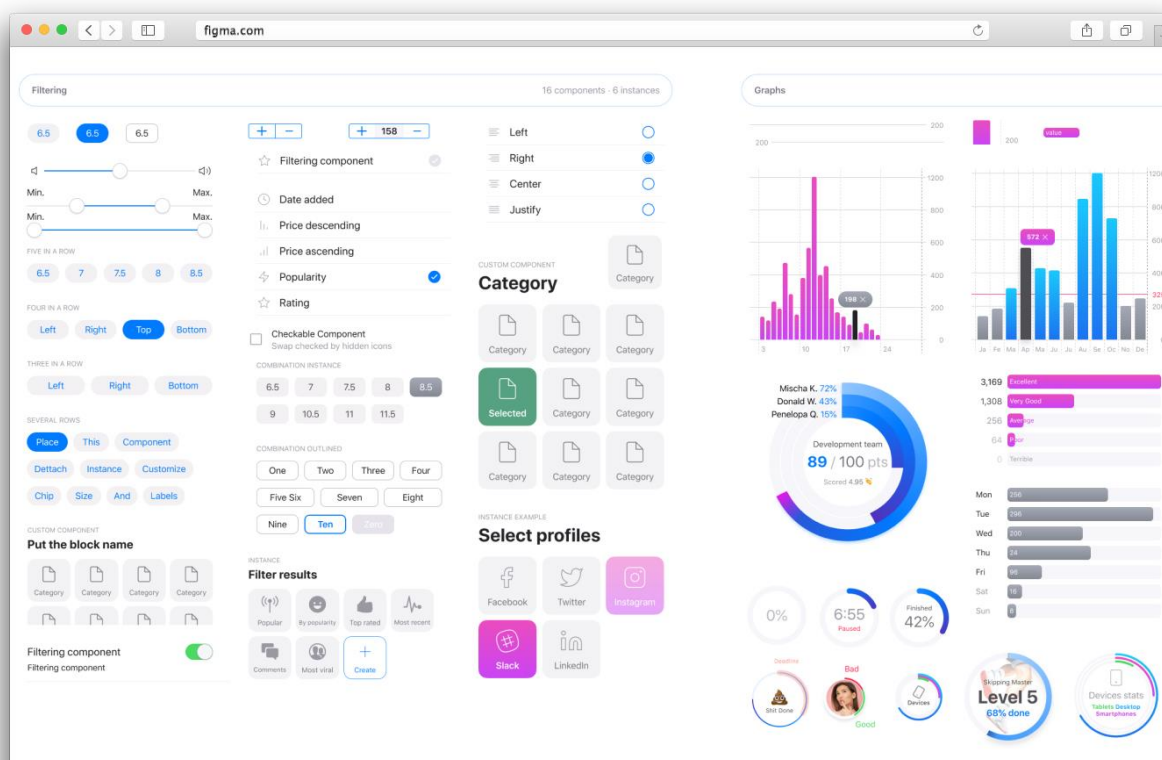


Рисунок 4.37 — Інтерфейс

### 4.3.3 Що ще можна робити у Фігмі

Поза основними функціями дизайн-редактора, Figma пропонує широкий спектр додаткових можливостей, які роблять її універсальним інструментом для різноманітних творчих та бізнес-задач. Однією з найпотужніших особливостей є система плагінів, яка дозволяє розширювати функціональність платформи під конкретні потреби проекту.

Система коментарів та анотацій перетворює Figma на платформу для збору та обробки фідбеку від клієнтів та стейкхолдерів. Замість того, щоб створювати окремі документи з коментарями або проводити довгі наради, всі зауваження можуть бути залишені безпосередньо на макетах, що значно прискорює процес ітерацій та вдосконалення дизайну.

Презентаційний режим дозволяє використовувати Figma для демонстрації проектів клієнтам або команді без необхідності експортувати файли в інші програми. Можливість створення інтерактивних презентацій з

переходами та анімаціями робить такі демонстрації більш переконливими та ефективними.

Інтеграція з іншими інструментами розробки та управління проектами розширює можливості використання Figma в рамках більших робочих процесів. Наприклад, інтеграція з Jira дозволяє автоматично створювати задачі для розробників на основі дизайн-макетів, а інтеграція з Slack забезпечує миттєве сповіщення команди про зміни в проектах.

Можливості створення та ведення документації безпосередньо в Figma роблять платформу корисною не тільки для дизайнерів, але й для технічних письменників, продуктових менеджерів та інших спеціалістів, які працюють з цифровими продуктами.



Рисунок 4.38 — Розширені можливості Figma

Підприємці, SMM-щики та менеджери проектів можуть вести у Фігмі проекти з Діаграмою Ганта, створювати рекламні оголошення з шаблонами для Facebook.

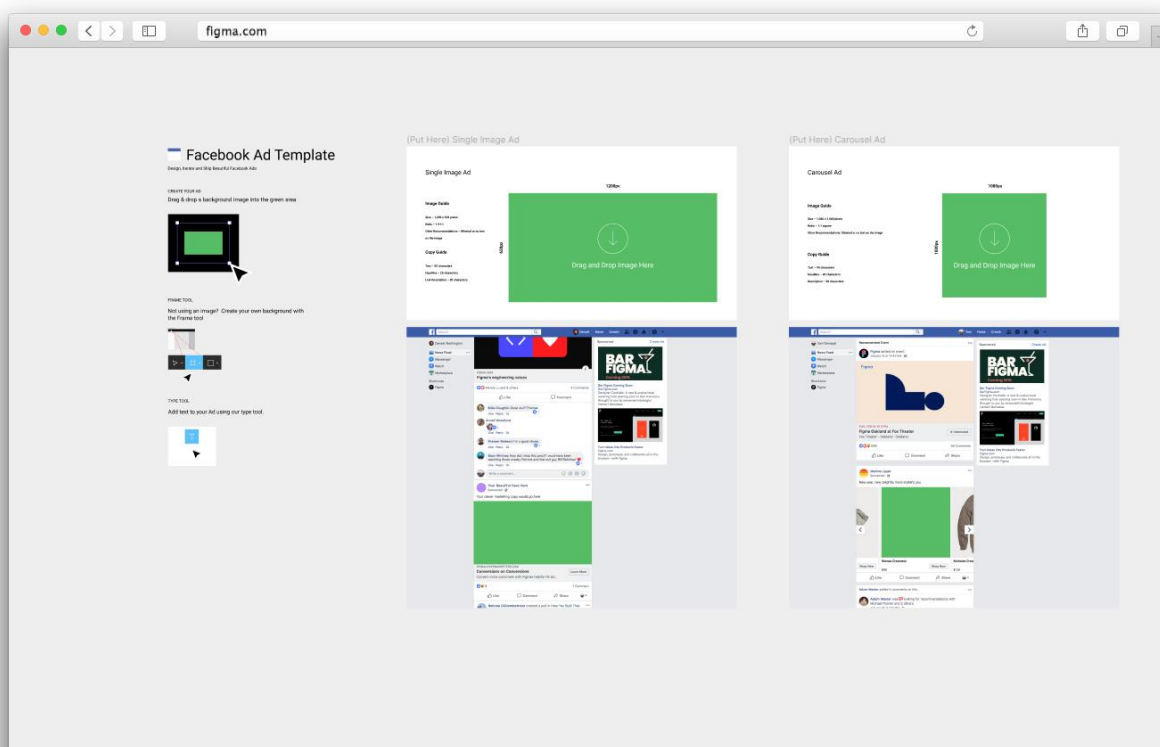


Рисунок 4.39 — Шаблони для Facebook Ads Manager

Для розробників Фігма створила режим Developer Handoff. Можна використовувати напрямні, щоб визначити відстані до об'єктів та їх розмір, скопіювати CSS-стилі елементів, їх код для Android та iOS.

Також є режим Dev Mode, що нагадує інспектор браузера. Якщо ви працюєте дизайнером і часто взаємодієте з розробниками, цей інструмент допоможе полегшити комунікацію із нею.

У Dev Mode можна об'єднати бібліотеку дизайну, базу коду та інструменти керування проектами, не перемикаючись між вкладками.

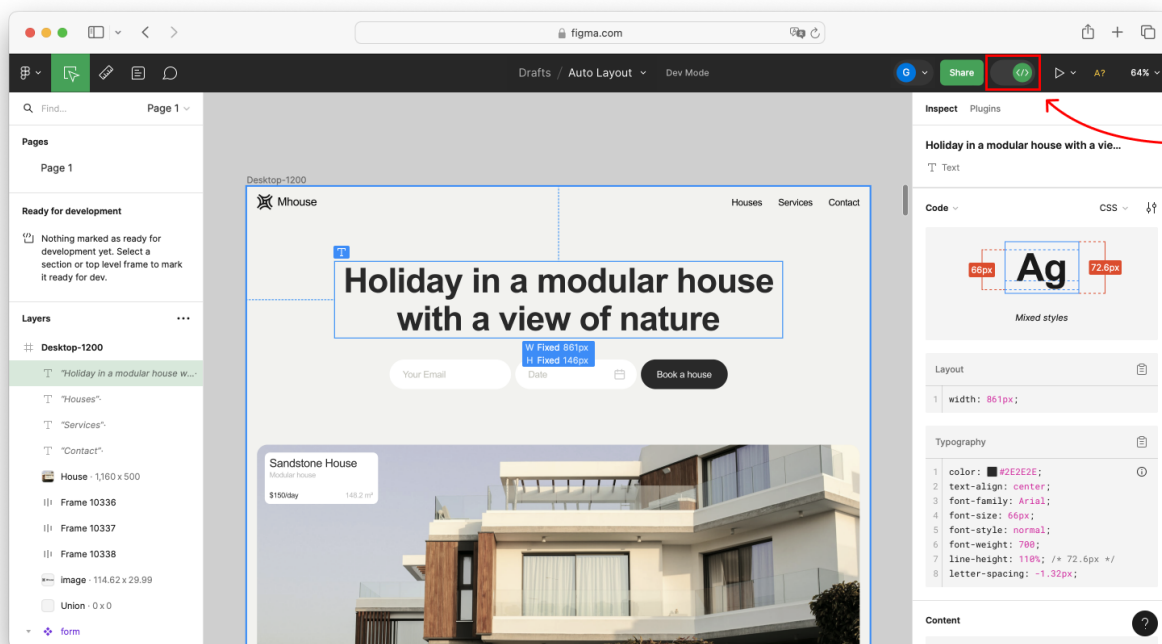


Рисунок 4.40 — Кнопка переходу в режим Dev Mode

#### 4.3.4 Спільна робота у Фігмі

Револьюційною особливістю Figma є її підхід до колаборативної роботи, який кардинально відрізняє цю платформу від традиційних дизайн-інструментів. Можливість одночасної роботи декількох користувачів над одним проектом в режимі реального часу створює новий рівень ефективності для дизайнерських команд.

Система прав доступу дозволяє гнучко налаштовувати рівні доступу для різних учасників проекту. Власники проекту можуть надавати права на редагування, коментування або тільки перегляд, що забезпечує контроль над процесом та запобігає випадковим змінам від неавторизованих користувачів.

Історія версій та можливість відкату до попередніх станів проекту забезпечують безпеку роботи з важливими проектами. Кожна зміна автоматично зберігається, а детальна історія дозволяє відстежити, хто і коли вніс конкретні модифікації.

Система коментарів працює не тільки як інструмент для фідбеку, але й як засіб комунікації між учасниками команди. Коментарі можуть бути

прив'язані до конкретних елементів дизайну, що робить обговорення більш структурованим та зрозумілим.

Можливість створення спільних бібліотек компонентів дозволяє забезпечити консистентність дизайну в рамках великих організацій. Коли дизайнер оновлює компонент в основній бібліотеці, всі інші проекти, що використовують цей компонент, отримують сповіщення про можливість оновлення.

#### 4.3.5 Початок роботи у Фігмі

Процес початку роботи з Figma спроектований таким чином, щоб мінімізувати бар'єр входу для нових користувачів, незалежно від їхнього попереднього досвіду роботи з дизайн-інструментами. Реєстрація в системі є простою та швидкою - достатньо мати електронну пошту або акаунт Google для створення облікового запису.

У сервісі можна працювати з браузера або завантажити програму Фігма на комп'ютер. Вибирайте браузер, якщо часто користуєтесь різними пристроями. Для цього зайдіть на сайт Фігми [figma.com](https://figma.com) та зареєструйтесь.

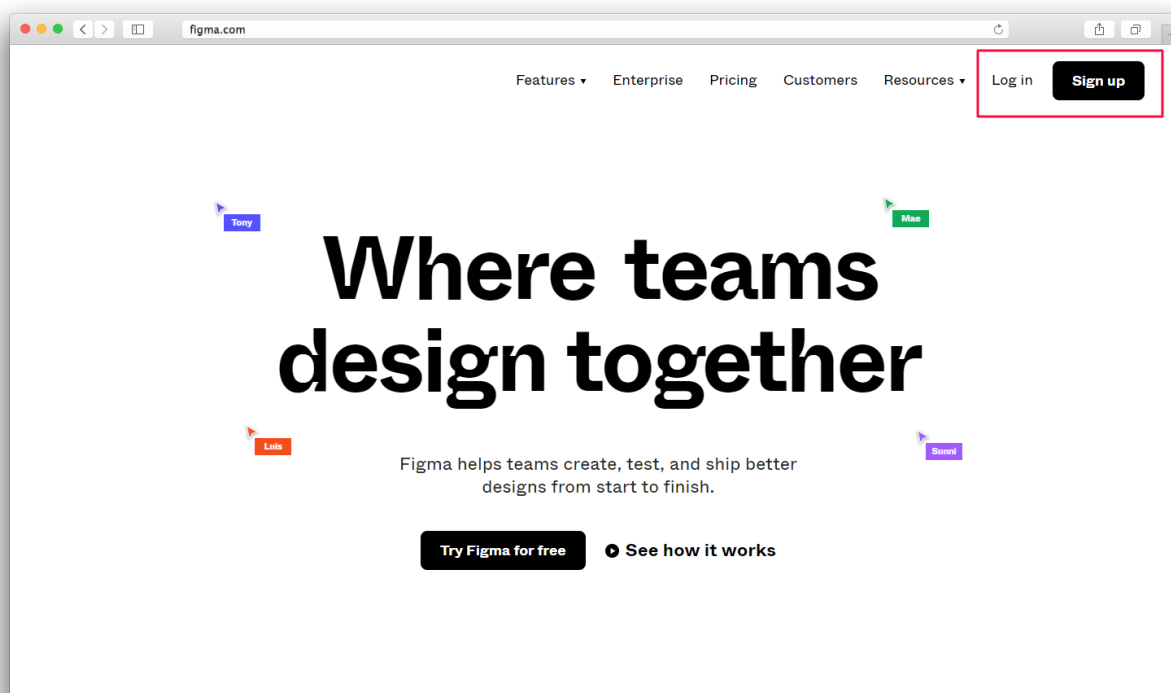


Рисунок 4.41 — Початок роботи

Після входу в систему користувач потрапляє в основний інтерфейс, який складається з декількох ключових областей. Бічна панель містить ієрархію проектів та файлів, центральна область призначена для роботи з дизайном, а верхня панель інструментів надає доступ до основних функцій редагування.

Після реєстрації користувачеві доступні дві робочі області - графічний редактор і менеджер файлів, в якому можна створювати проекти та змінювати налаштування профілю. Розглянемо пункти меню менеджера файлів.

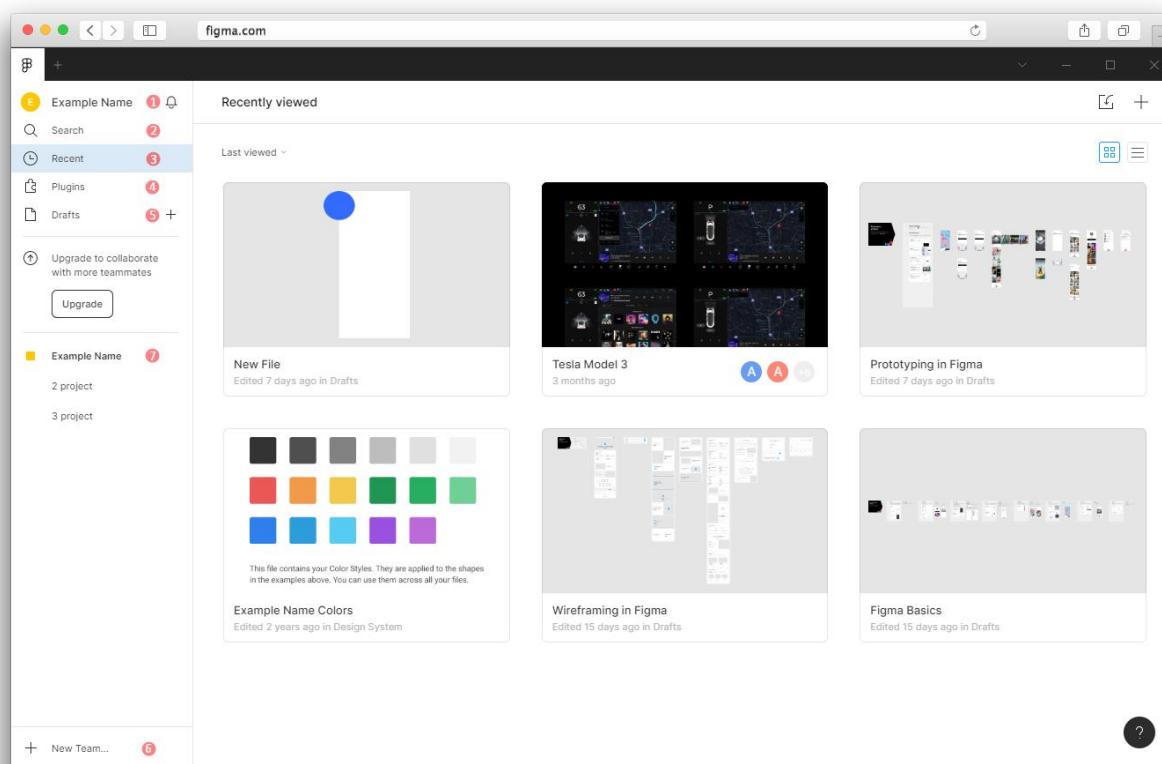


Рисунок 4.42 — Інтерфейс

## 1. Профіль

У налаштуваннях профілю можна завантажити аватарку, змінити ім'я, пароль та email, на який надходять оповіщення. Там же можна оновити тариф та видалити обліковий запис.

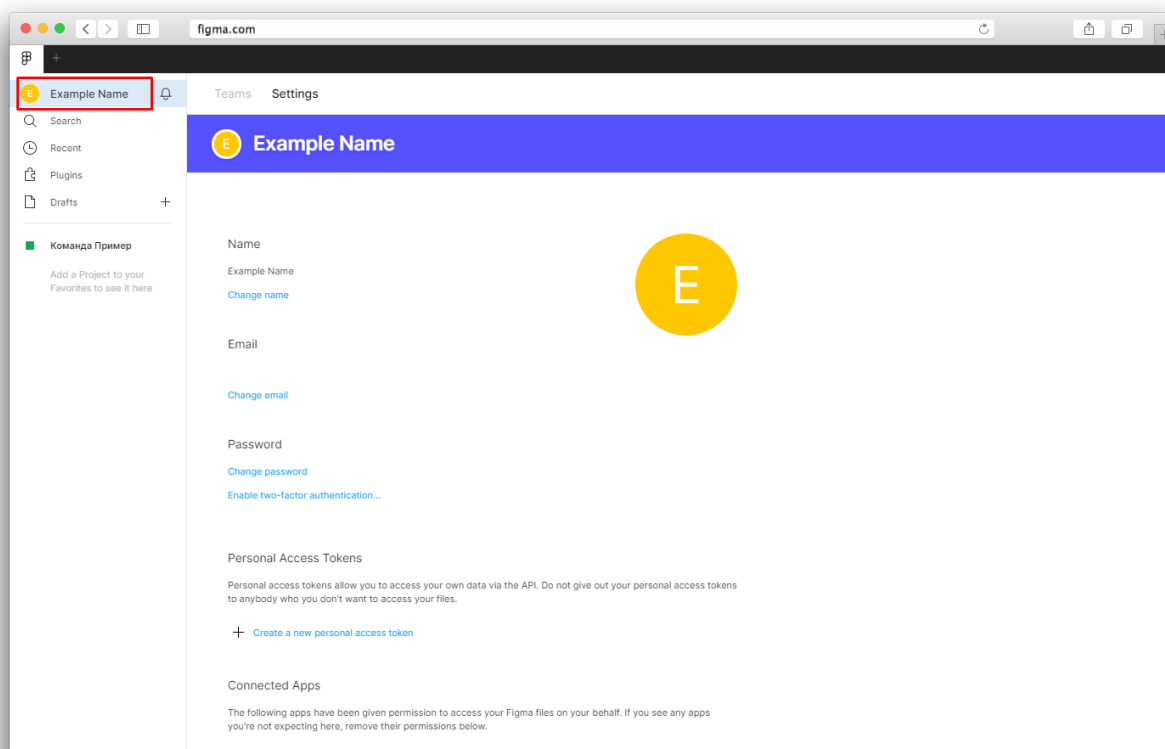


Рисунок 4.43 — Профіль

## 2. Пошук

Шукайте файли та проекти, у яких ви берете участь, через пошук «Search». Перейменуйте файли, з якими хочете працювати в майбутньому, щоб їх було легко знайти.

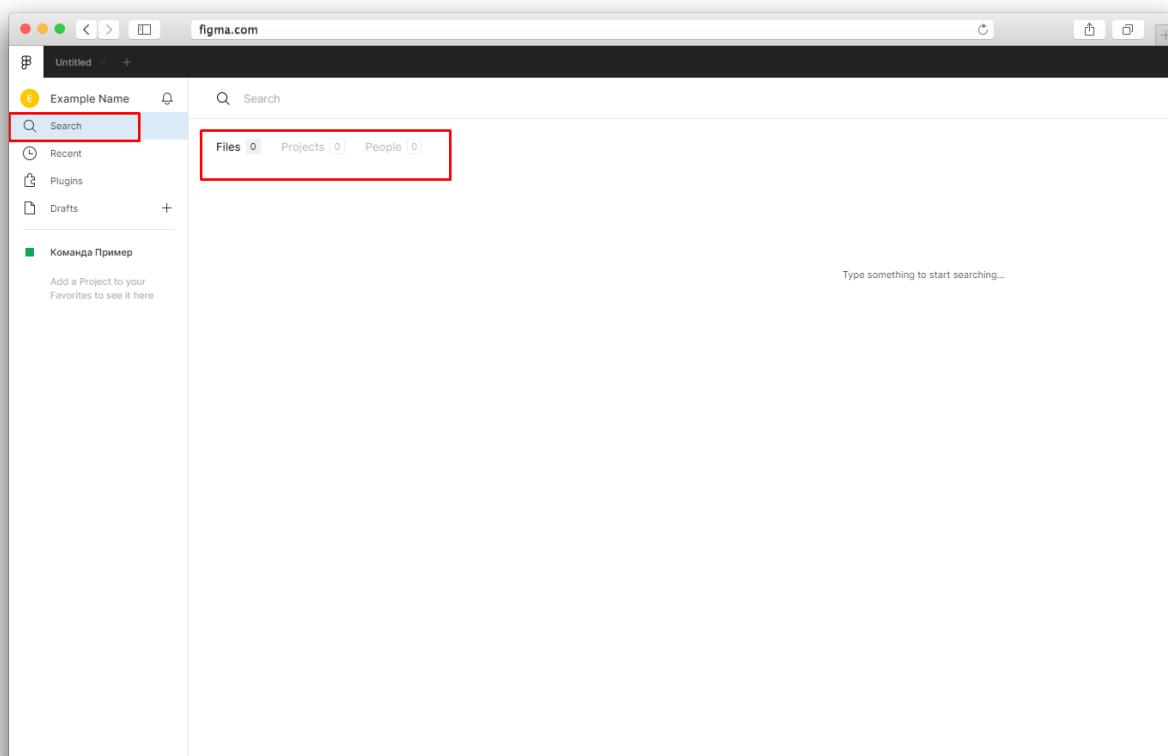


Рисунок 4.44 — Пошук

### 3. Нещодавні файли

У Фігмі файли зберігаються автоматично. Усі файли, які ви відкривали, можна побачити у вкладці Recent .

Фігма зберігає історію змін кожного файлу. Ви легко зможете відновити резервну копію роботи.



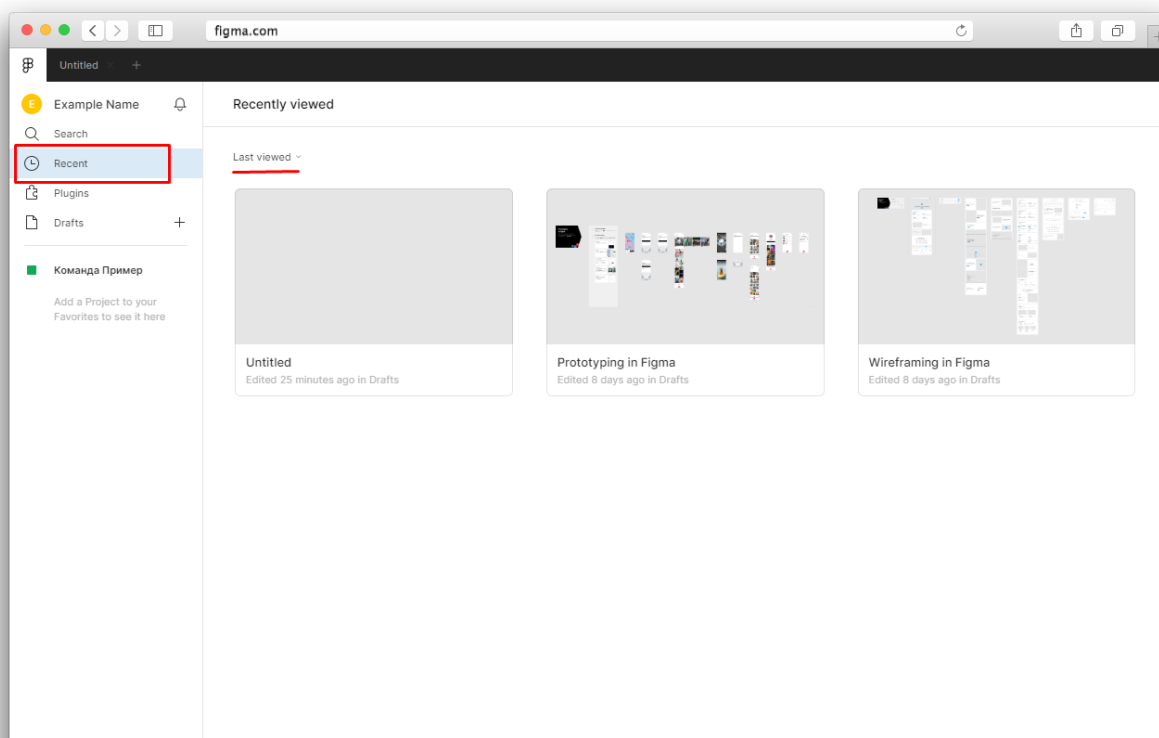


Рисунок 4.45 — Нещодавні файли

#### 4. Плагіни

У пункті Plugins зібрано розширення, які допоможуть прискорити роботу у Фігмі. Наприклад, із плагіном Unsplash можна знаходити стокові ілюстрації, не залишаючи Фігми. А з Iconify під рукою дизайнера будуть 40 тисяч векторних іконок на всі випадки життя.

Плагіни відсортовані за рекомендованими, популярними та встановленими. Щоб переглянути всі плагіни, натисніть Browse all plugin.

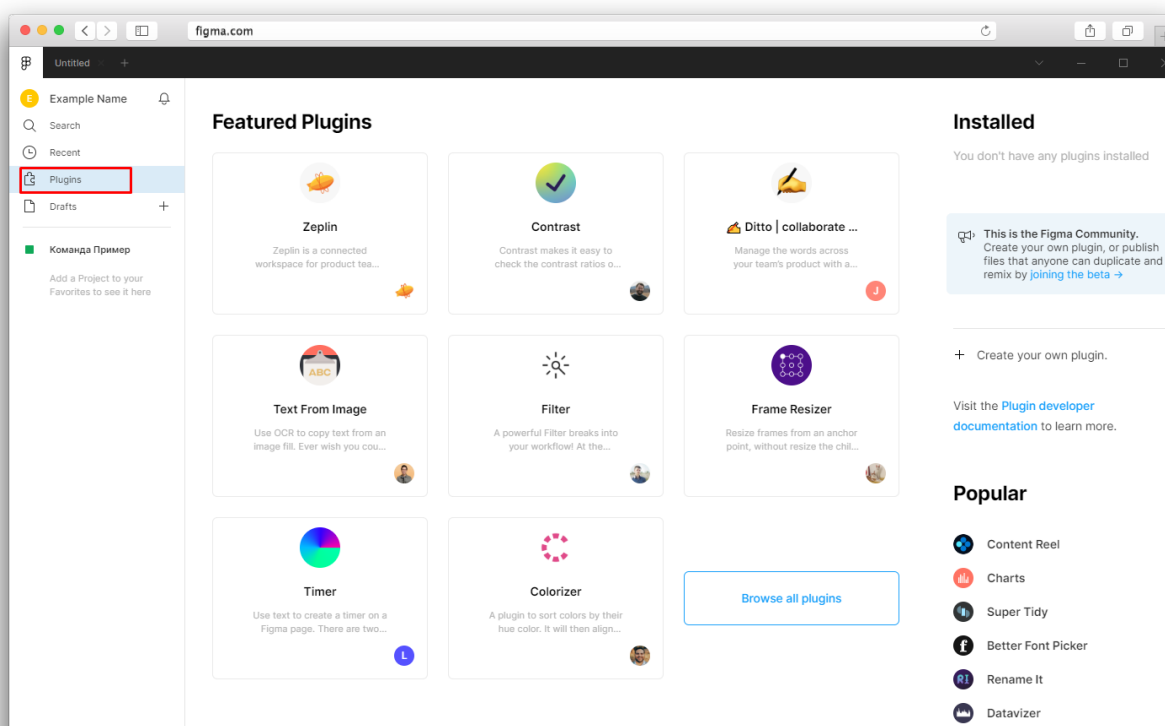


Рисунок 4.46 — Плагіни

## 5. Новий файл

Створити новий файл у Фігмі можна через пункт Drafts або New File у правому кутку екрана. За замовчуванням файл називається **Untitled**.

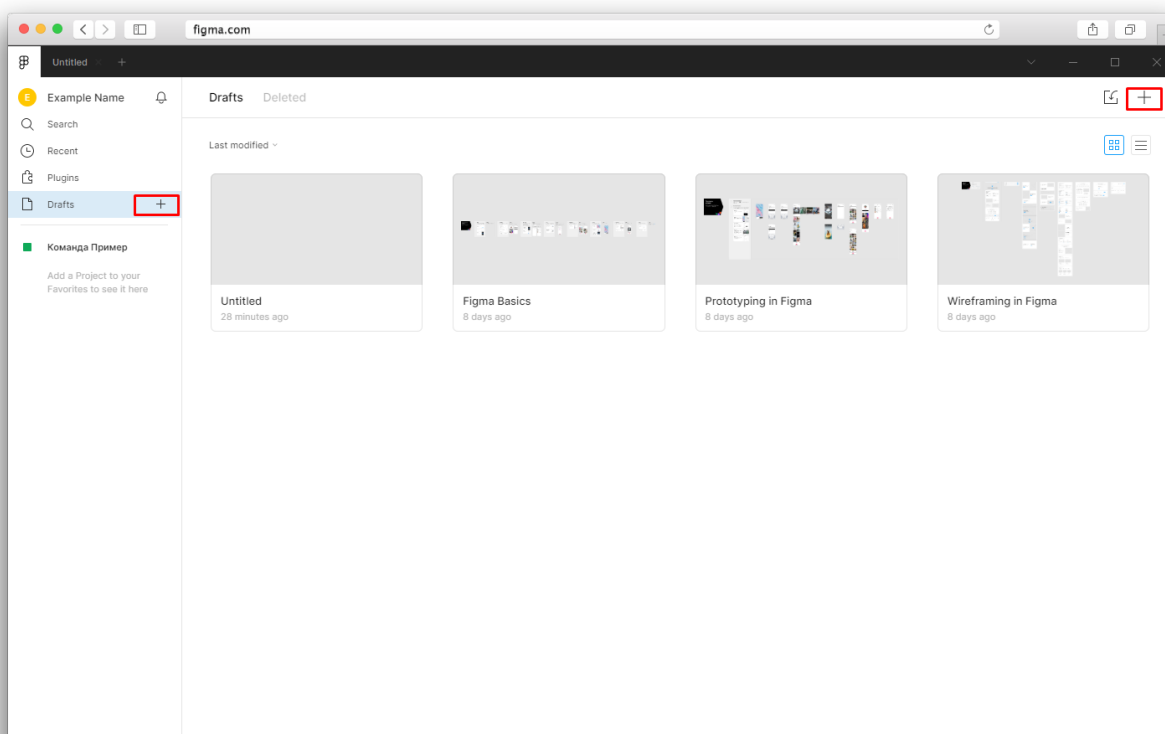


Рисунок 4.47 — Новий файл

## 6. Команда

Щоб розпочати спільний проект, створіть команду: натисніть New Team і придумайте назву.

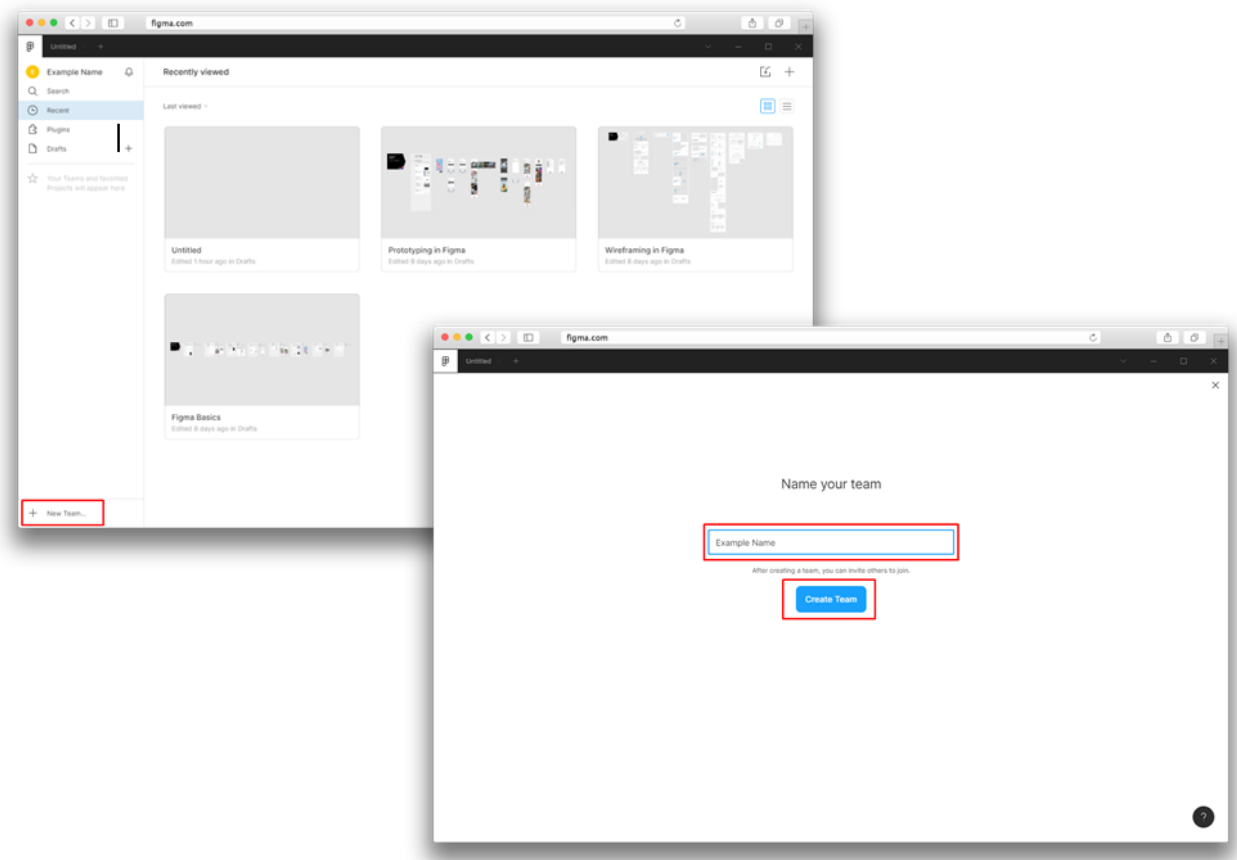


Рисунок 4.48 — Команда

Ви можете запросити учасників по email та позначити їх ролі: лише перегляд або редагування файлу. На безкоштовному тарифі права редактора можуть мати дві особи, включаючи вас.

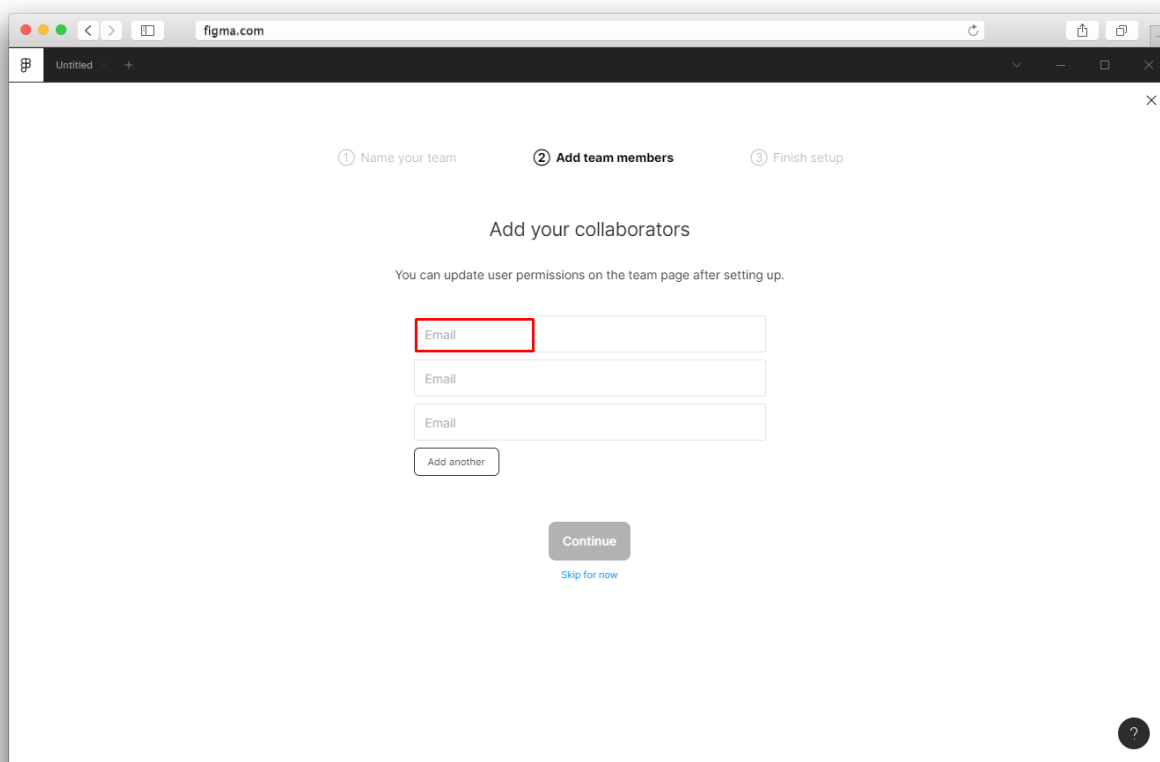


Рисунок 4.49 — Запросити учасників

## 7. Проекти

За умовчанням при додаванні команди, Фігма створює новий проект . Проект – це папка. У ній команда зберігає всі файли, які належать до одного проекту. Це зручно, якщо в кожному проекті у вас є десятки файлів і ви не хочете в них плутатися. На безкоштовному тарифі ви зможете створити максимум три проекти.

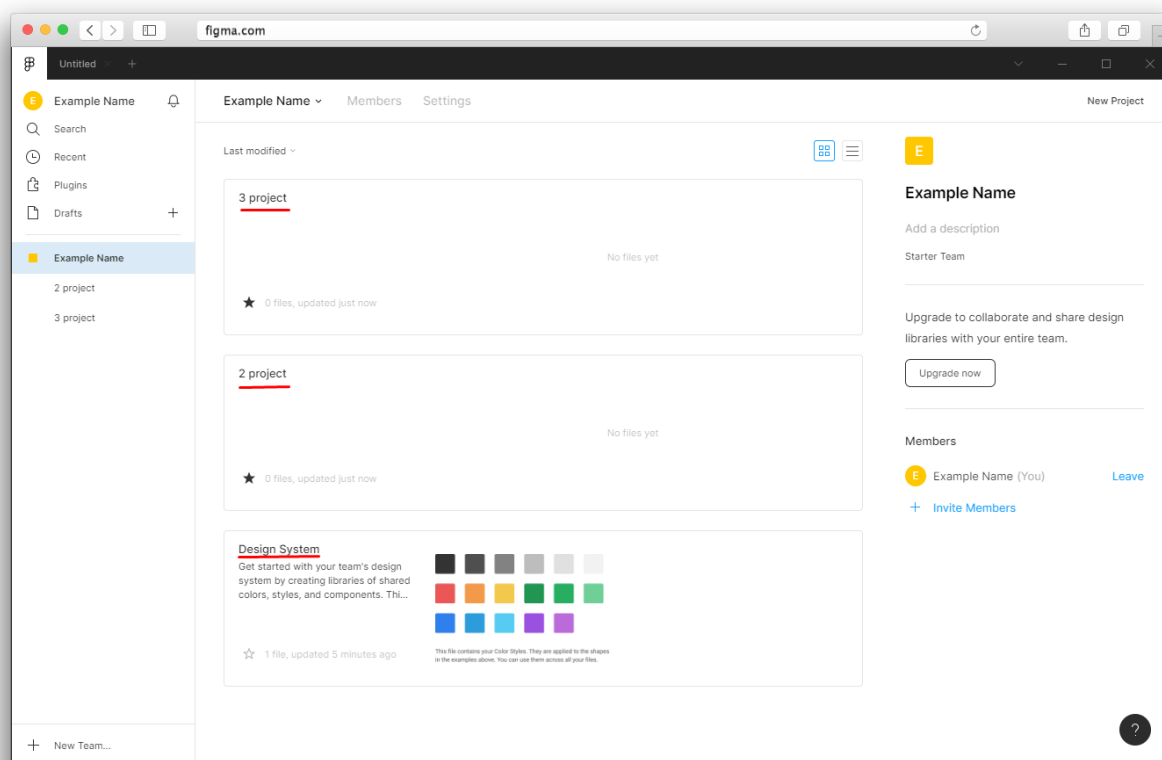


Рисунок 4.50 — Проекти

Щоб створити додатковий проект усередині команди, натисніть New Projects. Всередині проекту створюйте нові файли або перетягуйте сюди SVG, PNG, JPEG-файли.

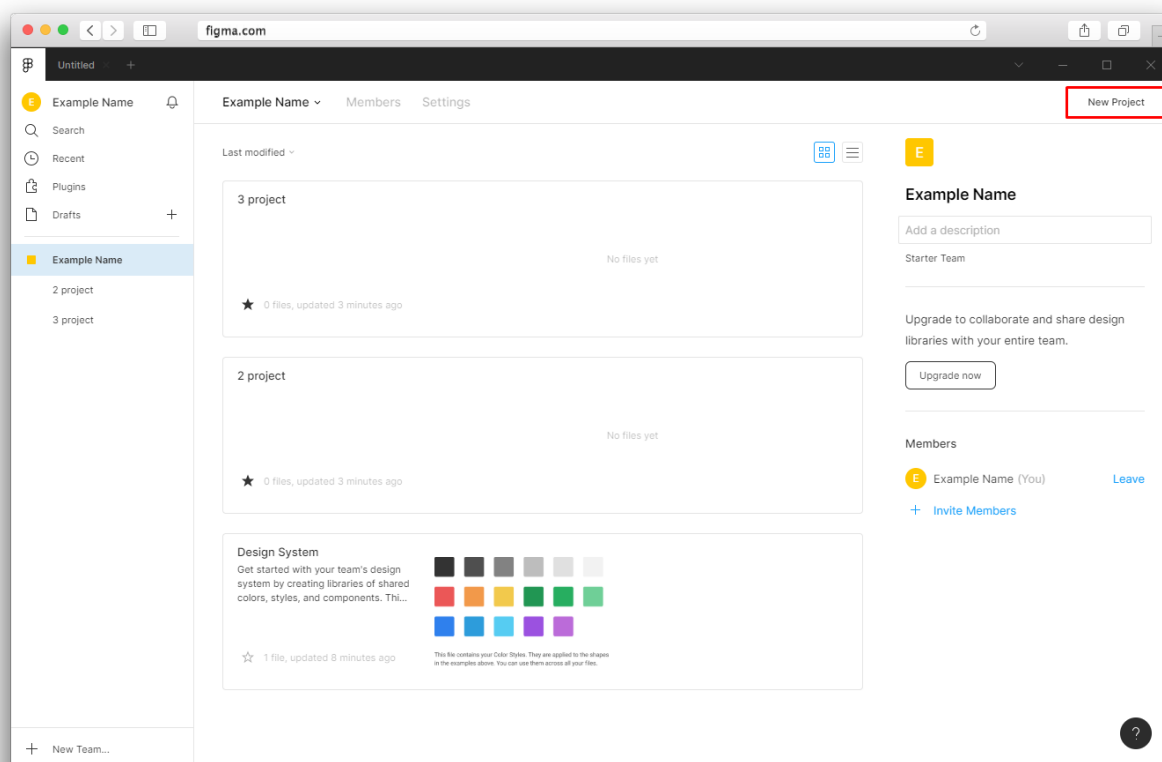
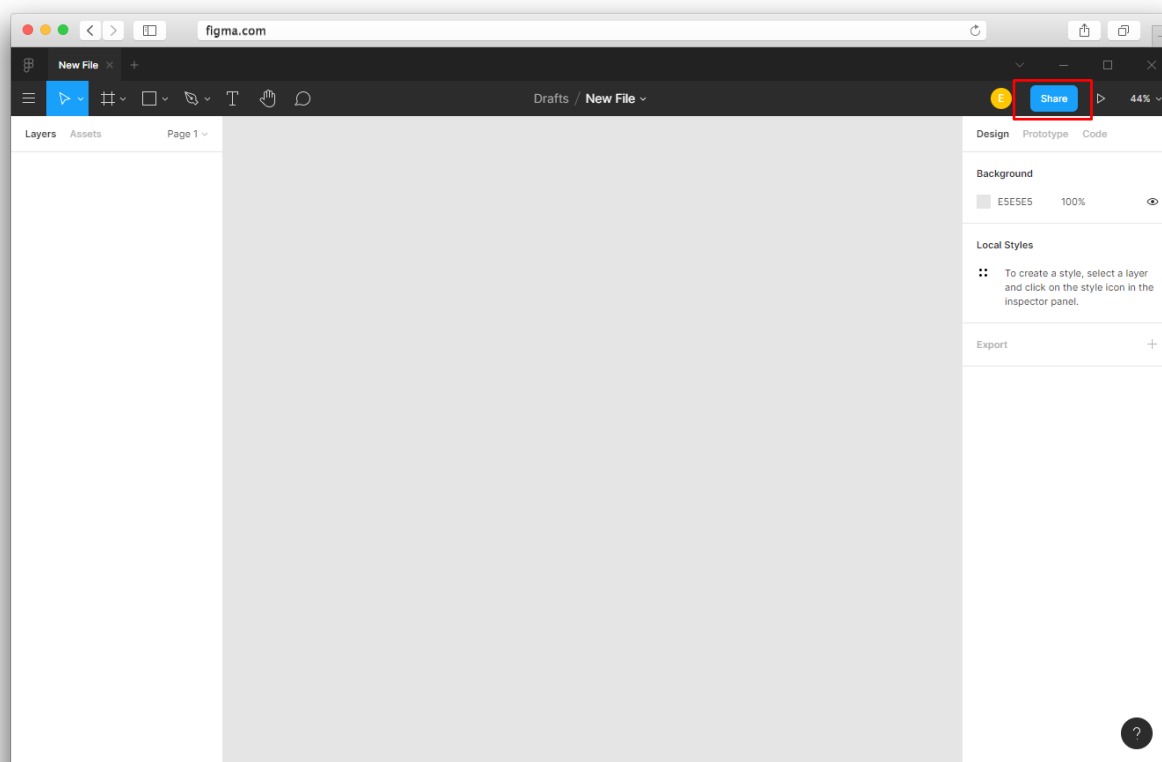


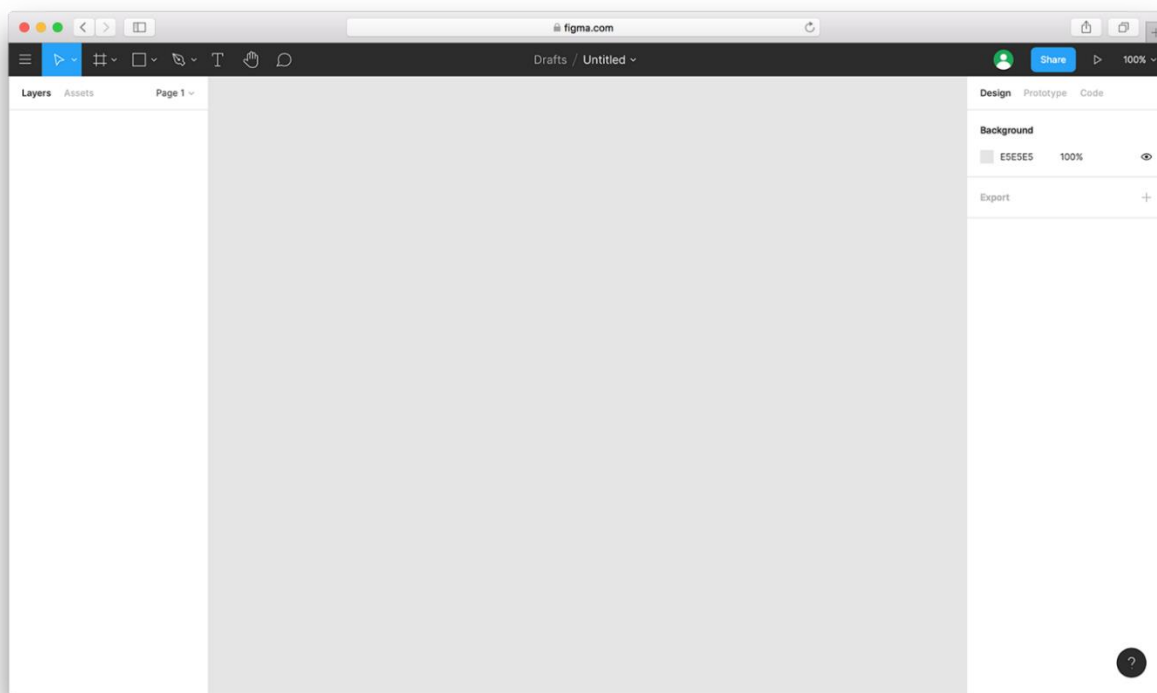
Рисунок 4.51 — New Projects

Якщо ви є фрілансером, і вам не потрібно створювати команди і вести проекти, запрошуйте учасників до роботи з файлом через кнопку Share .



## Рисунок 4.52 — Share

У новому файлі користувачеві доступні три області: робоча область, панель інструментів та панель шарів. Розглянемо важливі функції пунктів меню.



## Рисунок 4.53 — Області

Пошук шукає пункти меню, а не елементи, які ви створювали. Стане в нагоді, якщо вам потрібна конкретна команда, але ви забули, де саме вона знаходиться.

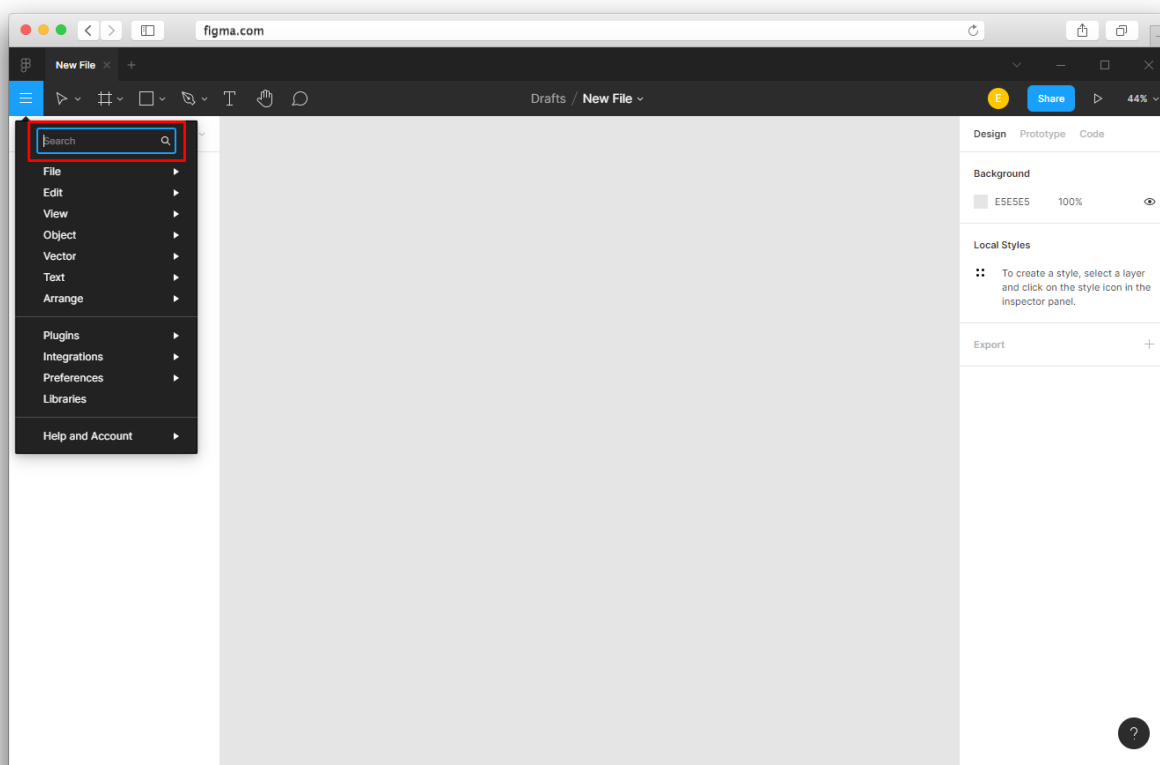


Рисунок 4.54 — Пошук

Панель File. Інструменти панелі допомагають застосувати дію до файлу, у якому ви працюєте. Наприклад, за допомогою New file from Sketch можна імпортувати файли, створені у графічному редакторі Скетч.



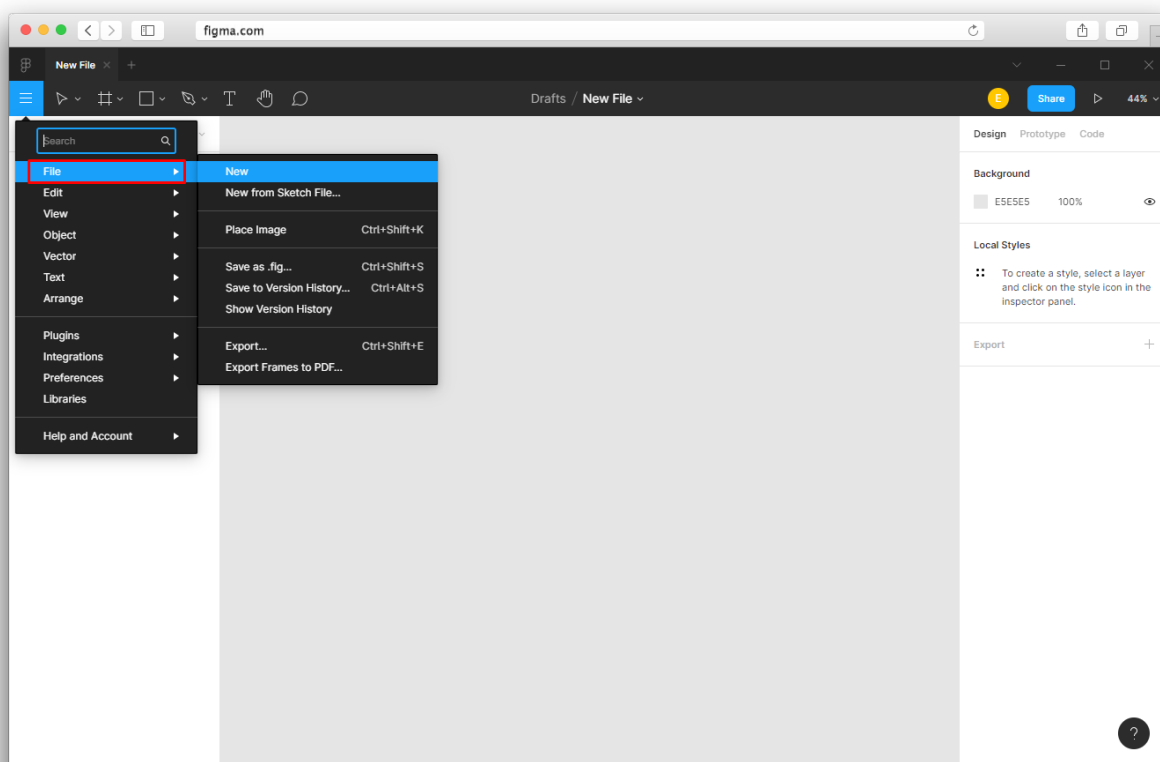


Рисунок 4.55 — Панель File

Фігма - це інструмент для спільної роботи, вона автоматично зберігає зміни файлу.

Ви можете зберегти зміни негайно, натиснувши **Save to Version History** або переглянути історію змін колег, натиснувши **Show Version History**.

Щоб зберегти файл Фігми у PDF, використовуйте **Export Frames to PDF**. Стане в нагоді для презентацій.

Панель **Edit** дозволяє працювати з елементами дизайну у файлі. Щоб застосувати дію до елемента, потрібно виділити. Наприклад, об'єкт можна скопіювати як CSS, SVG або PNG-файл за допомогою **Copy As**.

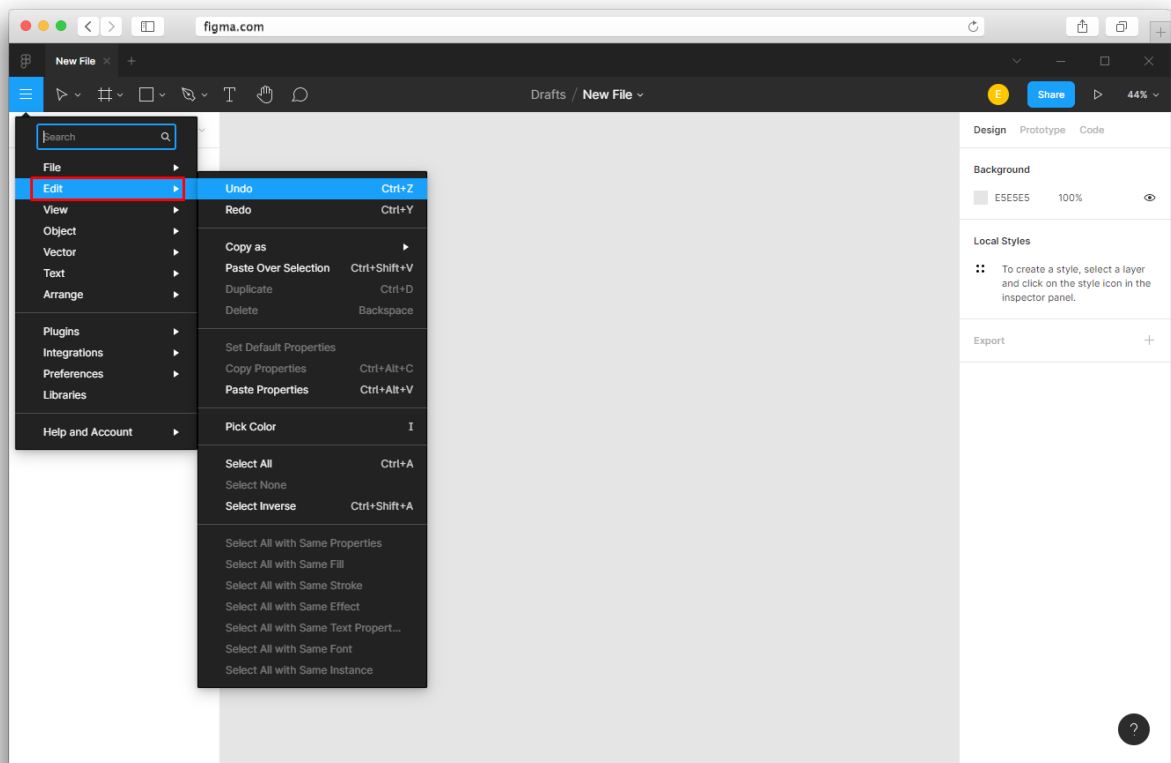


Рисунок 4.56 — Панель Edit

Функція Paste Over Selection дозволяє розмістити скопійований елемент у верхньому лівому куті іншого об'єкта.

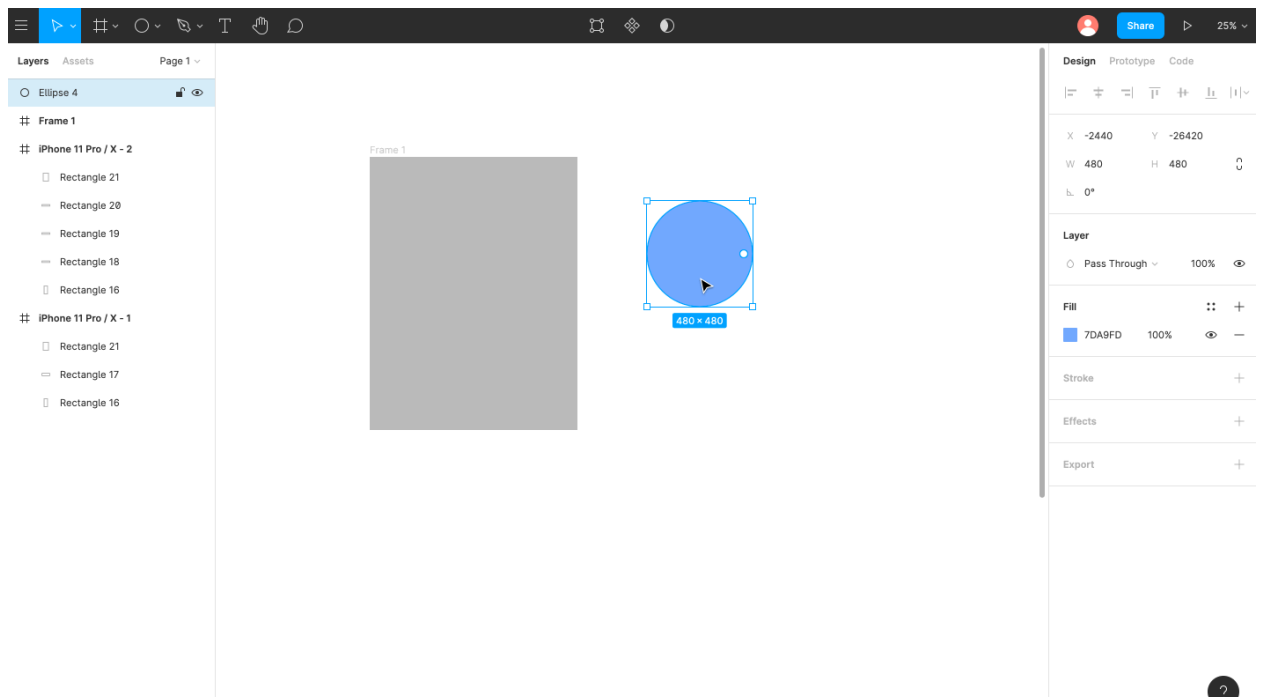


Рисунок 4.57 — Paste Over Selection

Всі нові об'єкти, створені у файлі, за промовчанням сірого кольору. Бувають ситуації, коли потрібно створити безліч об'єктів з іншим кольором. Використовуйте Set Default Properties : скопіюйте колір потрібного об'єкта один раз і всі наступні об'єкти будуть створені із заданими властивостями.

Фарбуйте об'єкт у потрібний колір піпеткою Pick Color .

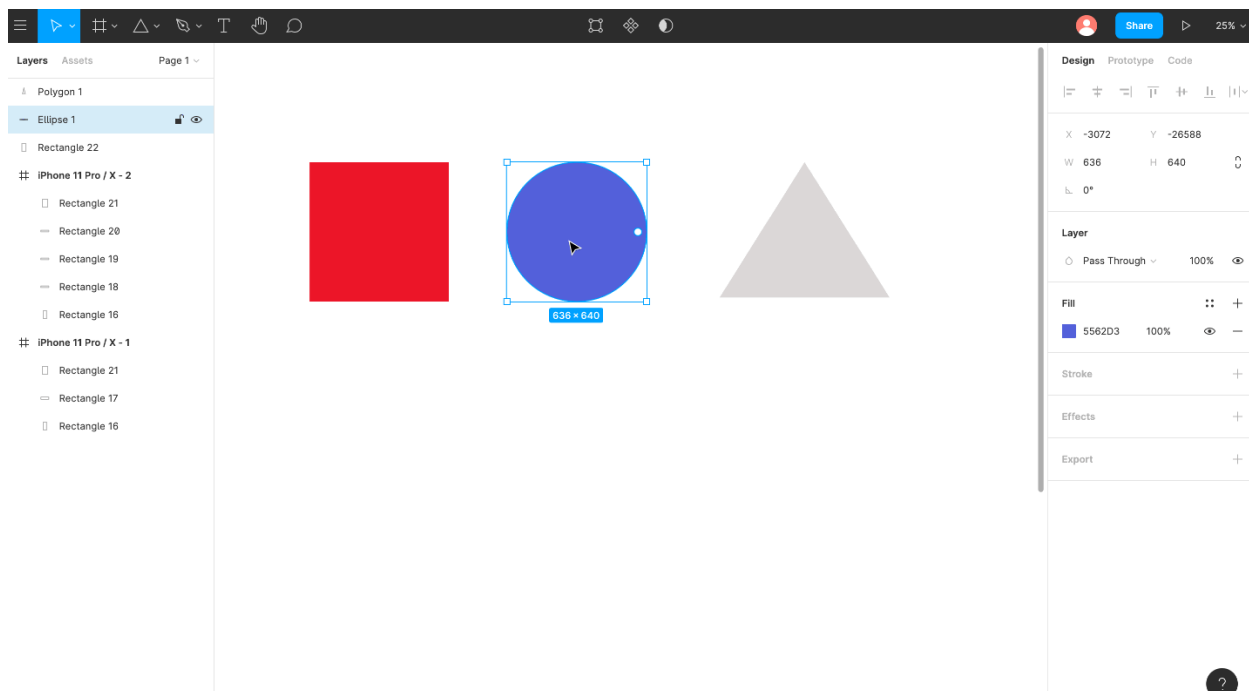


Рисунок 4.58 — Колір

Група команд Select All with підсвічує всі схожі об'єкти у файлі: з однаковими властивостями, заливкою, шрифтами, ефектами тощо.

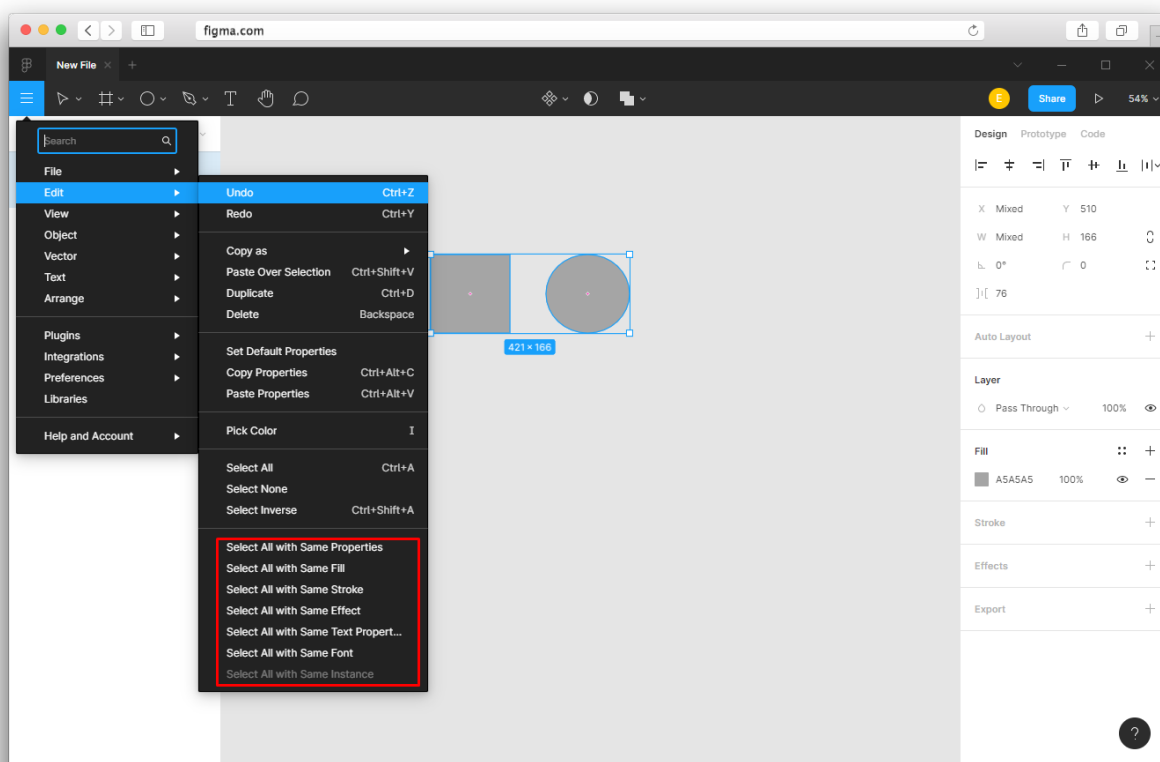


Рисунок 4.59 — Select All with

Панель View знаходиться ліворуч у меню редактора та дублюється праворуч. Вона відповідає за масштабування макета та навігацію по ньому.

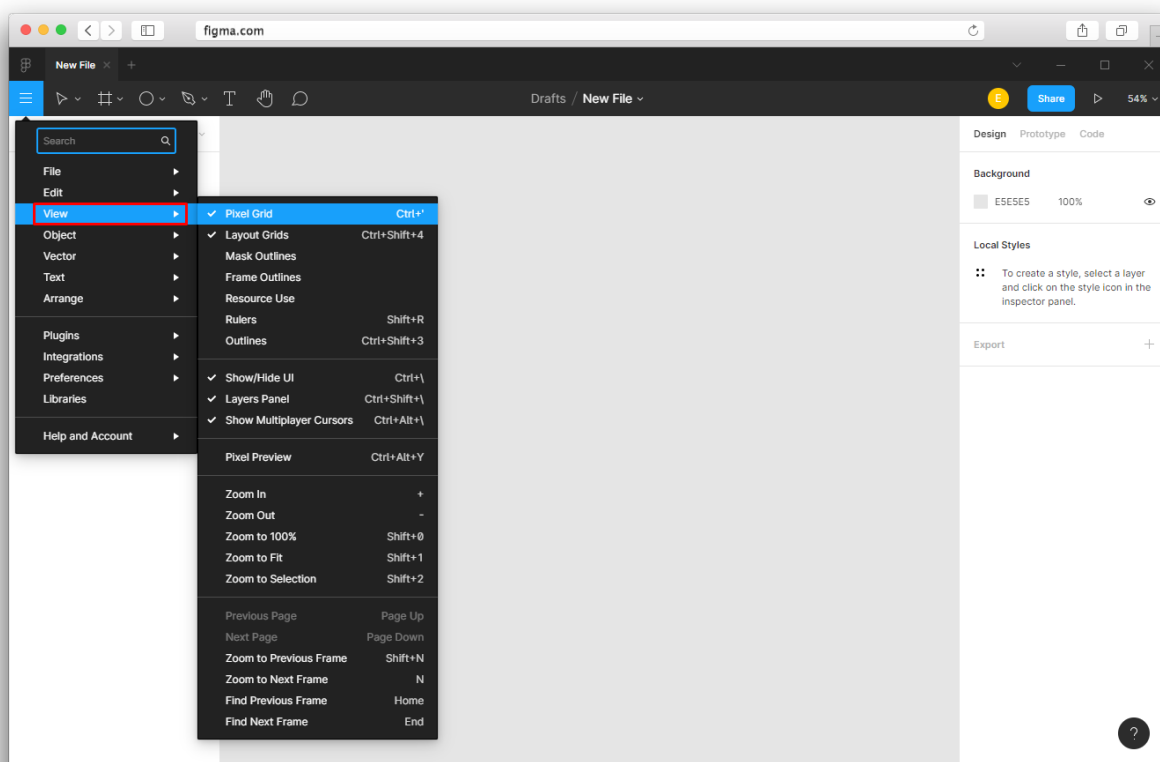


Рисунок 4.60 — View

З корисного – функція Pixel Preview. Фігма — векторний редактор, тож дизайнер бачить рівні векторні лінії. Підключаючи перегляд об'єкта в пікселях, можна побачити, як елемент виглядатиме на екрані.

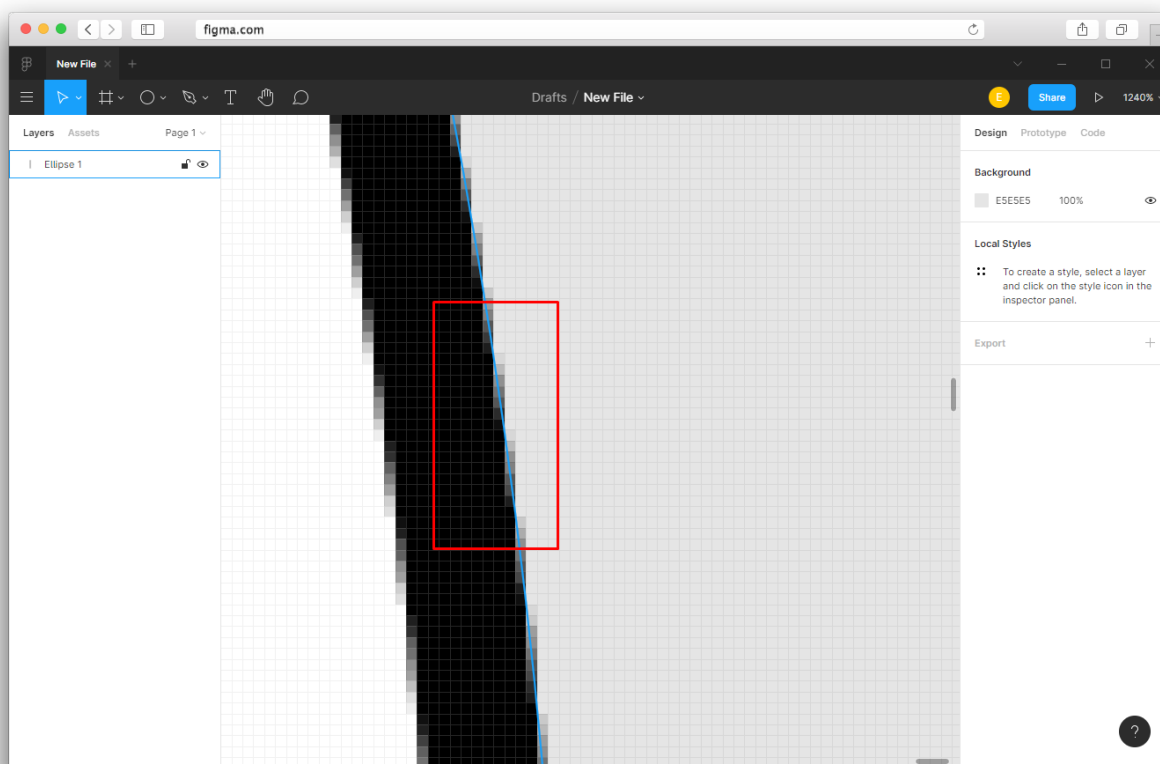


Рисунок 4.61 — Pixel Preview

Масштабувати макет можна функціями Zoom In/Zoom Out .

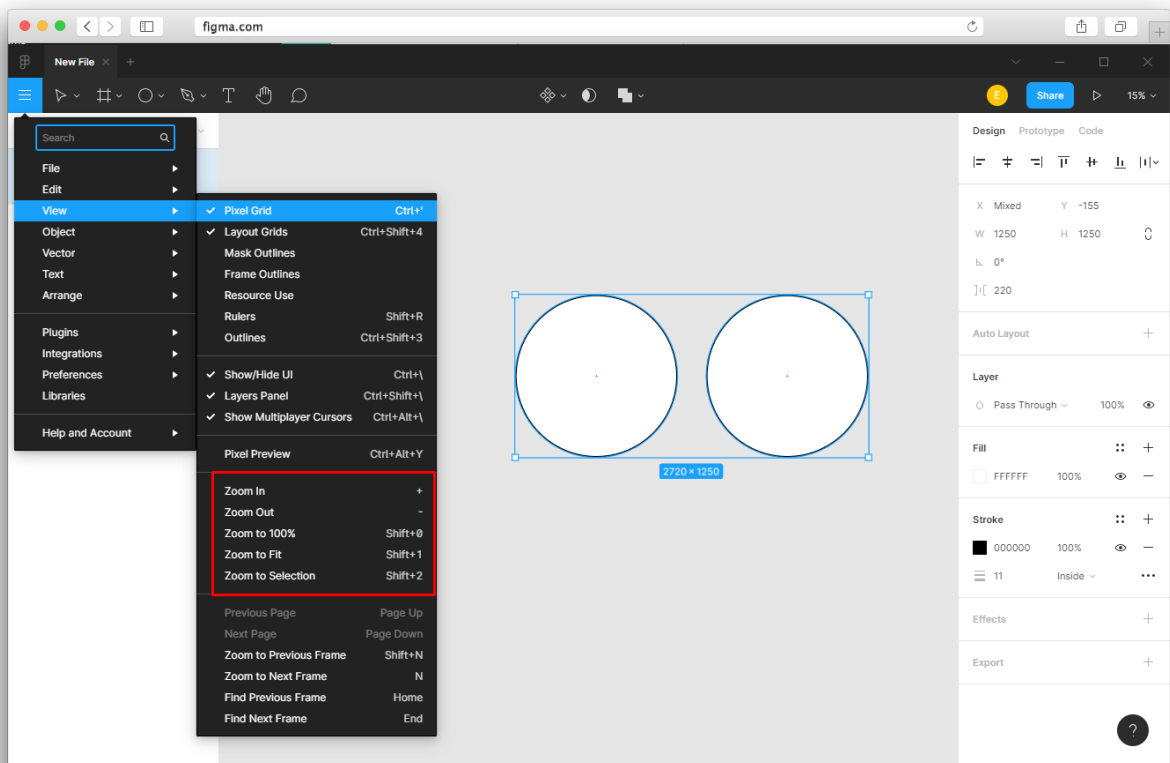


Рисунок 4.62 — Zoom In/Zoom Out

Щоб показати лінійки та витягнути напрямні у Фігмі, скористайтесь Rules .

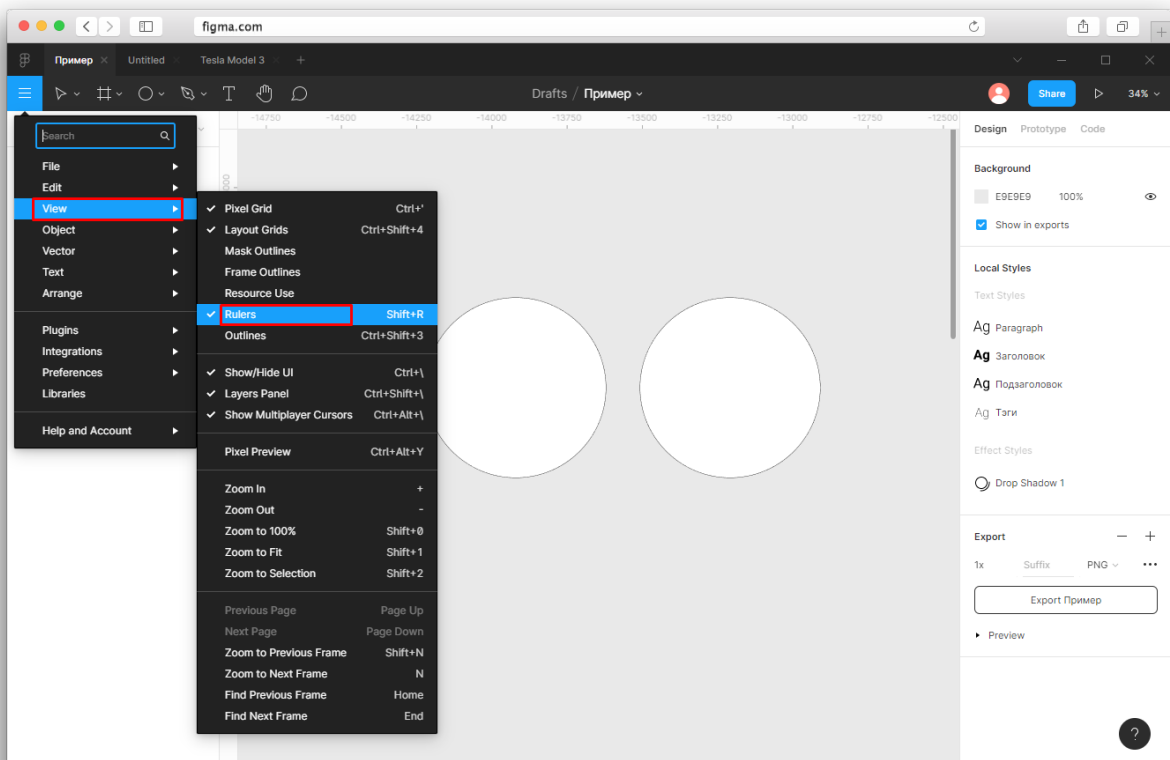


Рисунок 4.63 — Rules

Щоб розширити робочу область та отримати більше простору для натхнення, можна приховати панель шарів та інтерфейс командами Layers Panel, Show/Hide UI .

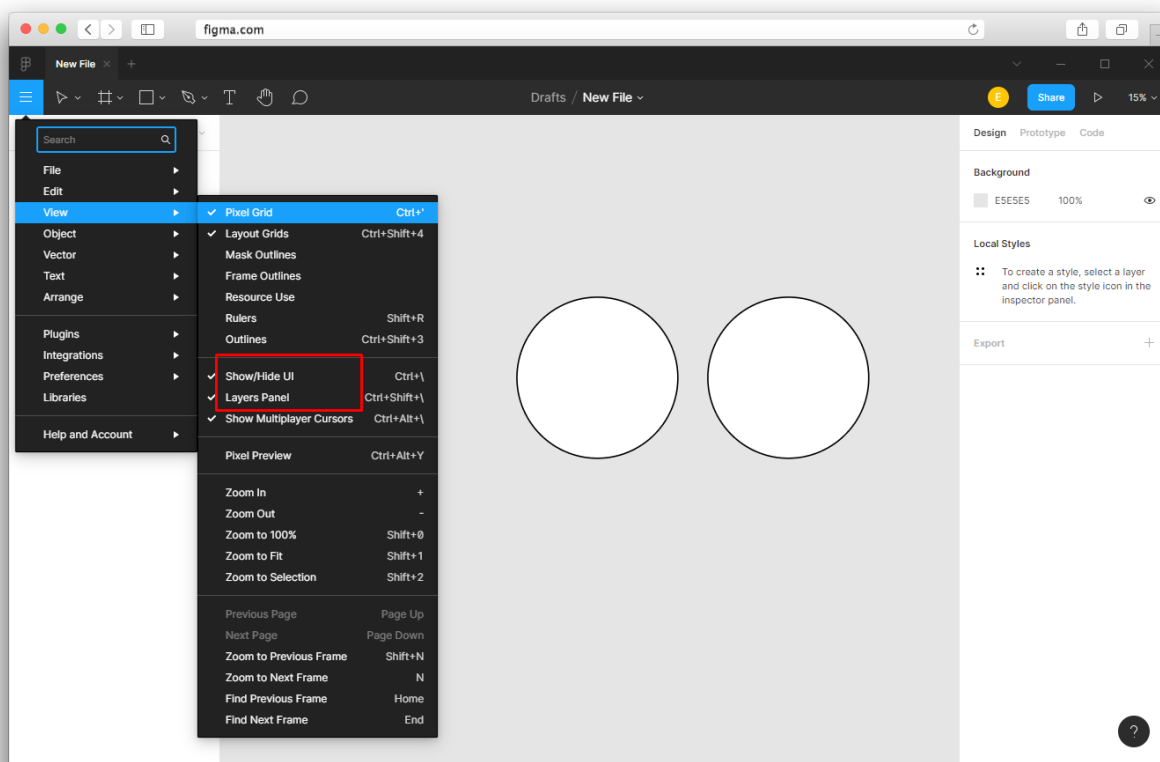


Рисунок 4.64 — Layers Panel, Show/Hide UI

Панель Preferences визначає навігацію у файлі. Перші три пункти – Snap to Geometry, Snap to Pixel Grid, Snap to Objects – налаштування прив'язки. Допоможуть злегка примагнітити елементи та уникнути «кривої» сітки, якщо ви працюєте швидко.

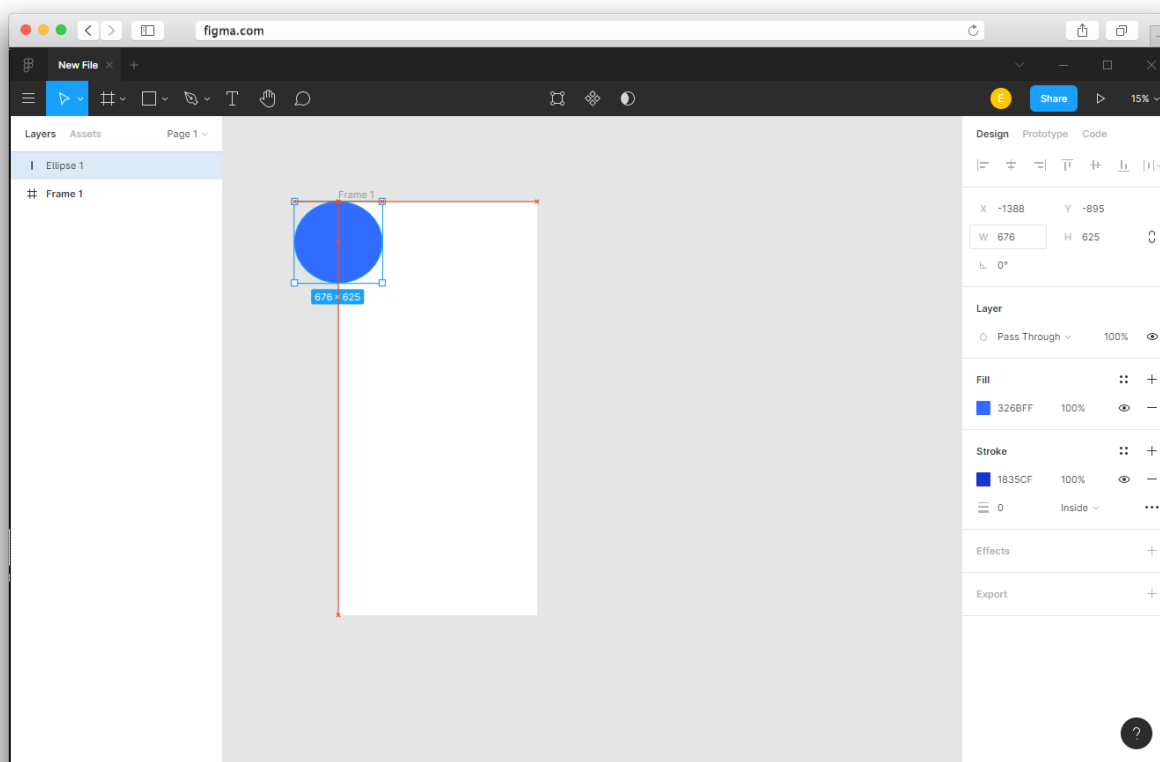


Рисунок 4.65 — Preferences

Highlight Layers on Hover краще тримати увімкненим завжди - функція підсвічує дрібні елементи дизайну при наведенні - кнопки, літери, іконки, значки.

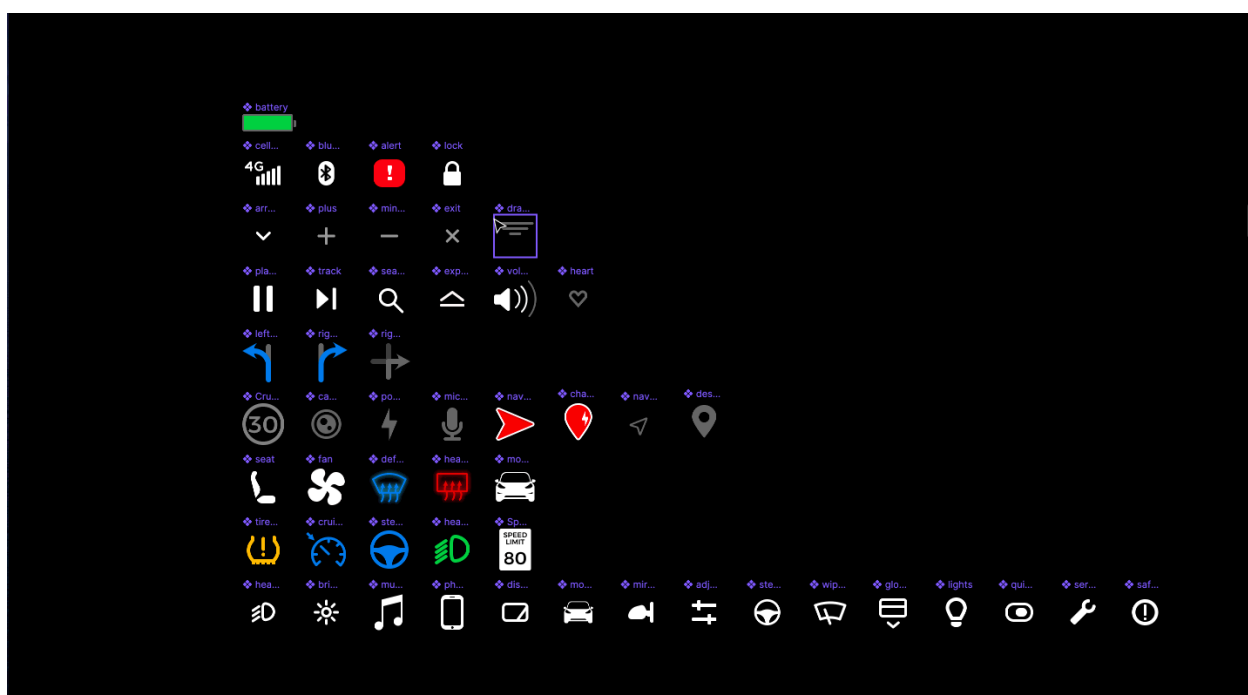


Рисунок 4.66 — Highlight Layers on Hover



Увімкнена функція Keyboard Zooms into Selection наблизитиме/вилучатиме макет щодо елемента, вибраного на екрані.

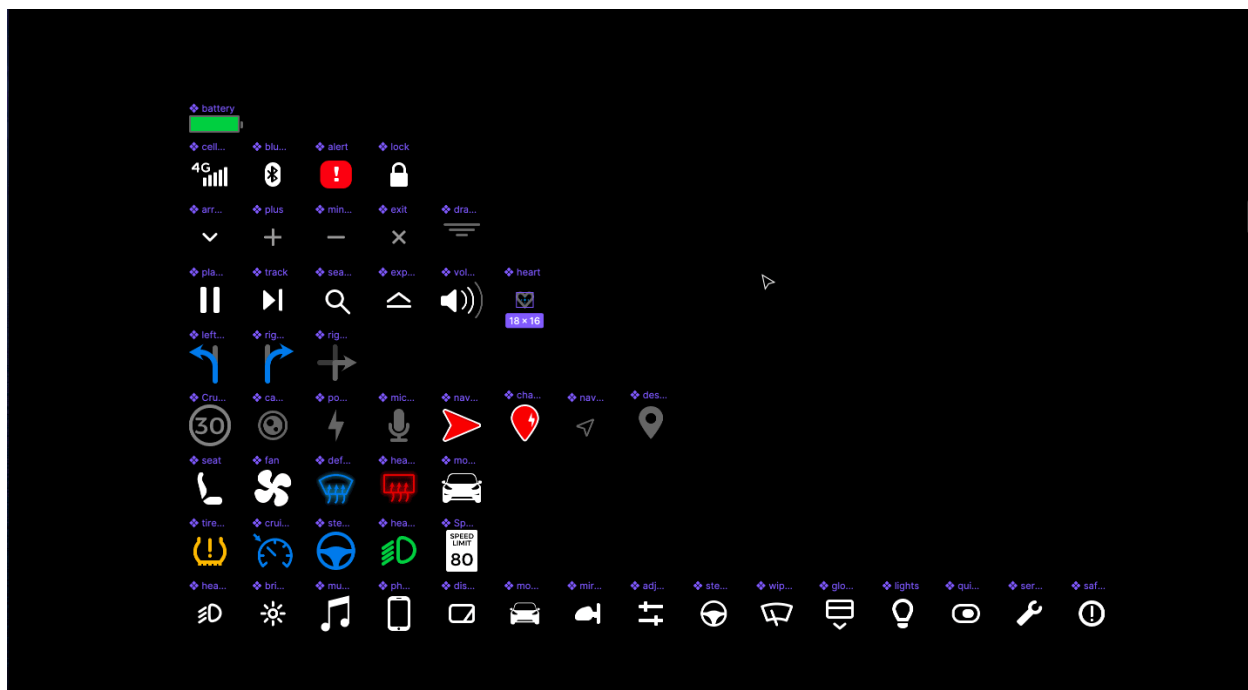


Рисунок 4.67 — Keyboard Zooms into Selection

Створення нового проекту може відбуватися як з чистого аркуша, так і на основі готових шаблонів, які Figma надає для різних типів проектів. Шаблони включають в себе найкращі практики та стандартні компоненти для веб-сайтів, мобільних додатків, презентацій та інших типів проектів.

Система артбордів (frames) в Figma аналогічна до монтажних областей в інших графічних редакторах, але має ряд унікальних особливостей. Артборди можуть мати різні розміри та орієнтації в рамках одного проекту, що дозволяє одночасно працювати з десктопними, планшетними та мобільними версіями інтерфейсу.

Навігація по проекту здійснюється за допомогою інструментів масштабування, панорамування та швидкого переходу між артбордами. Система шарів забезпечує організацію елементів дизайну та контроль над їхньою видимістю та взаємодією.

### 4.3.6 Інструменти та можливості Фігми

Інструментальний арсенал Figma включає в себе як базові інструменти для створення та редагування графічних елементів, так і спеціалізовані функції, призначені специфічно для UI/UX дизайну. Панель інструментів розташована у верхній частині інтерфейсу та надає швидкий доступ до найбільш використовуваних функцій.

Інструмент вибору (Selection tool) є основним інструментом для маніпуляцій з об'єктами - переміщення, масштабування, обертання та групування елементів. Він також надає доступ до контекстних налаштувань обраного об'єкта через панель властивостей в правій частині інтерфейсу.

Інструменти створення базових форм включають прямокутники, еліпси, багатокутники та зірки. Кожна форма має набір параметрів, які можна налаштовувати - радіус закруглення кутів для прямокутників, кількість сторін для багатокутників, кількість променів для зірок тощо.

Інструмент ручки (Pen tool) призначений для створення довільних векторних форм та кривих Безьє. Він дозволяє створювати складні геометричні форми, логотипи, ілюстрації та інші графічні елементи з високою точністю.

Текстовий інструмент надає широкі можливості для роботи з типографікою, включаючи підтримку веб-шрифтів, налаштування міжрядкових та міжсимвольних відстаней, створення текстових стилів та їх повторне використання в проекті.

Інструменти для роботи з растровими зображеннями дозволяють імпортувати, обрізати та застосовувати базові фільтри до фотографій та інших растрових матеріалів.

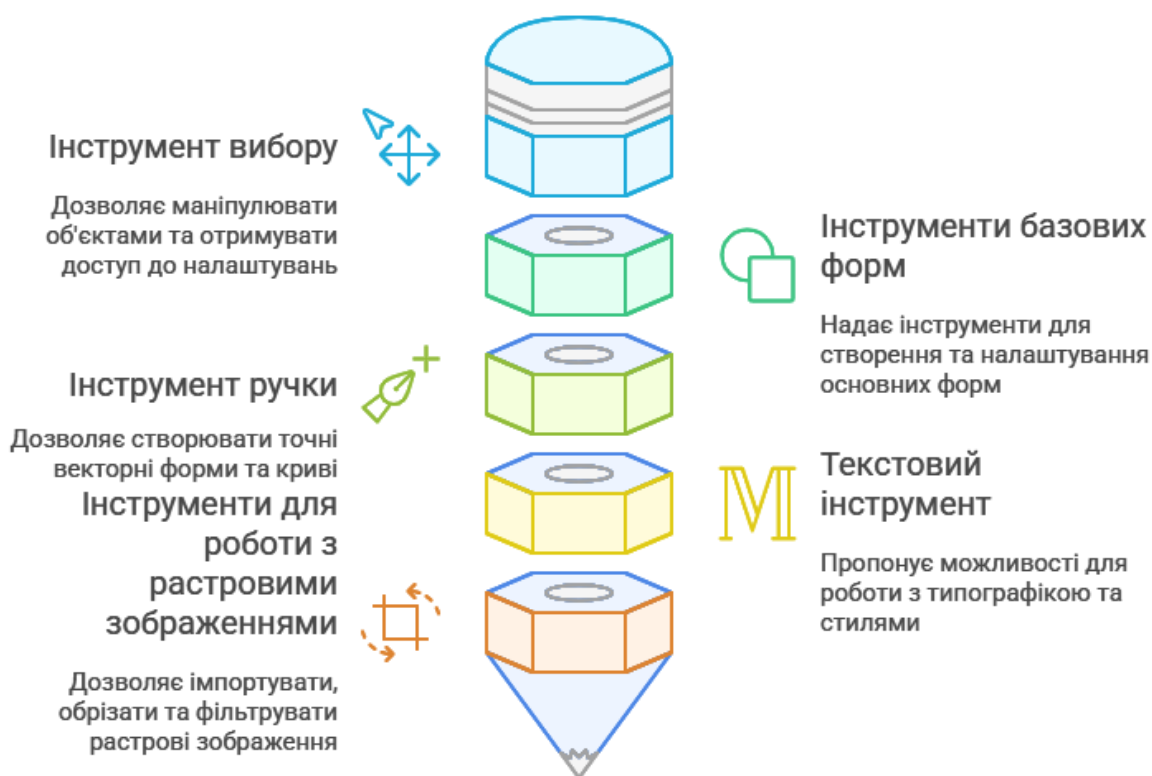


Рисунок 4.68 — Огляд інструментів Figma

Розглянемо панель інструментів для роботи з графікою та вирівнювання об'єктів, вибору шрифтів, створення ефектів для шарів та фігур.

Фрейм або артборд є основним елементом дизайну у Фігмі. Це закінчений документ, який може бути сторінкою сайту або екраном мобільного додатка. Ви можете встановити розмір кадру самостійно або вибрати готові розміри екранів популярних пристроїв - ноутбуків, годинників, планшетів - в панелі інструментів праворуч.

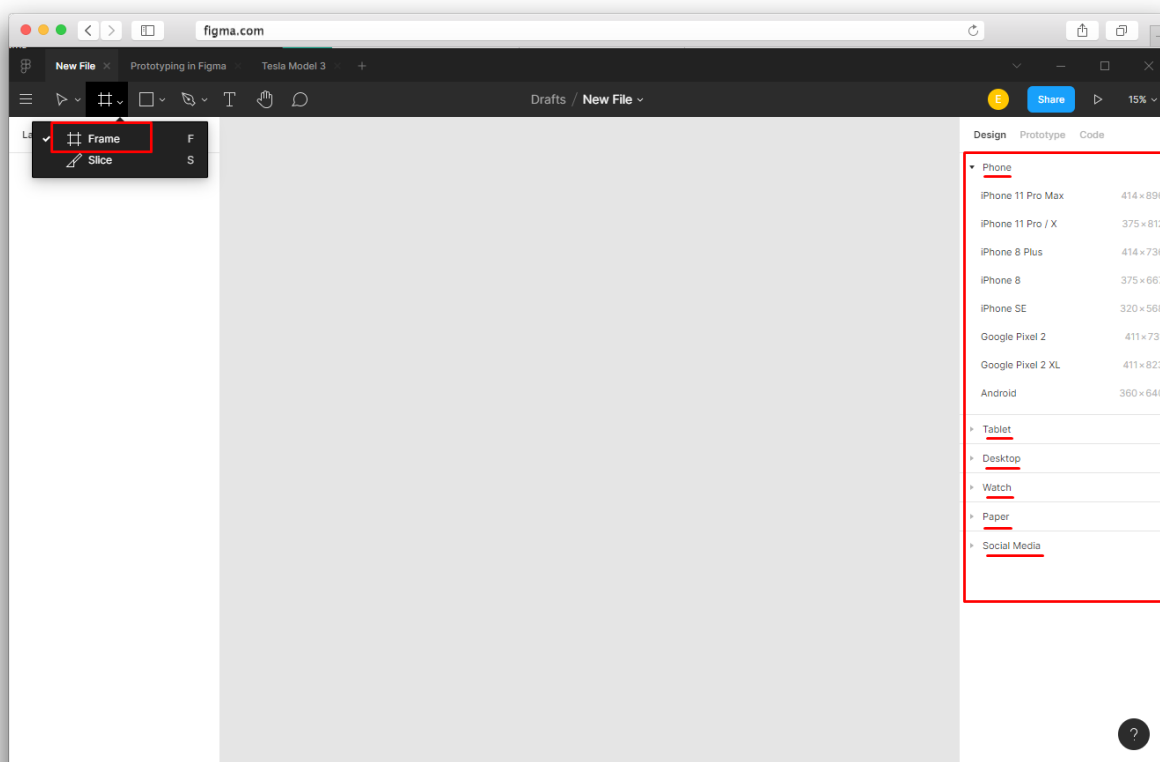


Рисунок 4.69 — Фрейм

Фрейм поєднує об'єкти в собі. Коли ви створюєте дизайн усередині кадру, до панелі властивостей зліва будуть додаватися шари. Шари у Фігмі — це вміст вашого кадру: об'єкти, текст, фотографії.

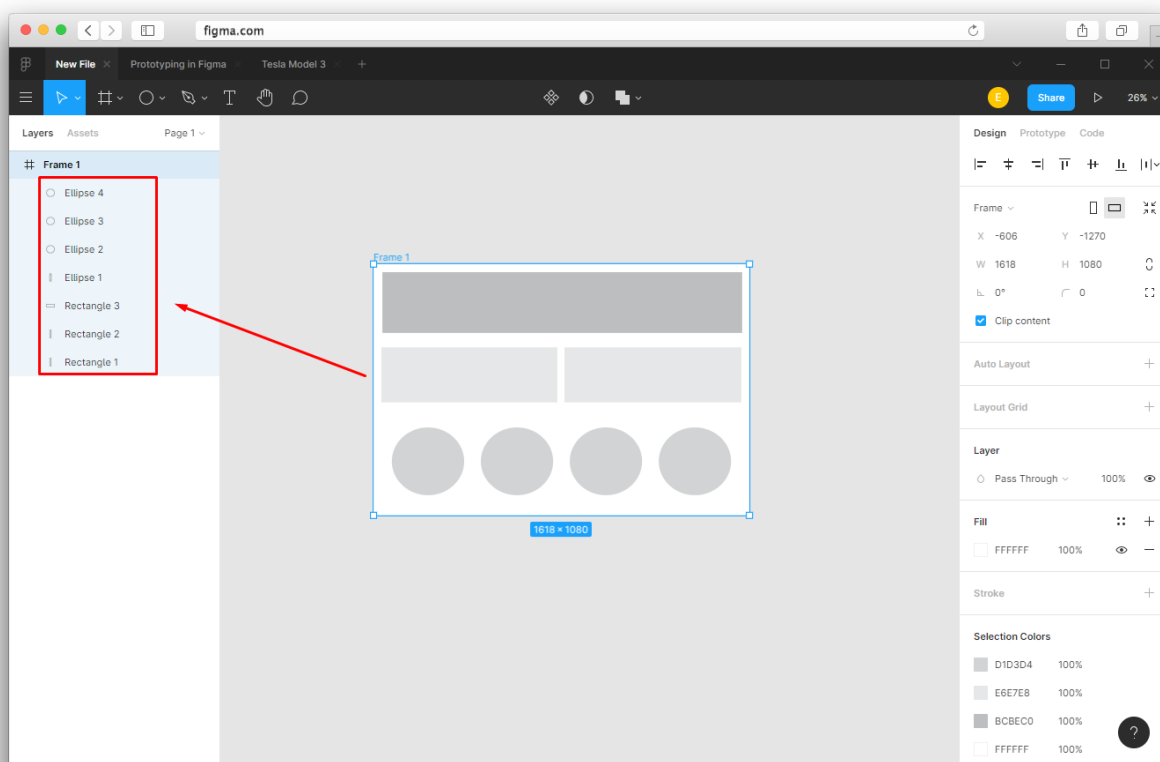


Рисунок 4.70 — Шари

Ви можете згрупувати окремі об'єкти у кадр або розбити його на об'єкти клавішами **Ctrl + Alt + G** або **Cmd + Opt + G**. Використовуйте панель праворуч, щоб вирівняти об'єкти у кадрі по осі X/Y або відстань між елементами.

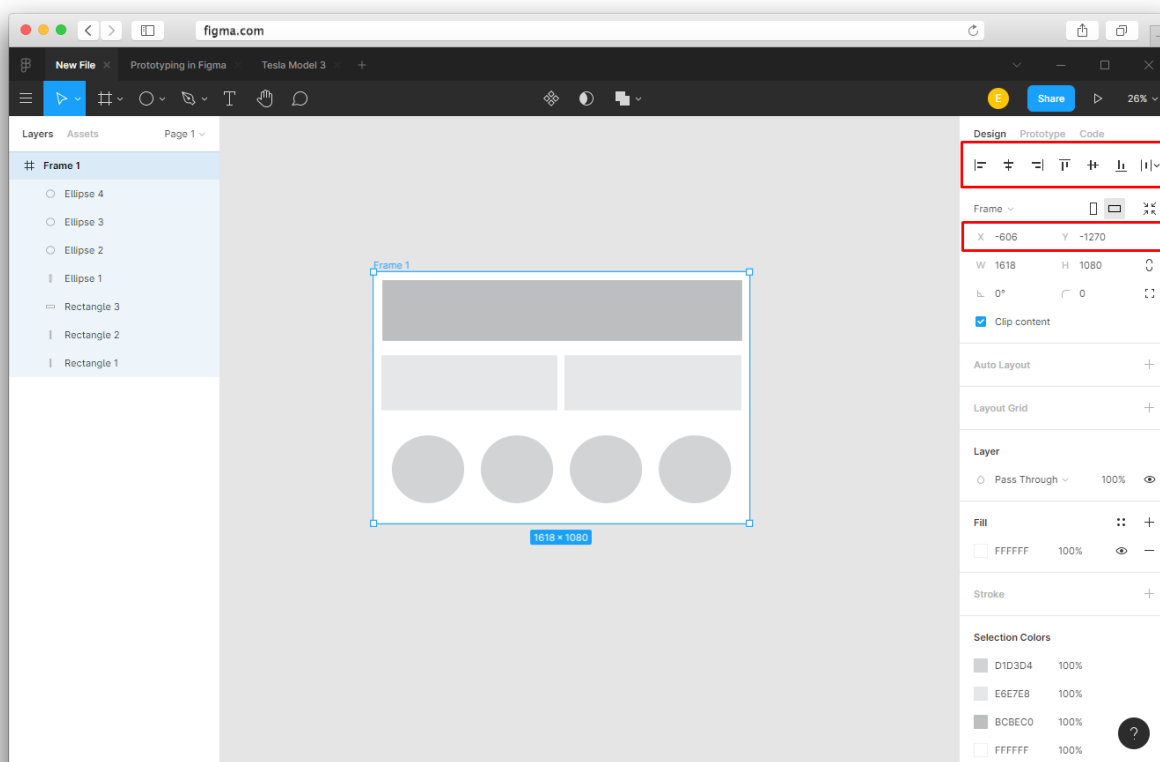


Рисунок 4.71 — Згрупувати окремі об'єкти у кадр

Об'єкти всередині кадру можна групувати. Допустимо, ви робите кілька сторінок сайту. На кожній сторінці внизу екрана будуть контакти: телефон і електронна пошта. Згрупуйте об'єкти з меню **Object** → **Group Selection** або клавішами **Ctrl (Cmd) + G**. Ви побачите зміни в шарах об'єктів — вони об'єдналися в групу. Тепер ви можете переміщати всі об'єкти разом.

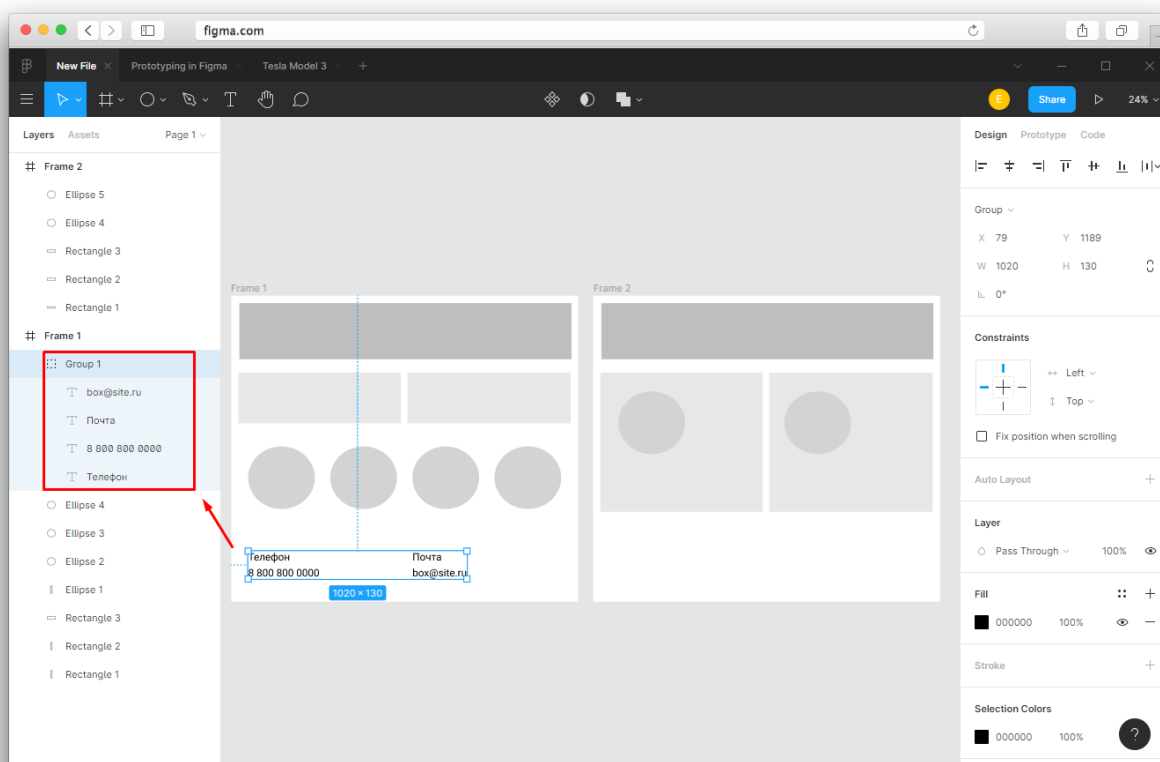


Рисунок 4.72 — Групування

Додатково ви можете обмежувати поведінку елементів у кадрі. Наприклад, ви хочете, щоб кнопка вашого дизайну завжди була в кутку екрана, навіть якщо розмір кадру зміниться. Виберіть об'єкт у кадрі та використовуйте інструмент Constraints у панелі властивостей праворуч.

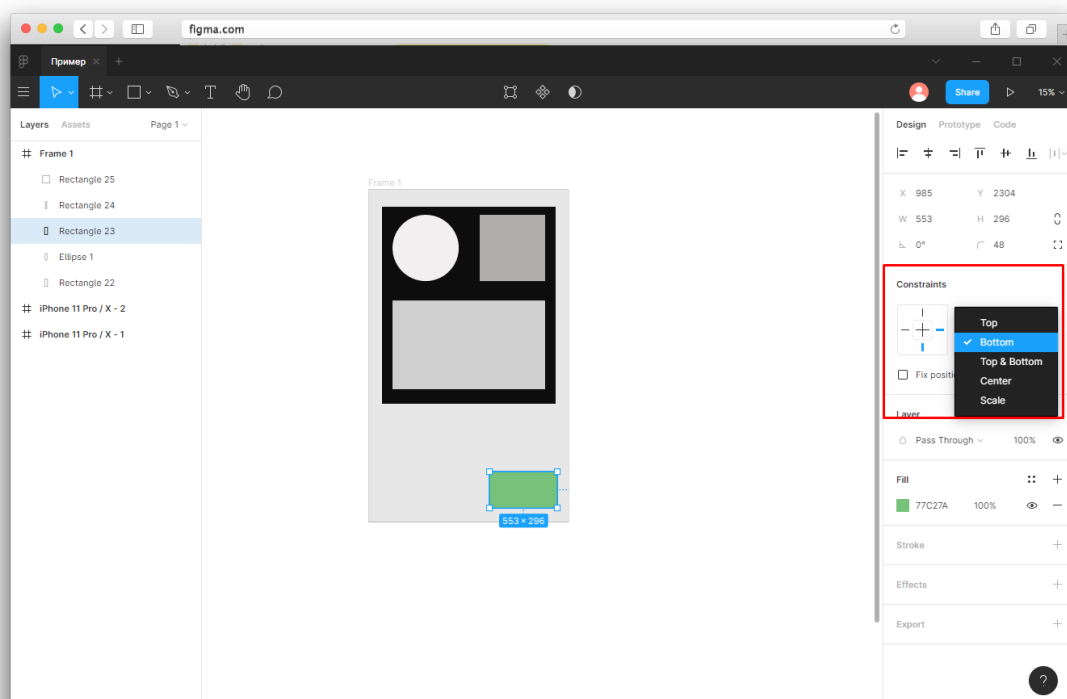


Рисунок 4.73 — Constraints

Фрейм це закінчений дизайн, його можна завантажити одним файлом. У разі збереження переконайтеся, що всі об'єкти знаходяться всередині кадру. Якщо випадково перемістити шар об'єкта за межі кадру, можна втратити частину дизайну. Перед скачуванням, перевірте прев'ю файлу та виберіть потрібний формат.

#### 4.3.7 Модульна сітка

Модульна сітка в Figma є фундаментальним інструментом для створення структурованих та візуально збалансованих дизайнів, особливо важливим для веб-проектів та мобільних додатків. Система сіток в Figma включає кілька типів, кожен з яких призначений для вирішення конкретних дизайнерських задач.



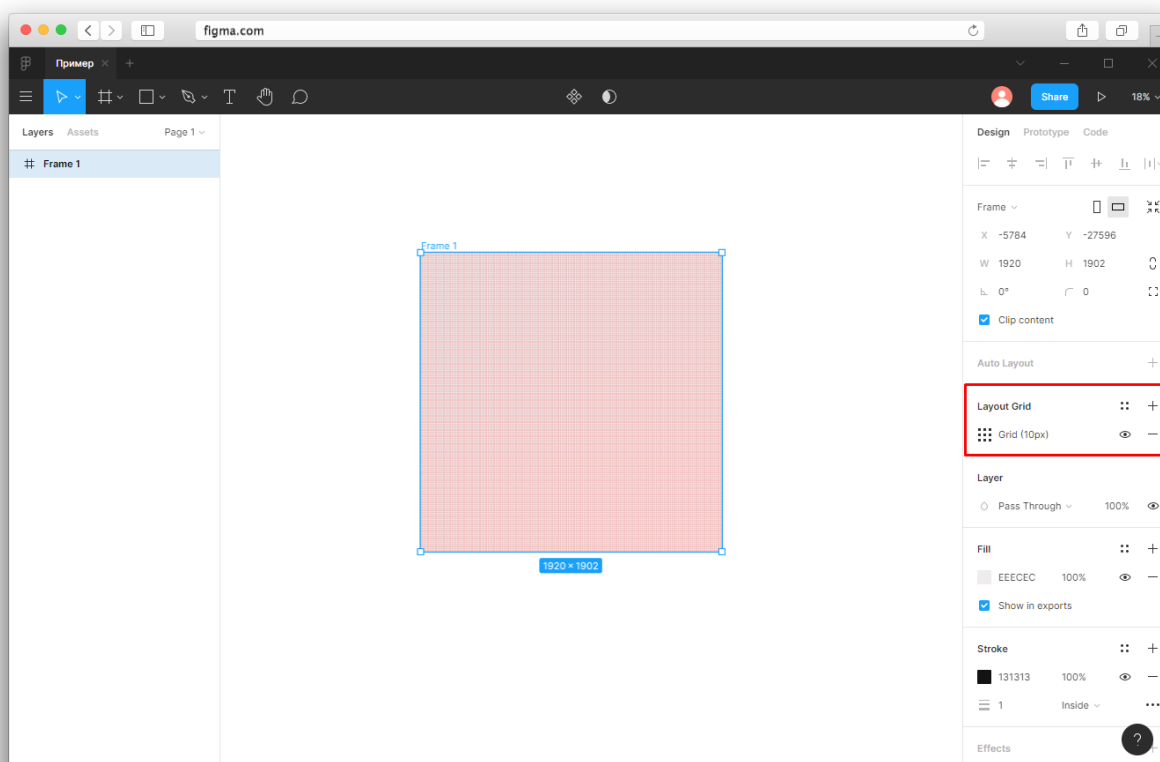


Рисунок 4.74 — Сітка

Колонкова сітка (Column Grid) є найбільш поширеним типом для веб-дизайну та працює за принципом поділу простору на вертикальні колонки з налаштовуваними проміжками між ними. Традиційно використовують 12-колонкову сітку для десктопних версій, яка може адаптуватися до 8, 4 або 2 колонок для планшетних та мобільних пристроїв відповідно.

Рядкова сітка (Row Grid) організує контент по горизонтальним лініям та особливо корисна для створення ритму та візуальної ієрархії в дизайні. Висота рядків зазвичай базується на висоті рядка основного тексту проекту, що забезпечує типографічну послідовність.

Квадратна сітка (Square Grid) створює рівномірну мережу квадратних комірок та найчастіше використовується для іконок, галерей зображень або інших елементів, які потребують рівномірного розподілу простору.

Налаштування сітки включає параметри розміру колонок, відстаней між ними (gutters), зовнішніх відступів (margins) та кольору відображення. Figma

дозволяє створювати кілька сіток одночасно на одному артборді, що дає можливість комбінувати різні типи для досягнення оптимального результату.

Встановіть кількість колонок, їх колір та прозорість. Ви можете налаштувати тип сітки – зробити її адаптивною, вирівняти по центру чи краю.

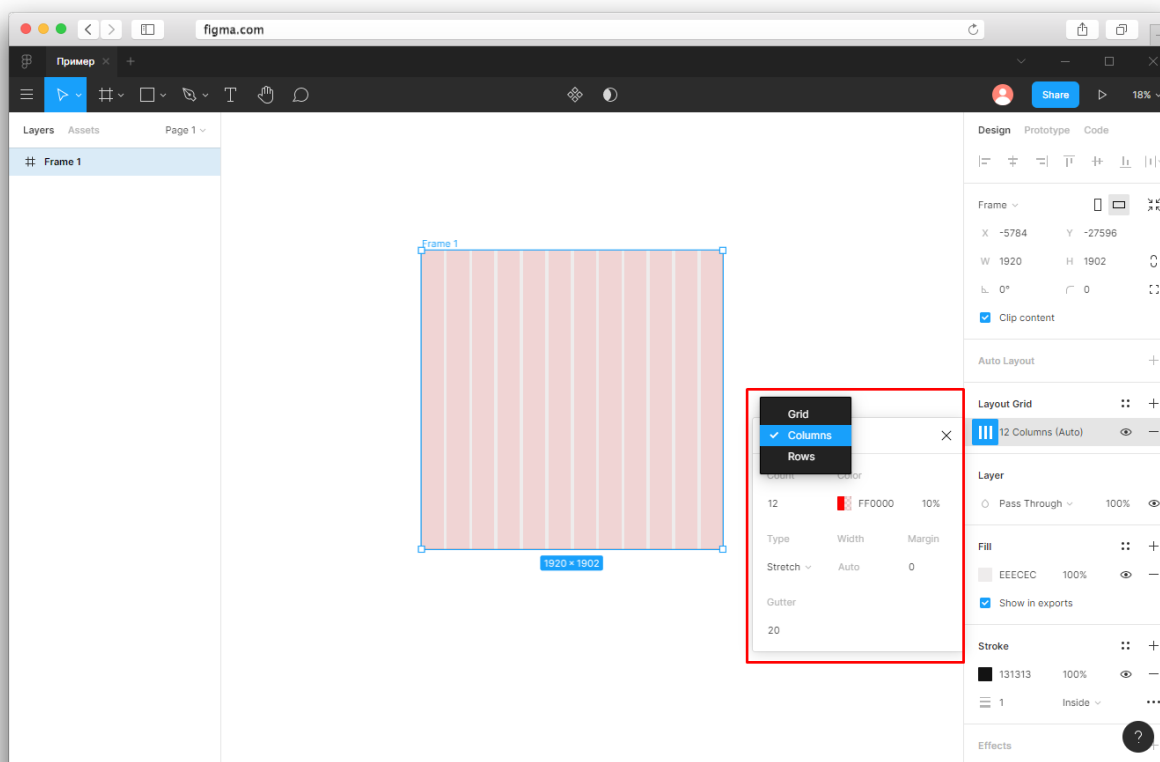


Рисунок 4.75 — Сітка

Деяким дизайнерам зручно додавати вертикальний ритм, щоб переміщувати об'єкти по сітці та налаштовувати відстань між елементами. Для цього натисніть "+" у блоці Layout Grid ще раз - з'явиться додаткова сітка з кроком 10 пікселів.

Щоб елементи дизайну не наїжджали на краї кадру, використовуйте монтажні області. Для контролю відступів можна використовувати додаткову сітку. Створіть одну колонку ліворуч або праворуч потрібної ширини, зробіть непомітний колір.

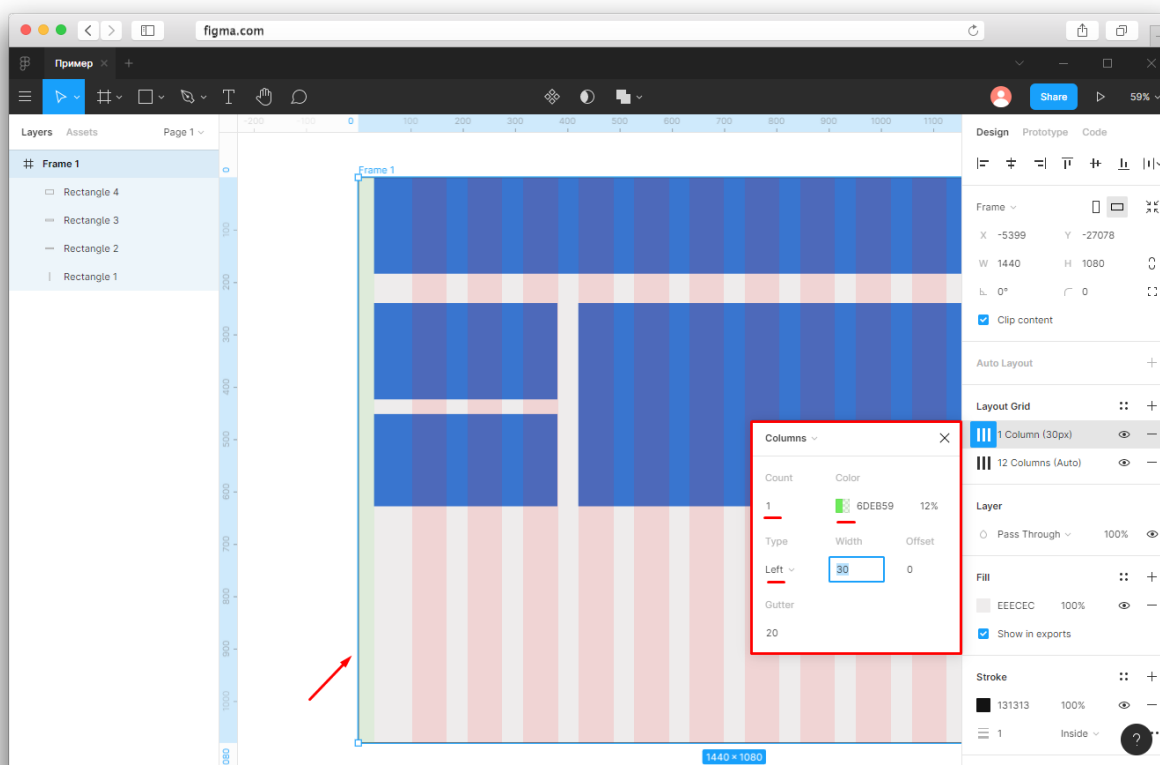
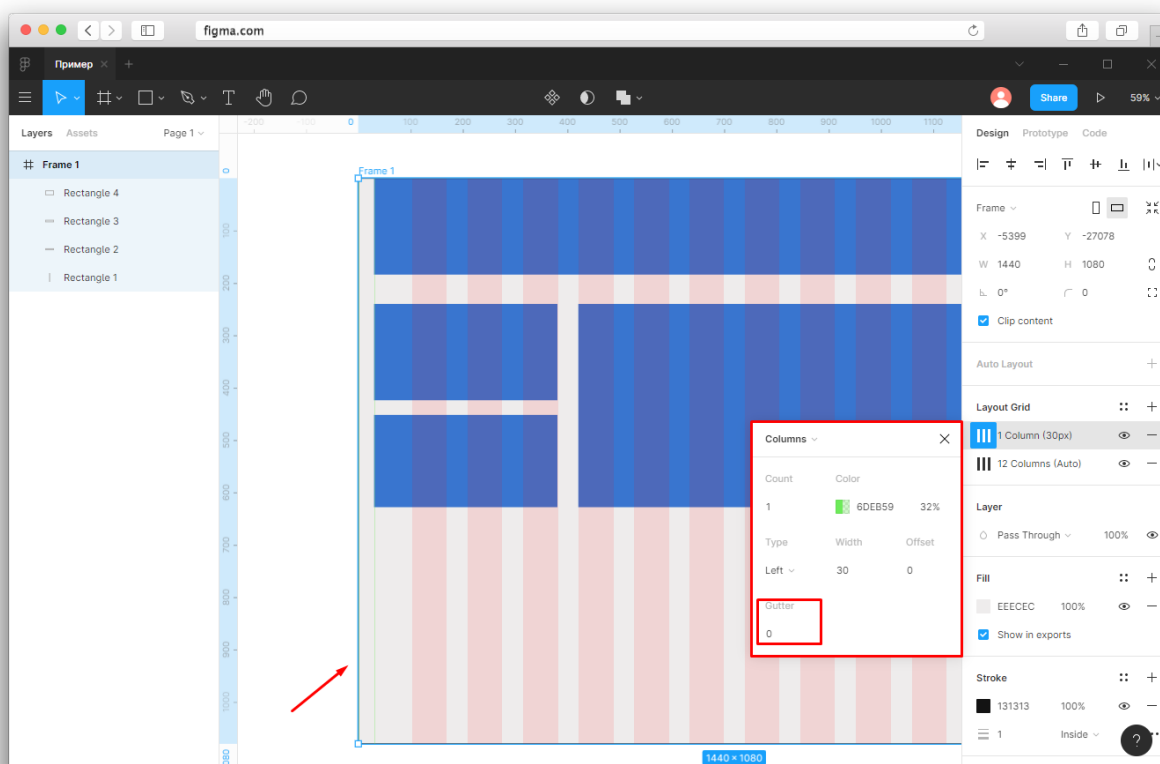


Рисунок 4.76 — Сітка

Для зручності можна виставити параметр  $Gutter = 0$  і перетворити колонку на лінію з одним видимим краєм.



### Рисунок 4.77 — Сітка

Використання модульної сітки значно прискорює процес дизайну та забезпечує консистентність розташування елементів. Вона також полегшує комунікацію з розробниками, оскільки надає чіткі керівні принципи для імплементації дизайну в коді.

#### 4.3.8 Векторні форми

Векторні форми в Figma є основою для створення масштабованих графічних елементів, які зберігають свою якість незалежно від розміру. На відміну від растрових зображень, векторна графіка базується на математичних формулах, що описують геометричні фігури, лінії та криві.

Базові векторні форми включають прямокутники, кола, багатокутники та зірки, кожна з яких має власний набір налаштувань. Прямокутники можуть мати закруглені кути з різними радіусами для кожного кута окремо, що дозволяє створювати складні форми кнопок та інтерфейсних елементів.

Еліпси та кола використовуються для створення аватарів, індикаторів стану, декоративних елементів та основ для іконок. Можливість перетворення кола в сектор або дугу розширює спектр застосування цього інструменту.

Багатокутники з налаштовуваною кількістю сторін від трикутників до складних багатогранників можуть використовуватися для створення геометричних паттернів, логотипів та декоративних елементів. Радіус закруглення кутів додає м'якості та сучасності формам.

Зірки з можливістю налаштування кількості променів та їх довжини корисні для створення рейтингових систем, декоративних елементів та спеціальних індикаторів.

Буліві операції (Boolean operations) дозволяють комбінувати прості форми в складні об'єкти через операції об'єднання, віднімання, перетину та виключення. Це потужний інструмент для створення логотипів, іконок та інших графічних елементів без необхідності малювання з нуля.

Створюйте векторні об'єкти інструментом Shape Tool. З його допомогою можна малювати елементи інтерфейсу, наприклад, іконки.

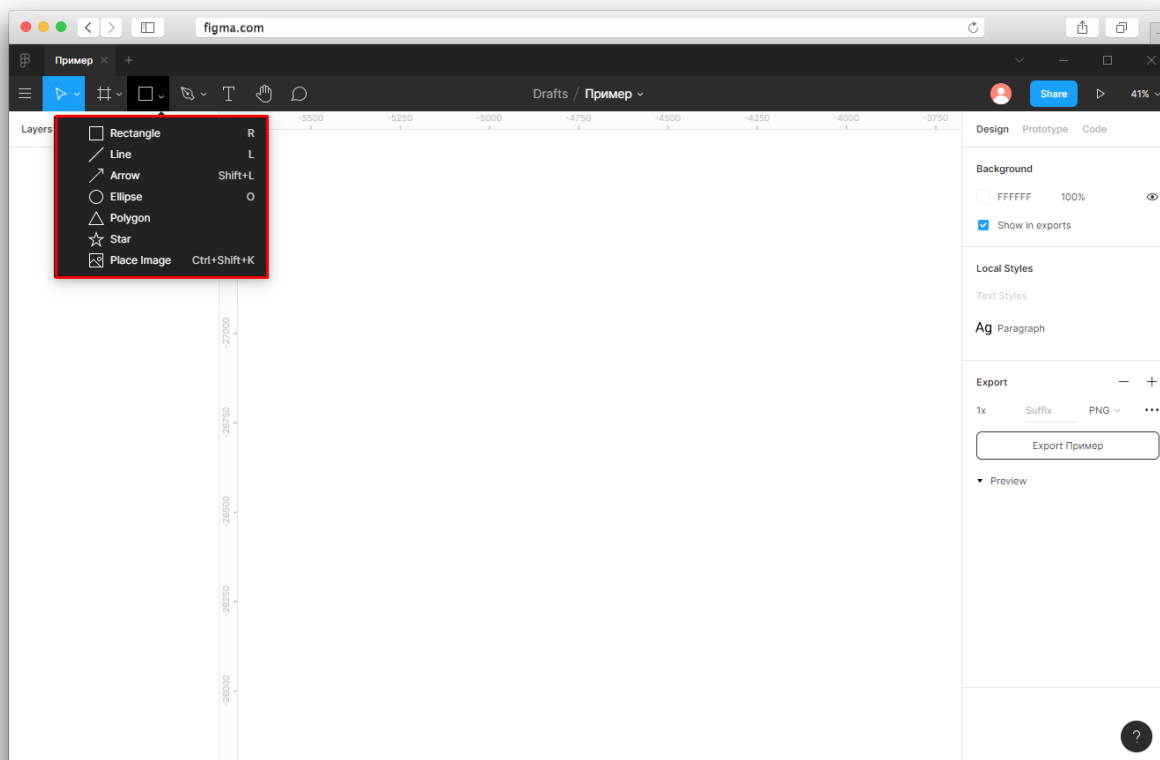


Рисунок 4.78 — Векторні об'єкти

Основні векторні об'єкти – прямокутник, лінія, трикутник, стрілка, коло, зірка. Ви можете вставляти об'єкти довільної форми або натиснути клавішу Shift і вставити об'єкт правильної форми. Якщо натиснути Alt, об'єкт розтягується з центру. Панель властивостей об'єктів знаходиться праворуч. Розглянемо основні функції роботи з об'єктами та покажемо, як зробити у Фігмі іконку «Закладка», використовуючи прямокутник довільної форми.

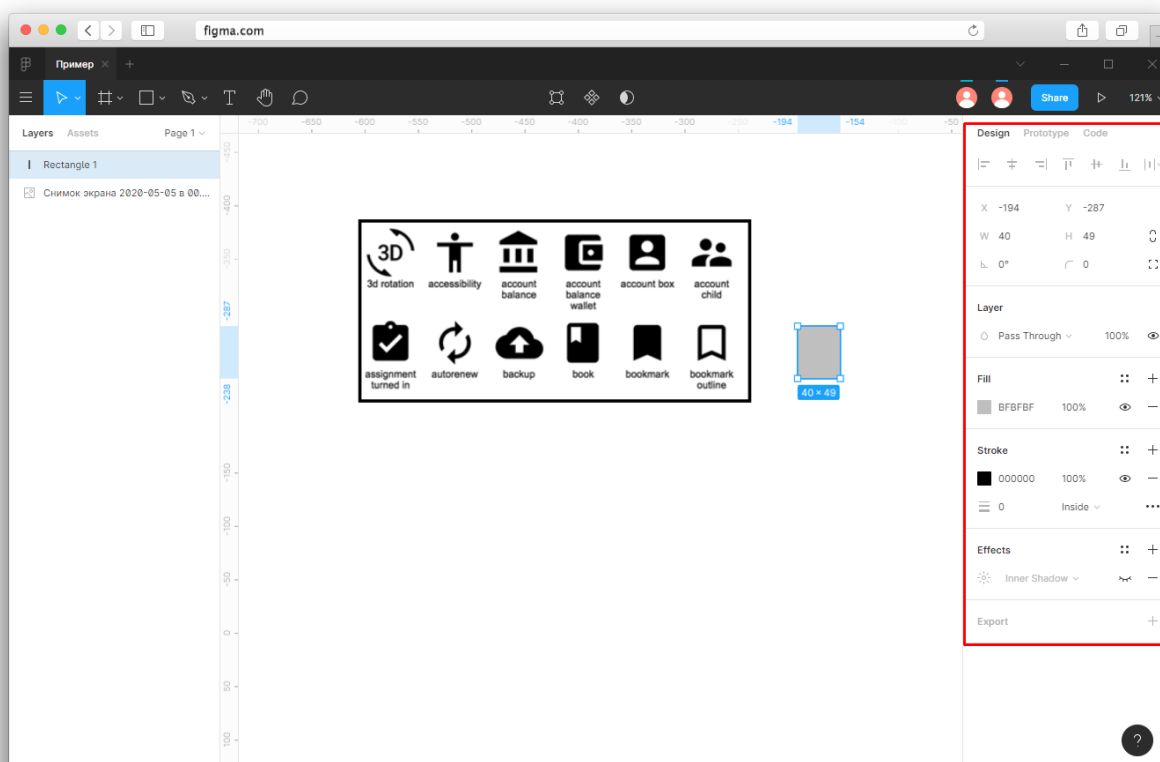


Рисунок 4.79 — Векторні об'єкти

Переміщайтеся між полями панелі властивостей за допомогою кнопки Tab. У верхній частині панелі можна змінити розташування об'єкта, розтягнути по осі X/Y. Виберемо Corner Radius, щоб заокруглити кути нашого прямокутника, як у закладки з прикладу. За промовчанням властивість застосовується до всіх сторін об'єкта. Щоб вибрати два верхні кути, натисніть Independent Radius.

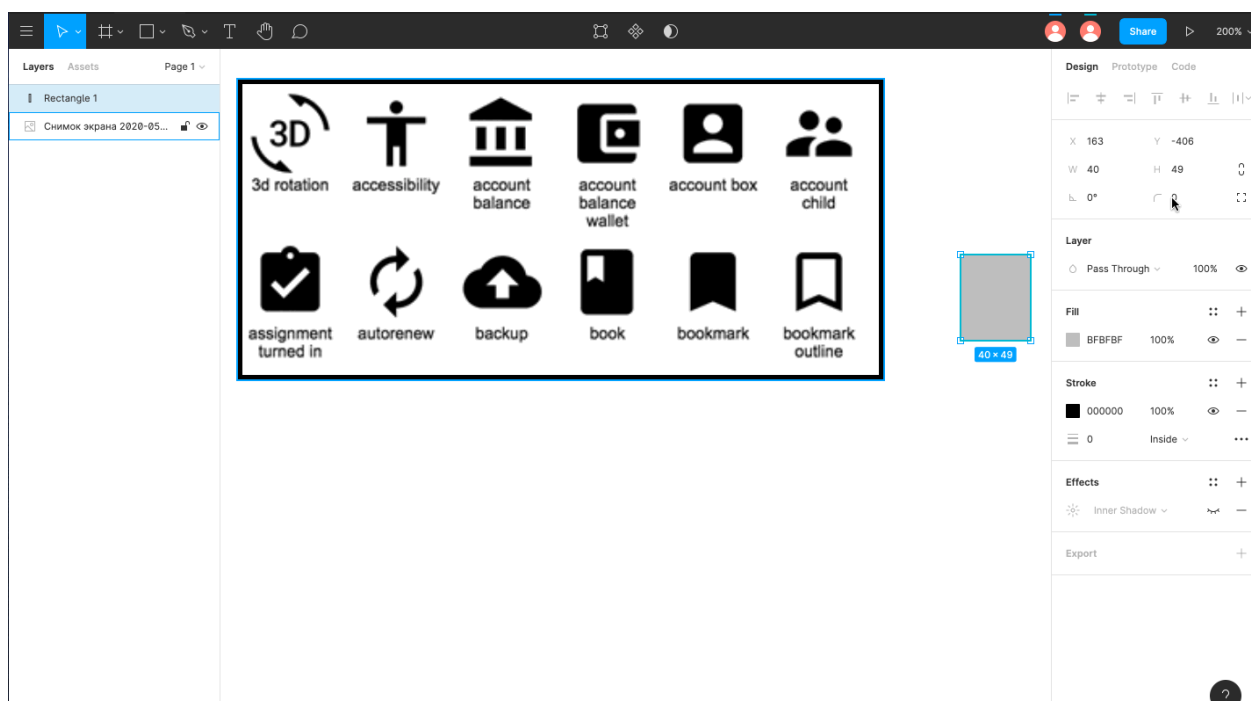


Рисунок 4.80 — Векторні об'єкти

Попрацюємо з обведенням об'єкта. Товщина лінії змінюється у пункті Stroke заданим значенням чи мишкою. Заливка об'єкта нам не потрібна, можна приховати її в пункті Fill, натиснувши на «очі».

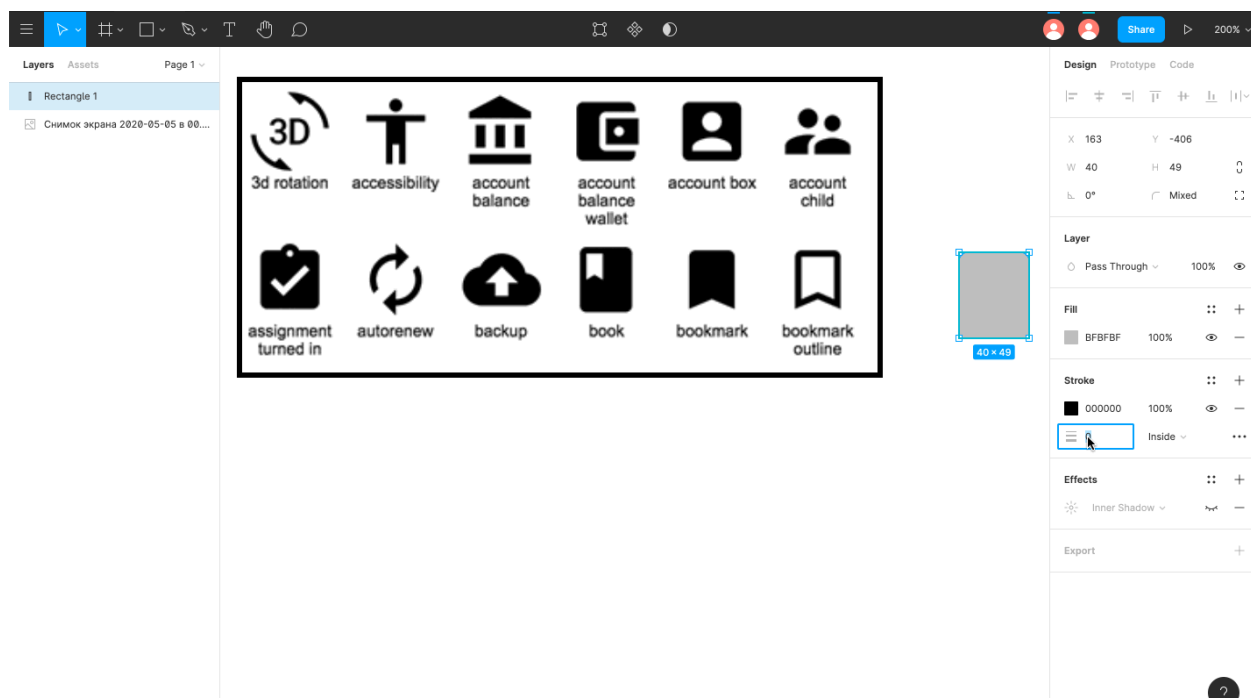


Рисунок 4.81 — Векторні об'єкти

Витягнемо нижню сторону прямокутника і зробимо його схожим на прапор. Щоб редагувати фігуру, натисніть на неї двічі.

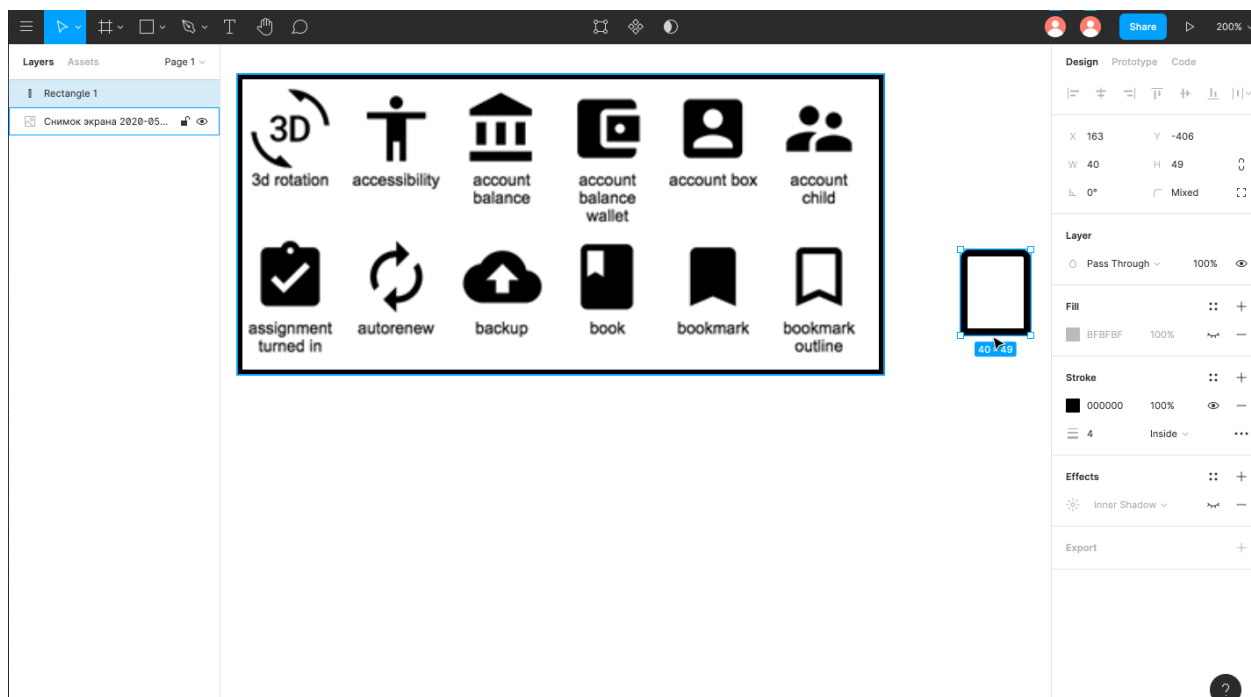


Рисунок 4.82 — Векторні об'єкти

Редагування фігури активує інструмент Pen Tool. У розділі Effects можна додати тінь або розмиття об'єкта. Щоб зберегти іконку, перейдіть до розділу Export та виберіть формат SVG.

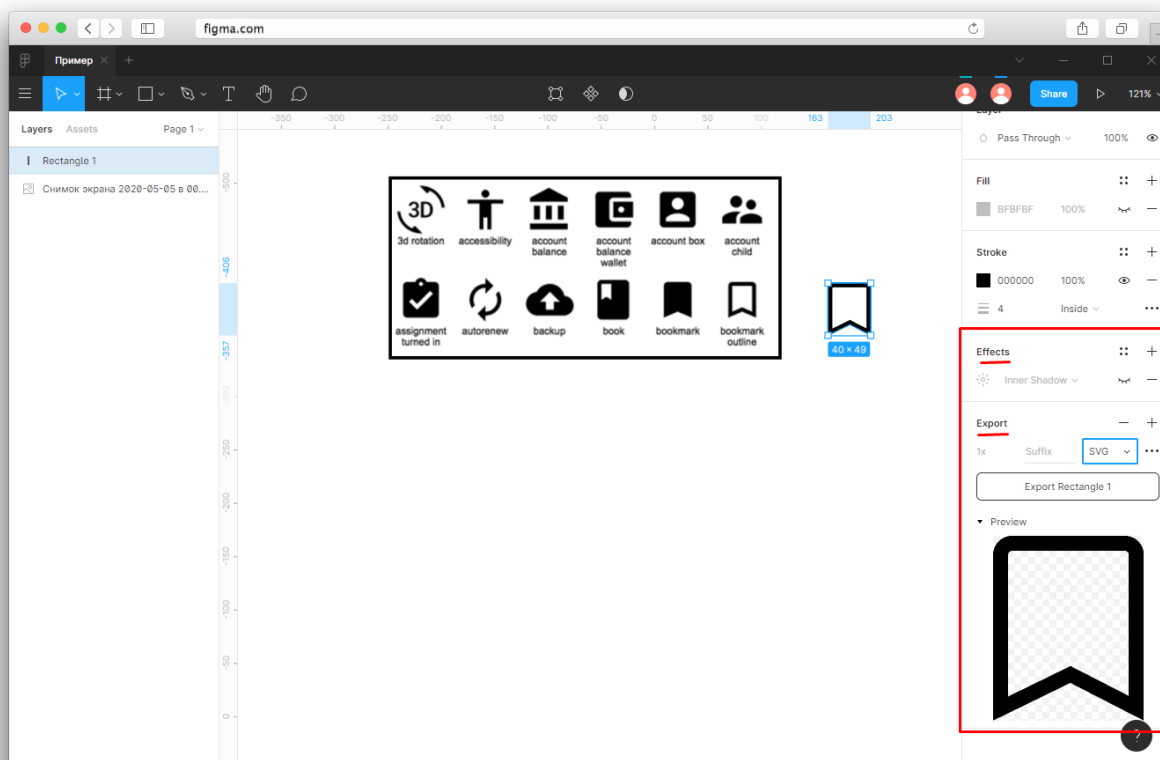


Рисунок 4.83 — Векторні об'єкти



### 4.3.9 Криві

Робота з кривими в Figma базується на використанні кривих Безьє - математичній моделі, яка дозволяє створювати плавні лінії та форми за допомогою контрольних точок та векторів напрямлення. Інструмент ручки (Pen tool) є основним засобом для створення та редагування кривих.

Створення кривих починається з постановки опорних точок (anchor points), які визначають основну форму лінії або фігури. Кожна точка може мати один з декількох типів - кутова точка створює гострий кут, гладка точка забезпечує плавний перехід, а симетрична точка підтримує однакову кривизну з обох сторін.

Контрольні маркери (handles) дозволяють налаштовувати кривизну сегментів між опорними точками. Довжина та напрямок цих маркерів безпосередньо впливають на форму кривої - довші маркери створюють більш плавні криві, а їх кут визначає напрямок викривлення.

Векторні мережі (Vector Networks) в Figma є унікальною особливістю, яка відрізняє цю платформу від інших векторних редакторів. На відміну від традиційних векторних шляхів, які мають чіткий початок та кінець, векторні мережі дозволяють створювати розгалужені структури з множинними з'єднаннями.

Редагування кривих включає можливості додавання та видалення точок, перетворення типів точок, розривання шляхів та їх з'єднання. Інструмент прямого вибору дозволяє редагувати окремі точки та сегменти без впливу на інші частини об'єкта.

Застосування кривих особливо важливе при створенні логотипів, ілюстрацій, іконок та інших графічних елементів, де потрібна висока точність та плавність ліній.

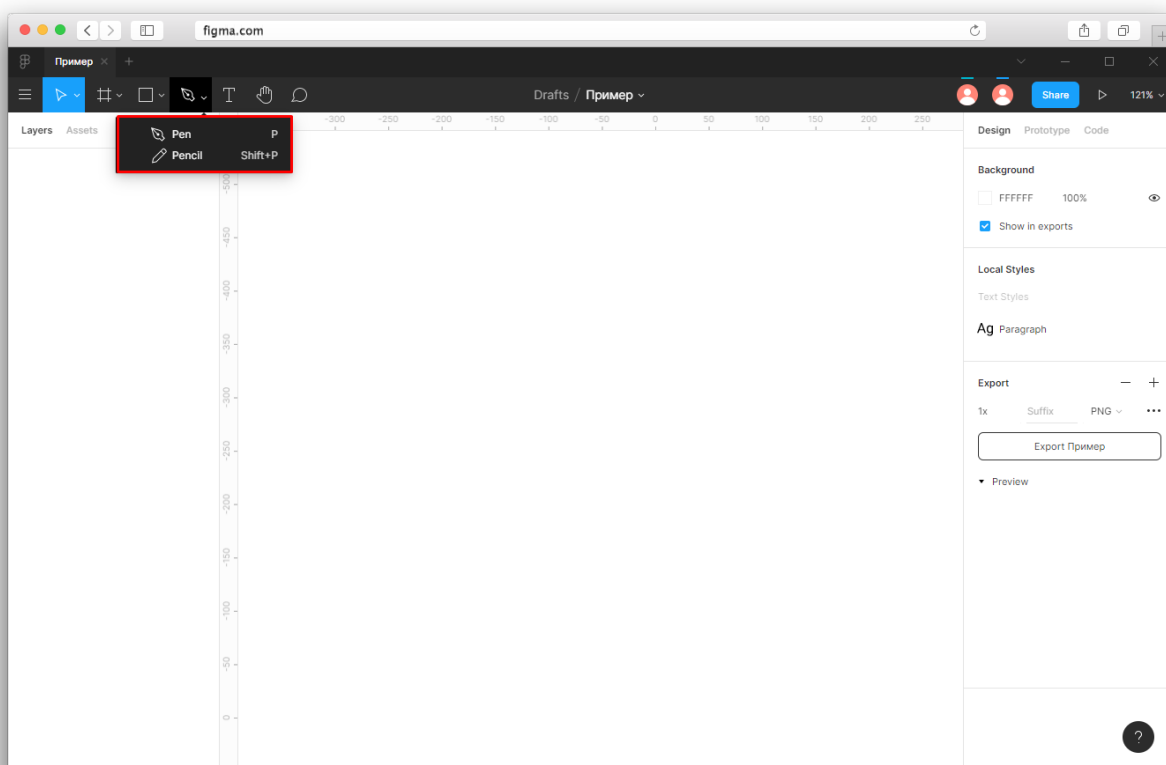


Рисунок 4.84 — Криві

Використовуйте додаткову опцію Bend Tool для заокруглення кривих або Paint Bucket для заливки закритого контуру.

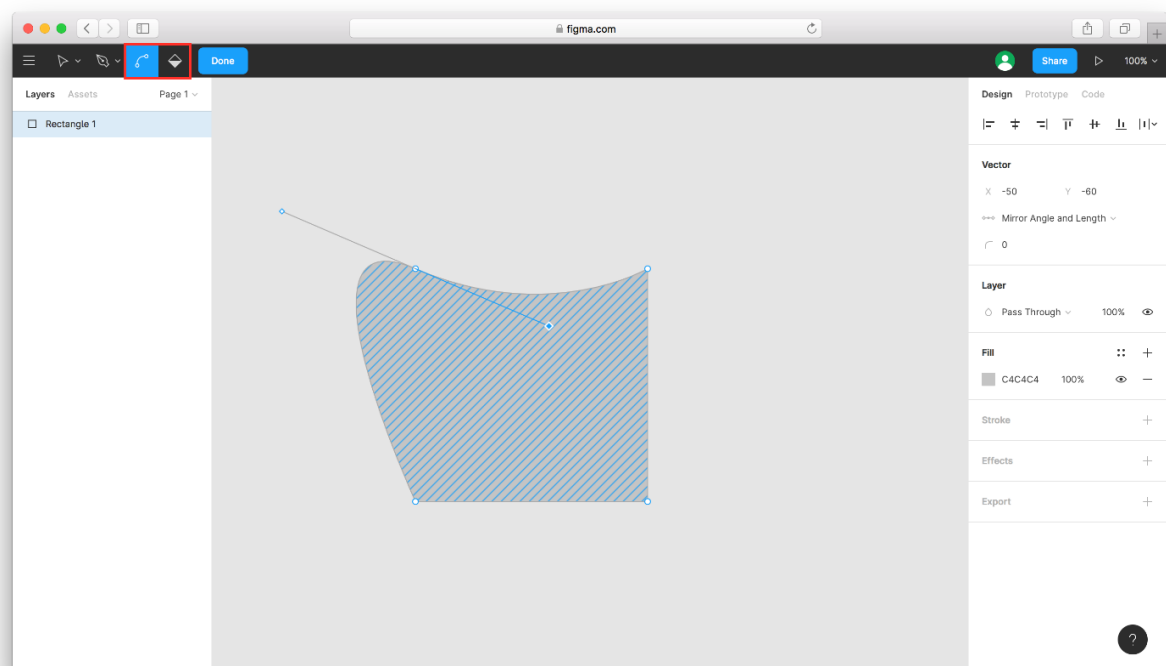


Рисунок 4.85 — Криві

### 4.3.10 Зображення

Робота з растровими зображеннями в Figma охоплює широкий спектр операцій - від простого імпорту до складної обробки та інтеграції з векторними елементами дизайну. Підтримувані формати включають JPEG, PNG, GIF, SVG та WebP, що покриває більшість потреб сучасного веб-дизайну.

Імпорт зображень може здійснюватися кількома способами - перетягування файлів з файлової системи, вставка з буфера обміну або завантаження через діалогове вікно. Figma автоматично оптимізує розмір файлів для забезпечення швидкої роботи з проектами.

Обрізання зображень здійснюється за допомогою спеціального режиму кропу, який дозволяє змінювати видиму область зображення без зміни оригінального файлу. Це особливо корисно при роботі з фотографіями, які потрібно адаптувати під різні формати.

Маски дозволяють приховувати частини зображення за допомогою інших об'єктів. Будь-яка векторна форма може стати маскою, що створює безмежні можливості для творчого оформлення фотографій та графічних елементів.

Заливки зображеннями (Image fills) дозволяють використовувати растрові зображення як текстури для векторних форм. Це корисно для створення патернів, фонових текстур або заповнення складних форм фотографічними матеріалами.

Налаштування відображення включають режими масштабування - заповнення (fill), припасування (fit), обрізання (crop) та мозаїки (tile), кожен з яких по-різному адаптує зображення до розмірів контейнера.

Додайте одне або кілька зображень до макету через панель File, інструмент Place Image або просто перетягніть їх з робочого столу.

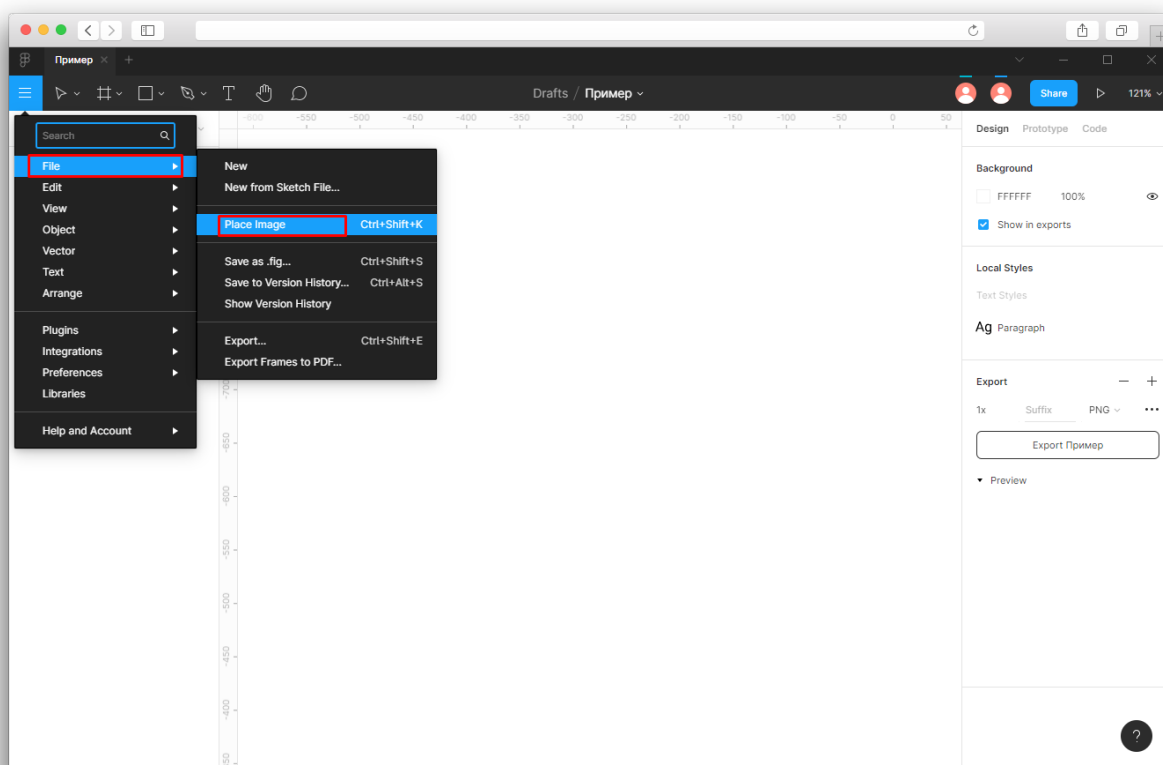


Рисунок 4.86 —Зображення

У Фігмі фото вставляється як share, а не окремий об'єкт. По суті, ми заливаємо зображенням фрейм — прямокутник. Тому ми можемо змінити заповнення кадру параметрами Fill, Fit, Crop, Tile.

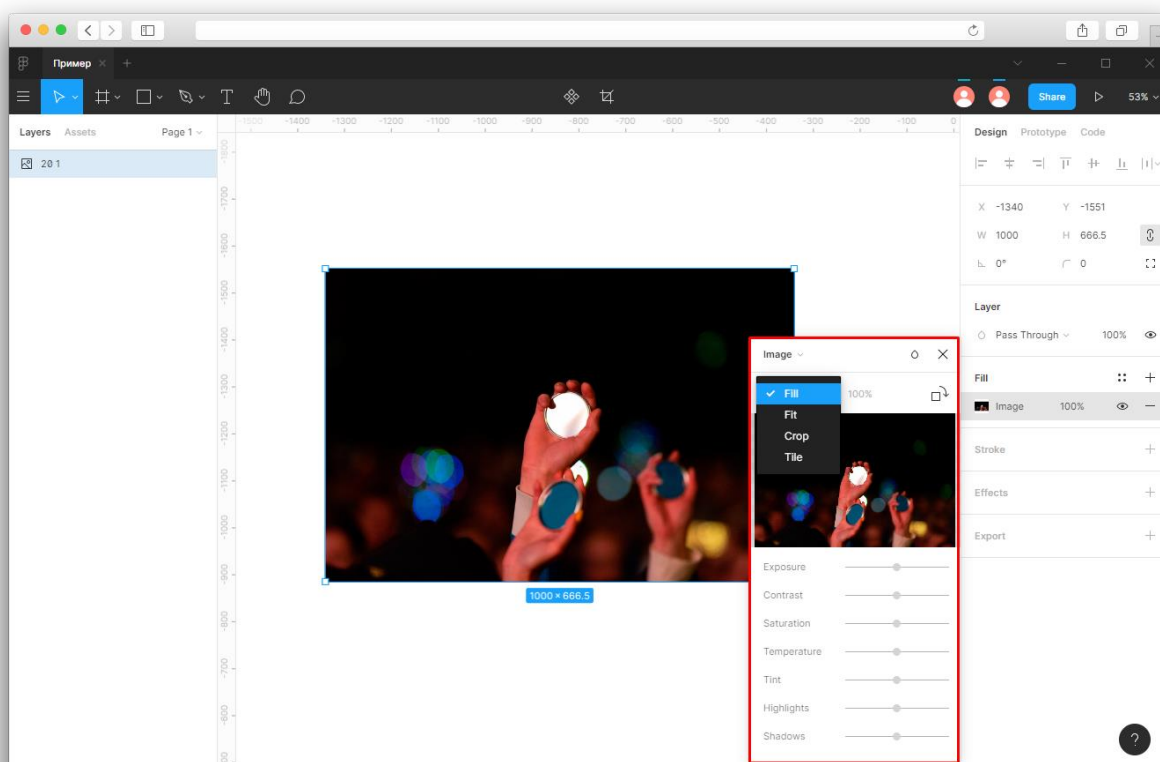


Рисунок 4.87 —Зображення

З увімкненим параметром Fill , зображення заповнює весь кадр. При цьому складно дотриматися його пропорції і картинка може обрізатися.

З увімкненим параметром Fit , зображення відображається у кадрі повністю. Якщо пропорції кадру не співпадають із пропорціями картинки, у ньому з'явиться порожній простір.

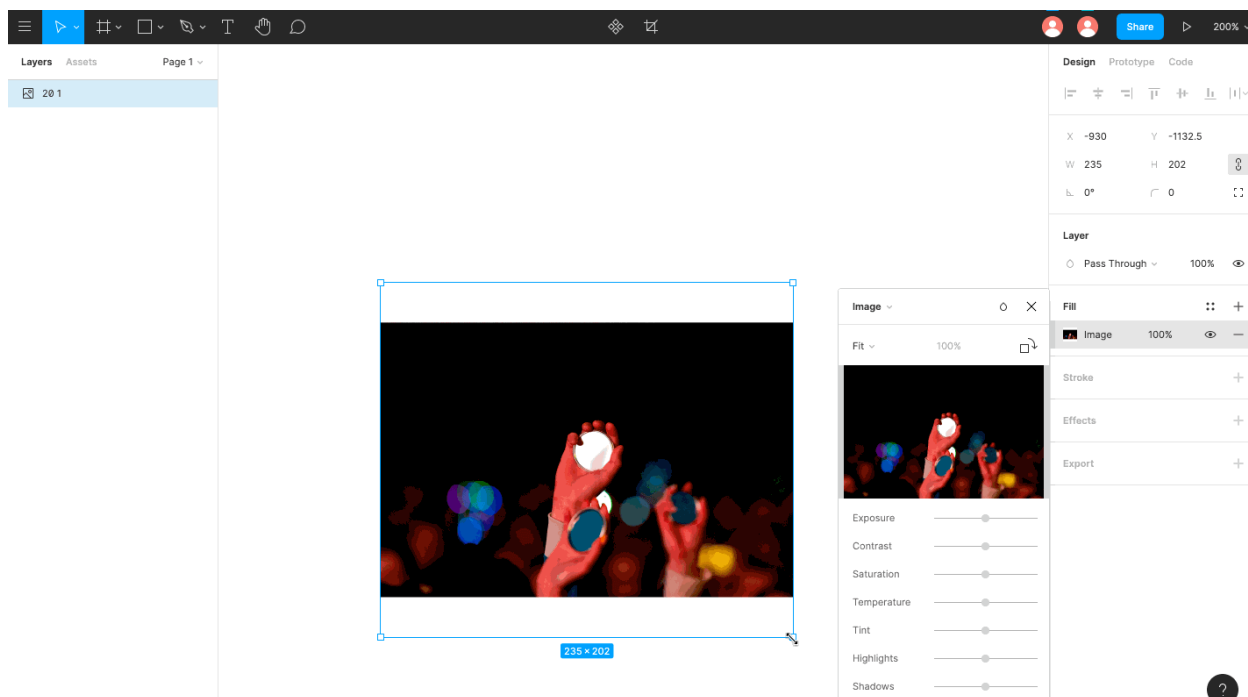


Рисунок 4.88 —Зображення

З увімкненим параметром Crop можна наблизити потрібний ракурс зображення, обрізавши «зайві» частини.

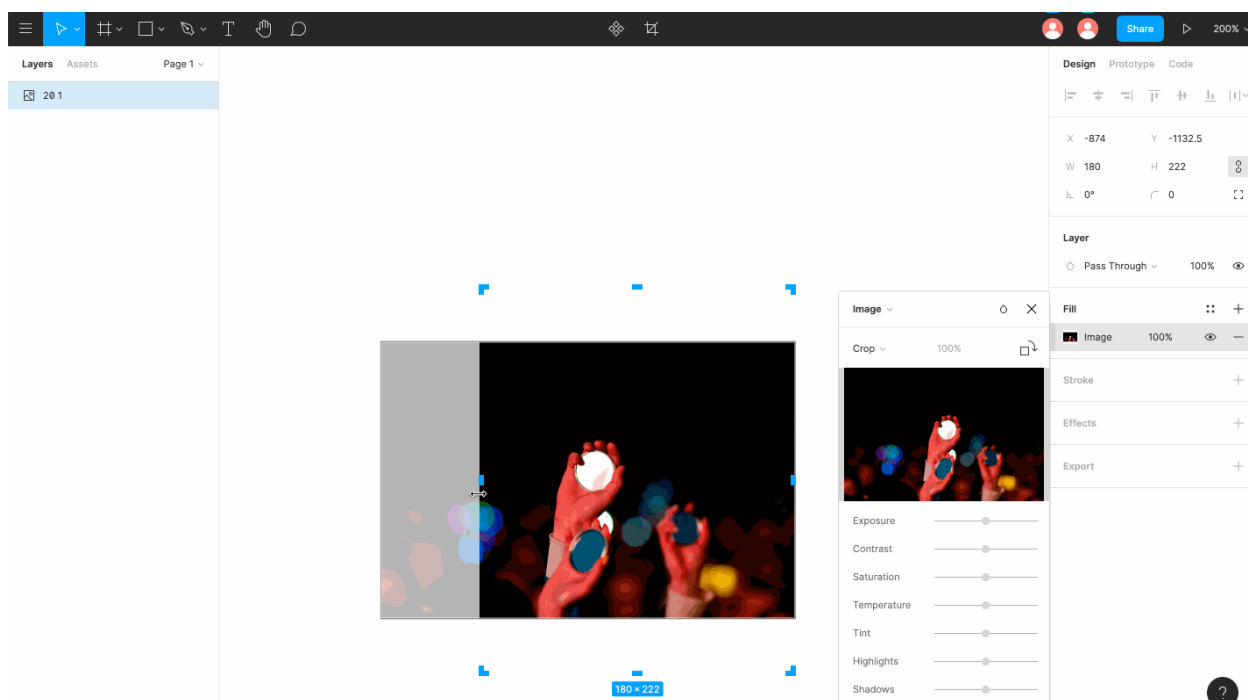


Рисунок 4.89 —Зображення

З увімкненим параметром Tile , ви зможете створити патерн або візерунок.

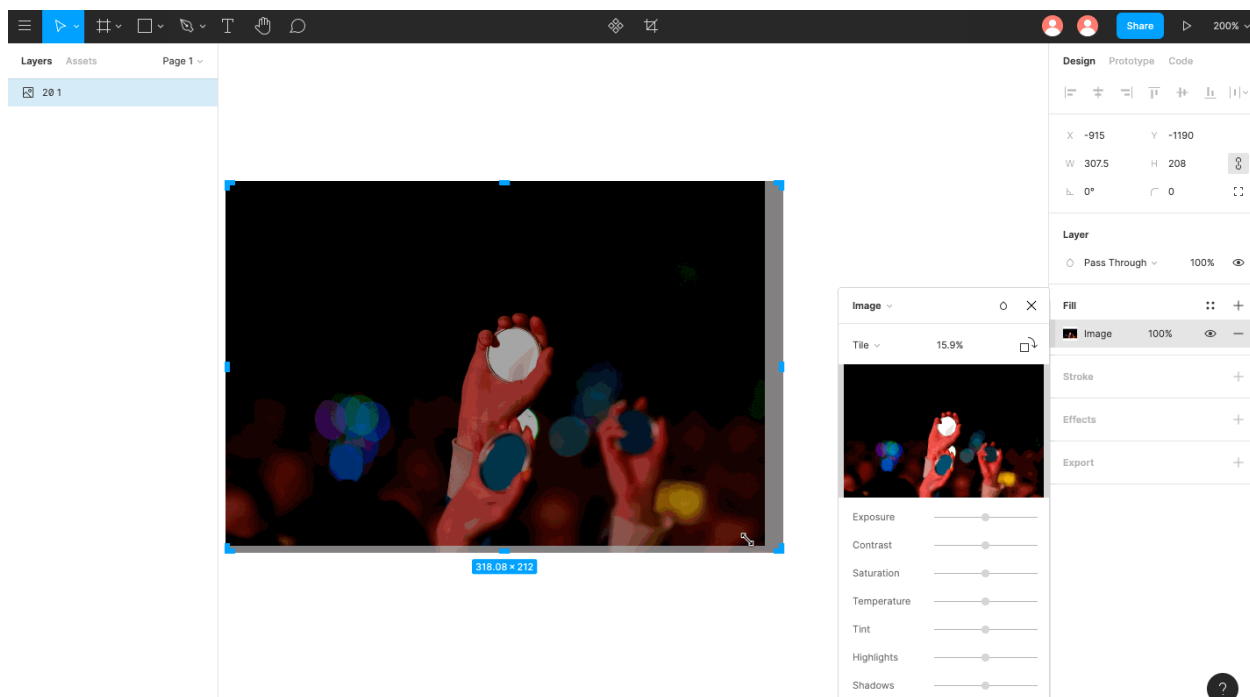


Рисунок 4.90 —Зображення

Регулюйте у Фігмі кольори, яскравість зображення, контрастність, насиченість, яскравість окремих ділянок фото.

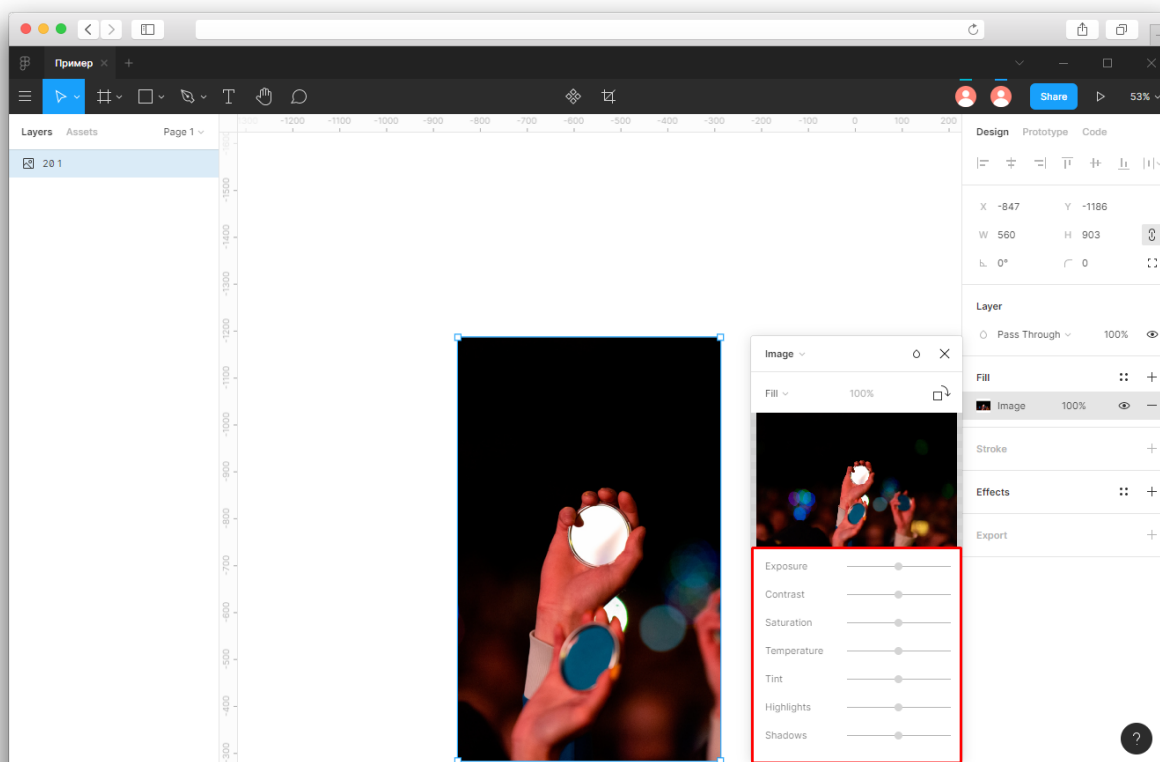


Рисунок 4.91 —Зображення

Щоб додати зображення до градієнта, використовуйте символ «+» на панелі властивостей. Виберіть стиль градієнта: Linear (Лінійний), Radial (Радіальний), Angular (Кутовий) або Diamond (Ромбовидний).

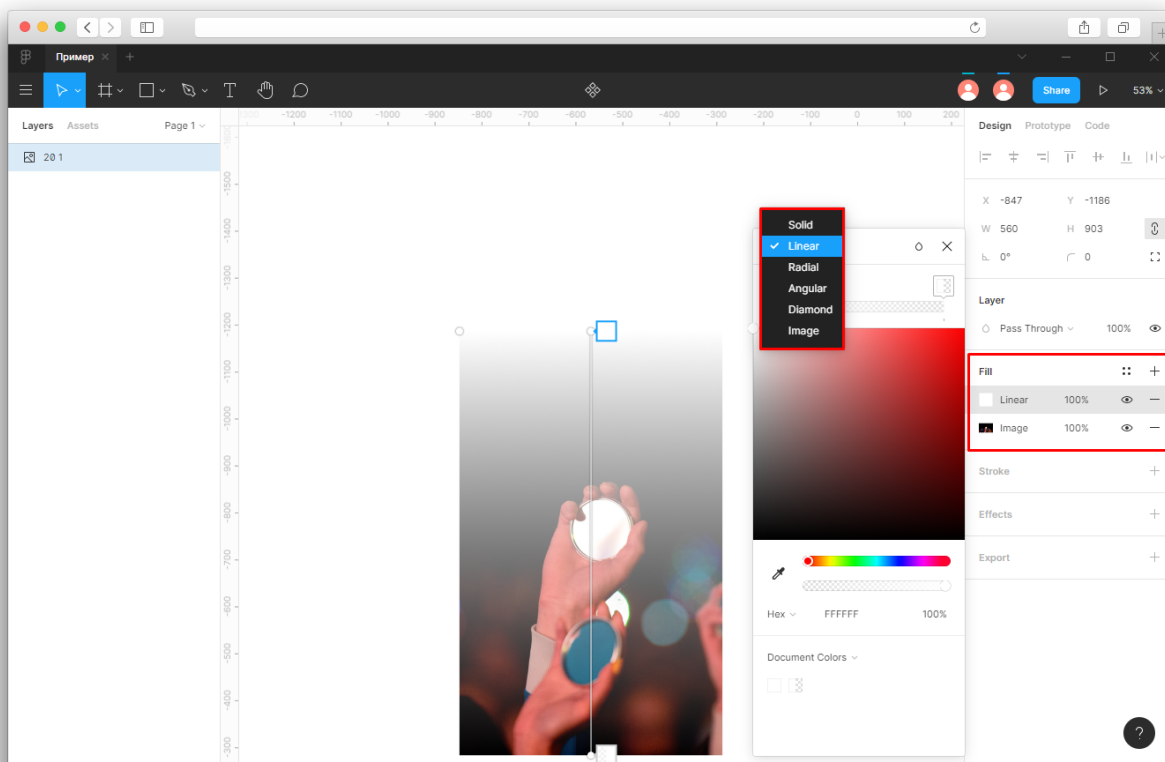


Рисунок 4.92 —Зображення

Використовуйте додаткові параметри кольору та змінюйте положення градієнта, щоб досягти потрібного ефекту зображення.

Ви можете додавати кольори або інші фотографії до зображення з інструментом змішування шарів. Наприклад, щоб потрапити до фірмового стилю сайту або додати фотографії індивідуальності.



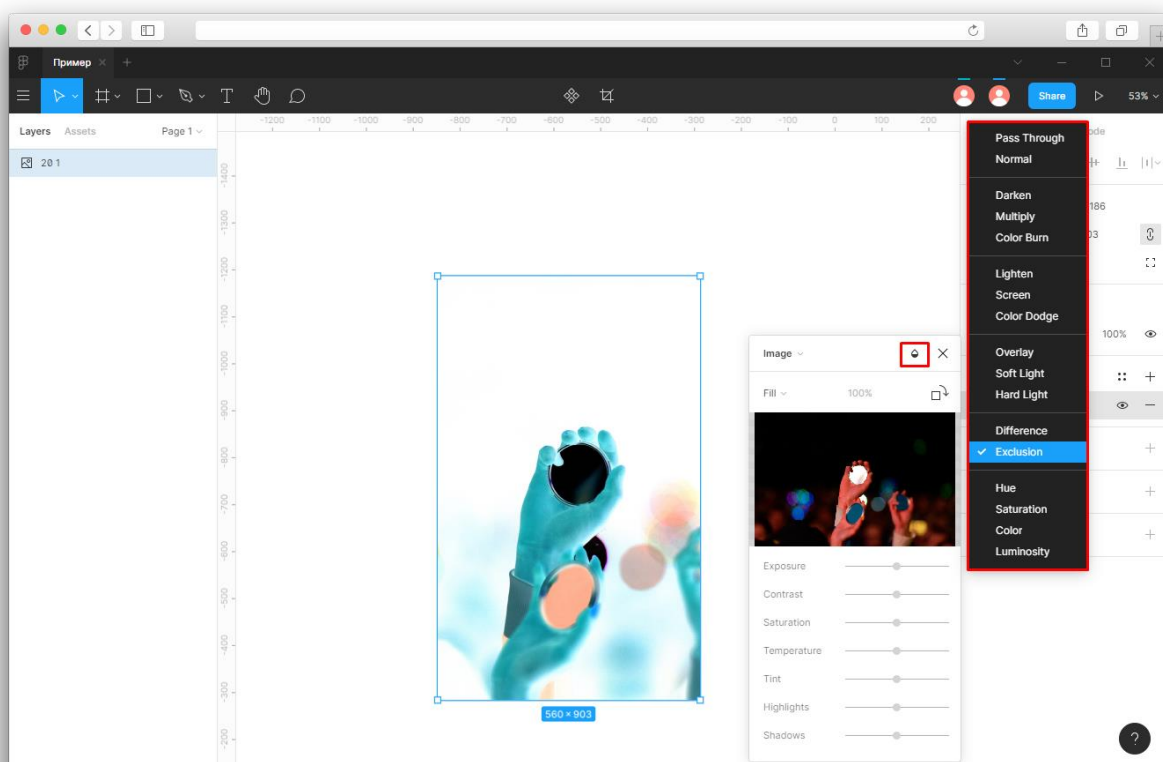


Рисунок 4.93 —Зображення

Ви можете застосовувати векторні об'єкти до фотографій та використовувати їх як маски.

Щоб зробити маску у Фігмі, створіть об'єкт, перенесіть шар нижче фотографії. Виділіть обидва шари та виберіть панель Object → Use as mask (гарячі клавіші для Windows: Ctrl+Alt+M, для Mac OS: Cmd+Opt+M).

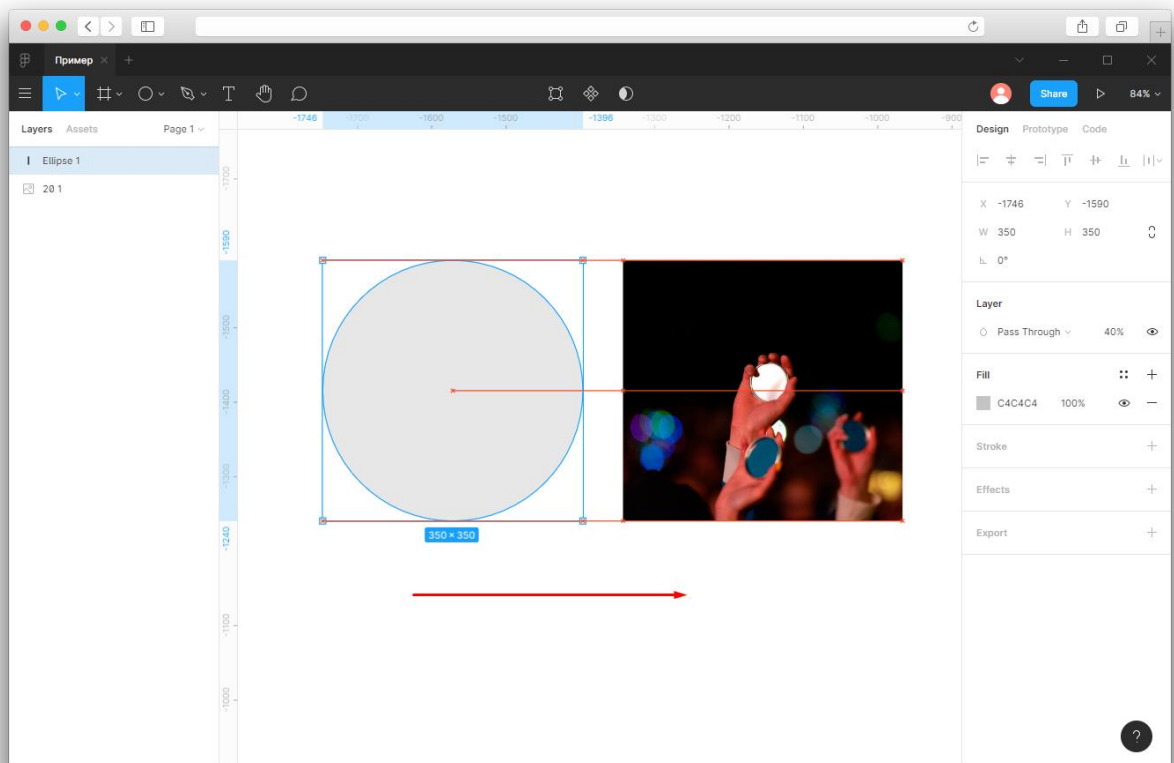


Рисунок 4.94 —Зображення

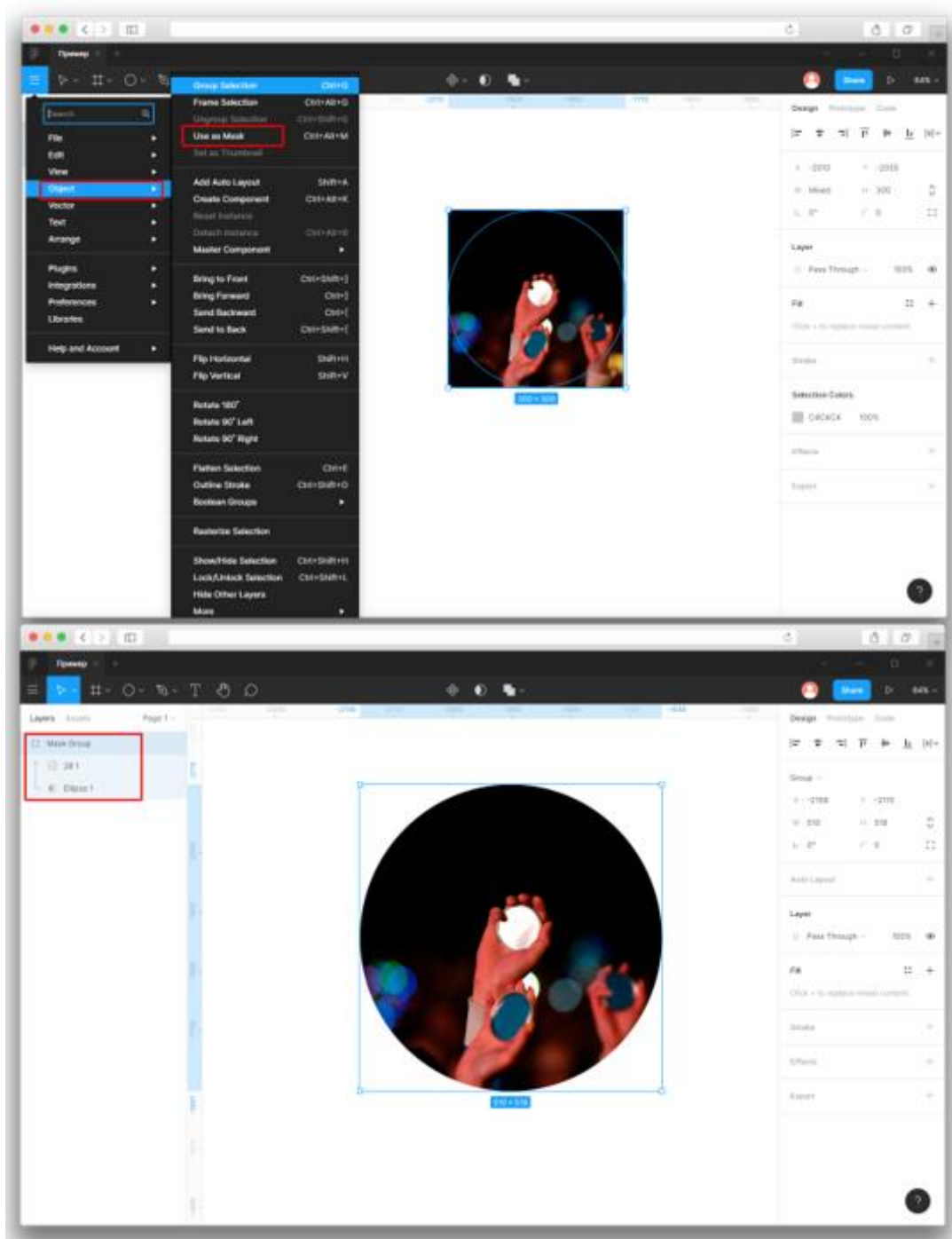


Рисунок 4.95 —Зображення

Ви можете завантажувати у Фігму SVG-файли та працювати з ними. SVG-файли можна завантажити з комп'ютера або переносити з Adobe Illustrator або Sketch за допомогою гарячих клавіш **Cntrl + C** і **Cntrl + V**.

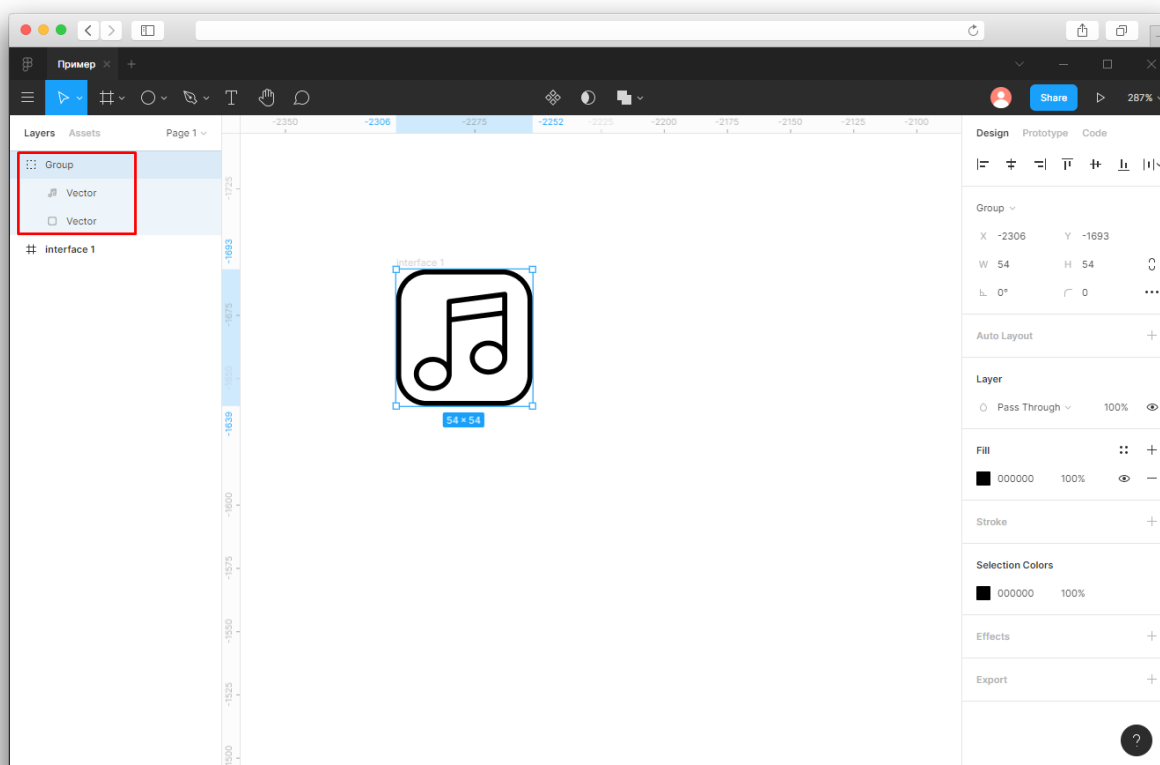


Рисунок 4.96 —Зображення

Зазвичай файли такого типу вставляються як кадр. Перейдіть до шарів, об'єднайте векторні об'єкти в групу клавішами **Ctrl+G** і перетягніть їх вище кадру.

#### 4.3.11 Ефекти та маски

Фігма дозволяє працювати із заливкою фреймів та текстових шарів. Інструменти заливки знаходяться у лівій панелі властивостей. Ви можете вибрати колір, прозорість, вказати конкретні значення кольору.

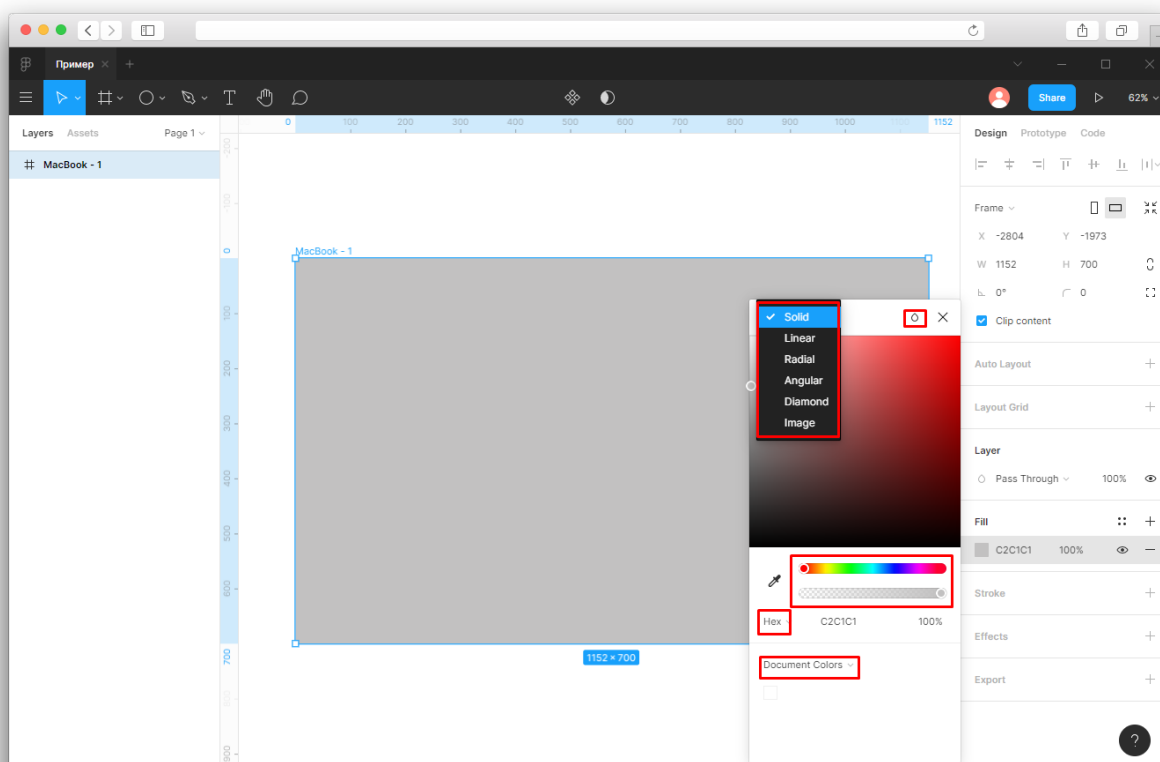


Рисунок 4.97 —Заливка

У Фігмі шість типів заливки: суцільний колір (Solid), лінійний градієнт (Linear), радіальний градієнт (Radial), кутовий градієнт (Angular), радіальний із чотирма променями (Diamond), зображення (Image). За замовчуванням для об'єкта вибрано режим заливки Solid .

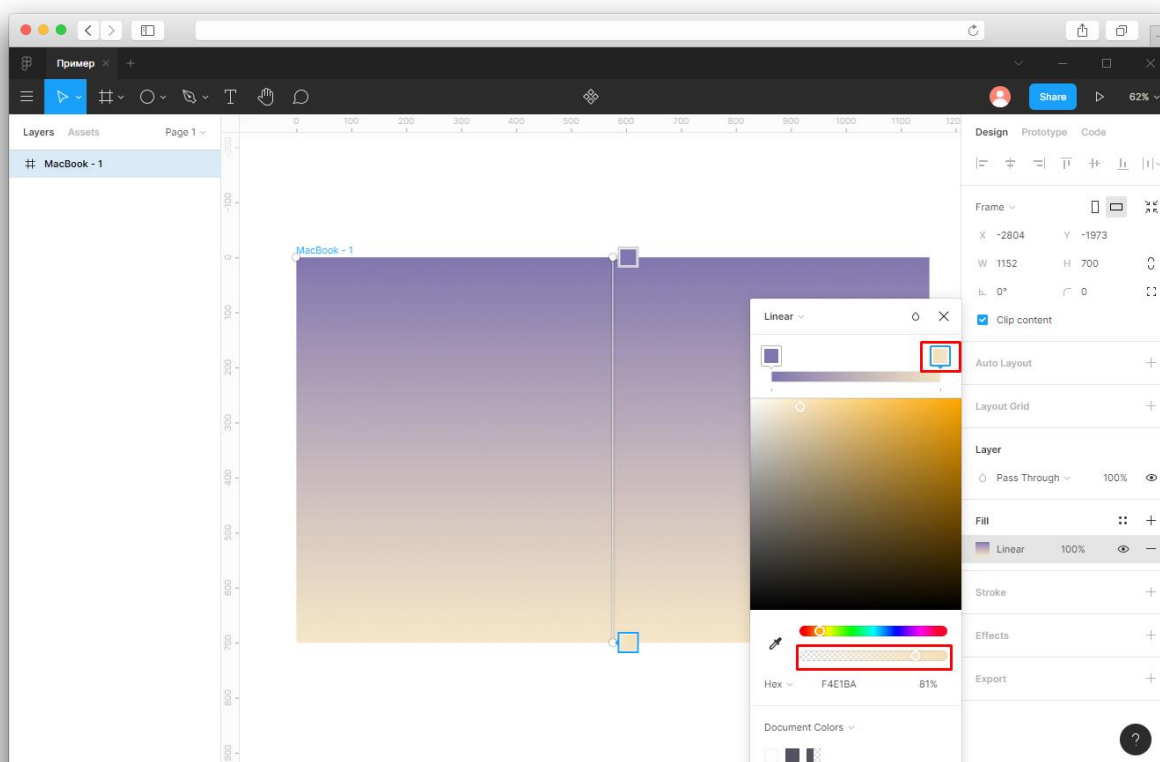


Рисунок 4.98 —Заливка

Перейдіть на Linear - градієнт з віссю, вздовж якої змінюється колір. За замовчуванням він має дві точки, одна з яких прозора. Додайте кольори повзунком або вкажіть значення кольору в полі Нех-коду.

Лінійний градієнт має третю непримітну ручку. Якщо затиснути Alt (Opt) і потягнути за неї, перехід кольорів відмальовується перпендикулярно їй.

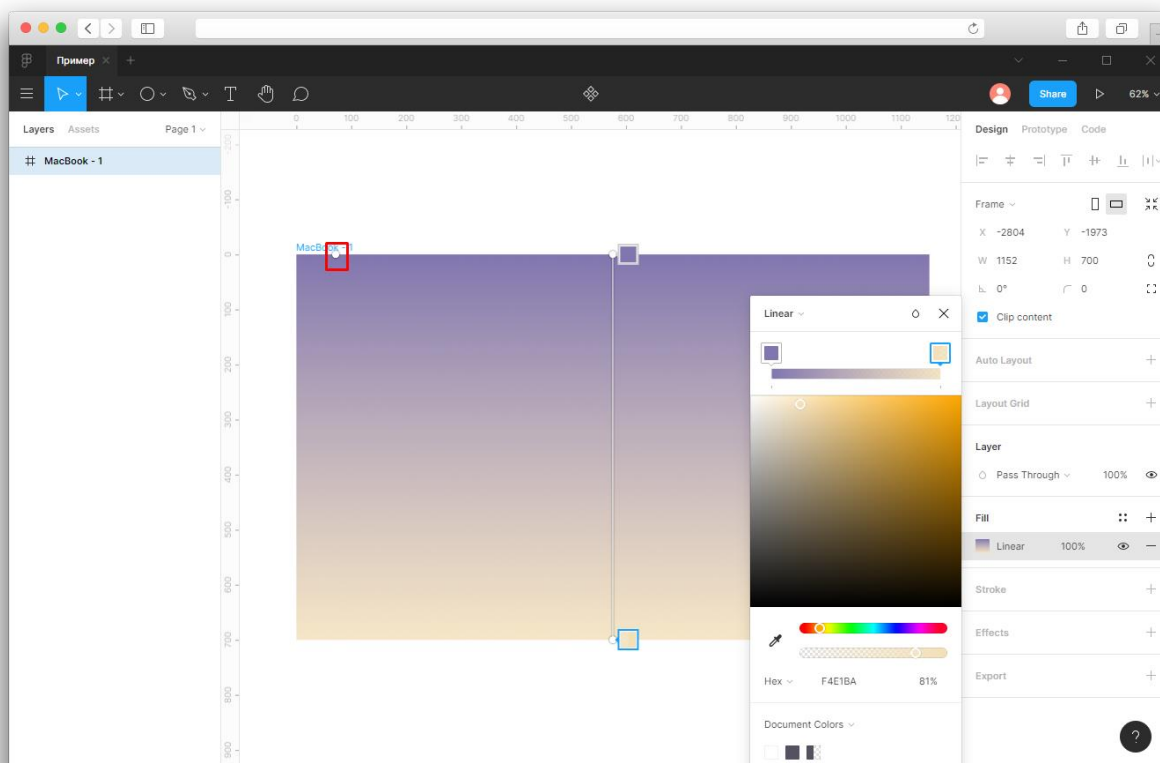


Рисунок 4.99 —Заливка

Ви можете змінювати розташування крайніх точок градієнта. Рухаючи повзунки, можна настроїти кут повороту градієнта і швидкість переходу від одного кольору до іншого.

Ви можете додати кілька кольорів до градієнта, клікнувши за шкалою над палітрою. Щоб видалити колір, натисніть Delete.

Додайте об'єктам заливки з різними градієнтами та поєднуйте фігури одна з одною. Ви можете використовувати Layer - режим накладання, накладати їх один на одного для цікавого рішення.

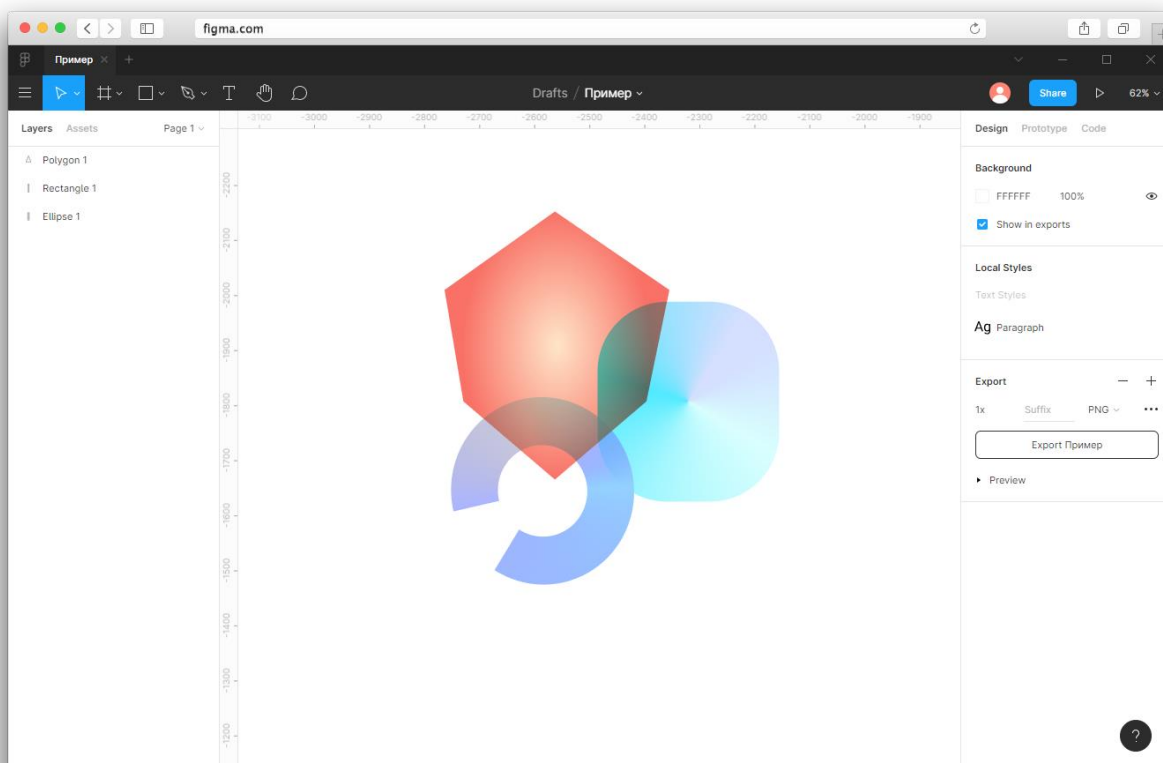


Рисунок 4.100 — Режим накладання

Система ефектів в Figma надає потужні можливості для створення візуально привабливих та сучасних інтерфейсів без необхідності використання зовнішніх графічних редакторів. Ефекти можуть застосовуватися до будь-яких об'єктів та комбінуватися для досягнення складних візуальних результатів.

Тіні (Drop Shadow) є одним з найбільш використовуваних ефектів в UI дизайні, оскільки створюють відчуття глибини та ієрархії елементів. Figma дозволяє налаштовувати всі параметри тіні - зміщення по осях X та Y, радіус розмиття, розповсюдження та колір з прозорістю.

Внутрішні тіні (Inner Shadow) створюють ефект заглиблення та часто використовуються для полів вводу, кнопок в натиснутому стані та інших елементів, які повинні візуально "провалюватися" в інтерфейс.

Розмиття (Blur) включає кілька типів - розмиття шарів для створення ефекту глибини поля, розмиття руху для динамічних елементів та розмиття фону, яке особливо корисне для створення ефекту скленого морфізму (glassmorphism).



Зовнішнє свічення (Outer Glow) та внутрішнє свічення (Inner Glow) додають елементам м'яке сяйво, що може використовуватися для створення неонових ефектів, виділення активних елементів або додавання магічності інтерфейсу.

Маски в Figma працюють за принципом приховування частин об'єктів за допомогою інших форм. Будь-який об'єкт може стати маскою для групи елементів, що розташовані під ним в ієрархії шарів. Це дозволяє створювати складні композиції з обрізаними зображеннями, текстом всередині форм та іншими творчими рішеннями.



Рисунок 4.101 — Ефекти

#### 4.3.12 Текст

Типографічна система Figma розроблена з урахуванням специфіки цифрового дизайну та надає всі необхідні інструменти для створення читабельних та естетично привабливих текстових композицій. Робота з текстом включає не тільки базові налаштування шрифтів, але й складні системи стилізації та управління типографічною ієрархією.

Вибір шрифтів в Figma включає доступ до Google Fonts - великої бібліотеки безкоштовних веб-шрифтів, а також можливість використання локальних шрифтів, встановлених на комп'ютері користувача. Це забезпечує гнучкість у виборі типографічного оформлення та гарантує, що дизайн буде коректно відображатися при передачі між різними учасниками команди.

Налаштування тексту включають традиційні параметри - розмір шрифту, висоту рядка, міжсимвольну відстань (tracking) та колір. Висота рядка може налаштовуватися як в абсолютних значеннях (пікселі), так і в відносних (відсоток від розміру шрифту), що важливо для створення послідовної типографічної системи.

Текстові стилі (Text Styles) є одним з найпотужніших інструментів для забезпечення консистентності типографіки в рамках проекту. Створивши набір стилів для заголовків різних рівнів, основного тексту, підписів та інших типографічних елементів, дизайнер може швидко застосовувати їх до нових текстових блоків та централізовано оновлювати при необхідності.

Автоматичні макети тексту (Auto Layout) дозволяють створювати адаптивні текстові блоки, які автоматично змінюють свій розмір залежно від кількості контенту. Це особливо корисно при створенні кнопок, меток та інших інтерфейсних елементів, де кількість тексту може варіюватися.

Робота з багатомовністю включає підтримку різних систем письма, налаштування напрямку тексту (зліва направо або справа наліво) та оптимізацію розташування для мов з різною довжиною слів та фраз.

За замовчуванням вам доступна бібліотека шрифтів Google Fonts, якщо ви хочете завантажувати свої шрифти до Фігми — завантажте десктопну версію Фігми або встановіть Font Installers ( Windows / Mac OS ), щоб завантажувати їх у браузер.

Інструменти для роботи з текстовим шаром стандартні: накреслення, розмір, вирівнювання тексту, висота рядка, відступ між параграфами та червоний рядок.

Також є прев'ю шрифтів та можливість перемикати категорії, наприклад, відобразити всі шрифти, лише популярні або встановлені на вашому пристрої.

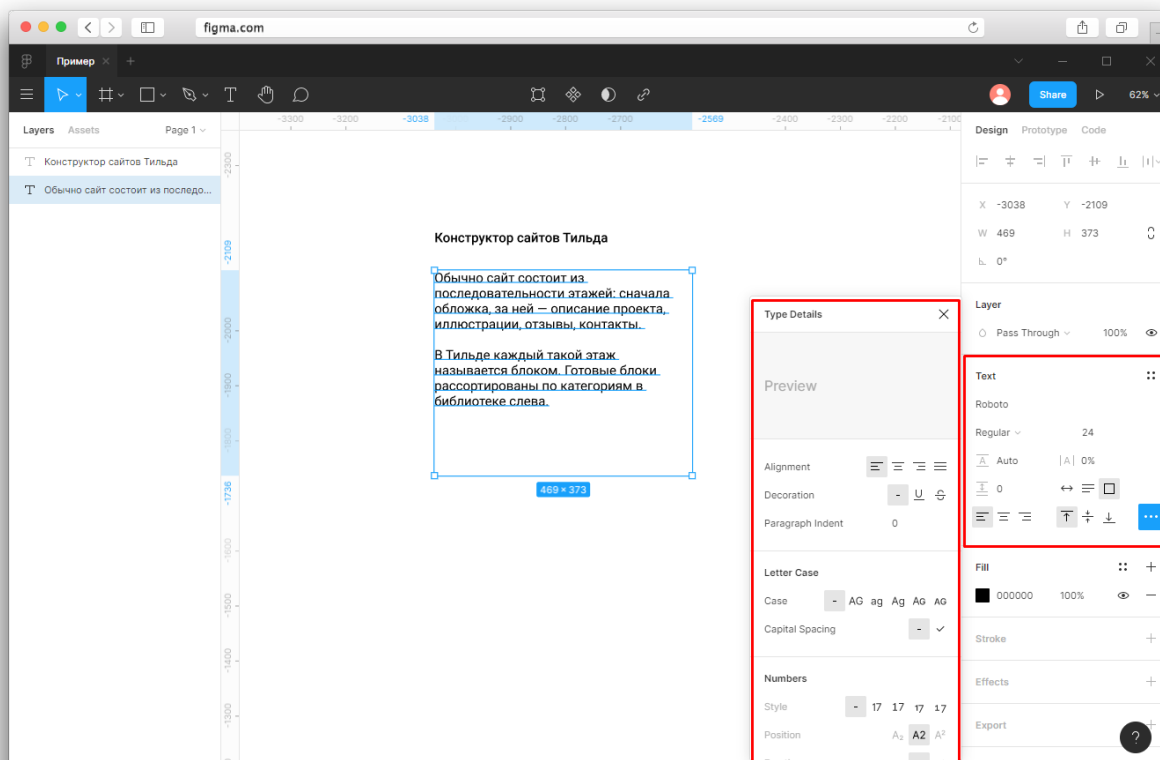


Рисунок 4.102 — Текст

У меню Advanced Type є три типи зміни розміру текстового блоку: 1) Width. Завжди створюйте окремі текстові блоки для заголовка та основного тексту. Так їх зручніше редагувати.

У Фігмі можна перекласти шрифт у криві та створити незвичайний об'єкт вашого дизайну. Виберіть текстовий шар, Flatten і двічі клацніть на текст.

Змінений шрифт можна зберегти як файл SVG і вставити в дизайн.

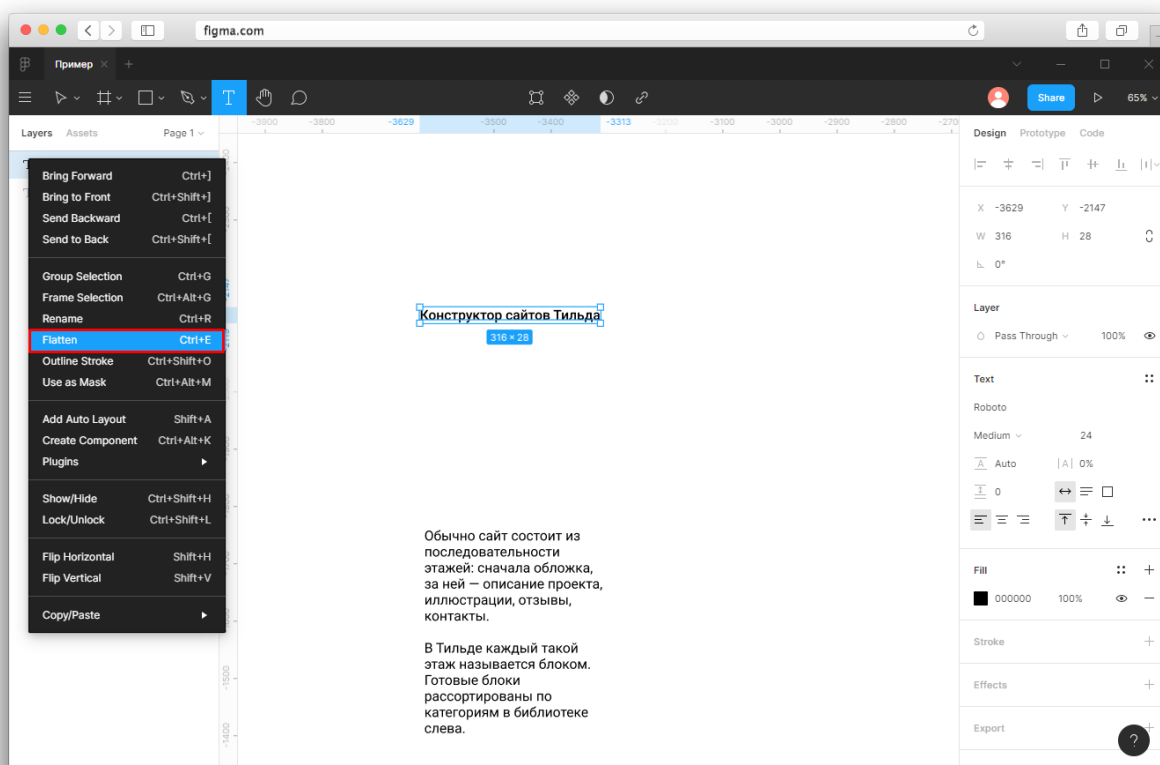


Рисунок 4.103 — Текст

Компоненти у Фігмі допомагають застосовувати зміни до групи елементів. Це заощаджує час дизайнера при змінах макета.

Допустимо, ви зробили макет на 50 сторінок, а замовник захотів змінити в ньому колір кнопок. З компонентами вам достатньо буде кілька разів клікнути мишкою і застосувати новий колір до всіх кнопок одночасно.

Щоб перетворити об'єкт або групу об'єктів на компонент, виділіть їх та натисніть Create Component (Ctrl+Alt+K).

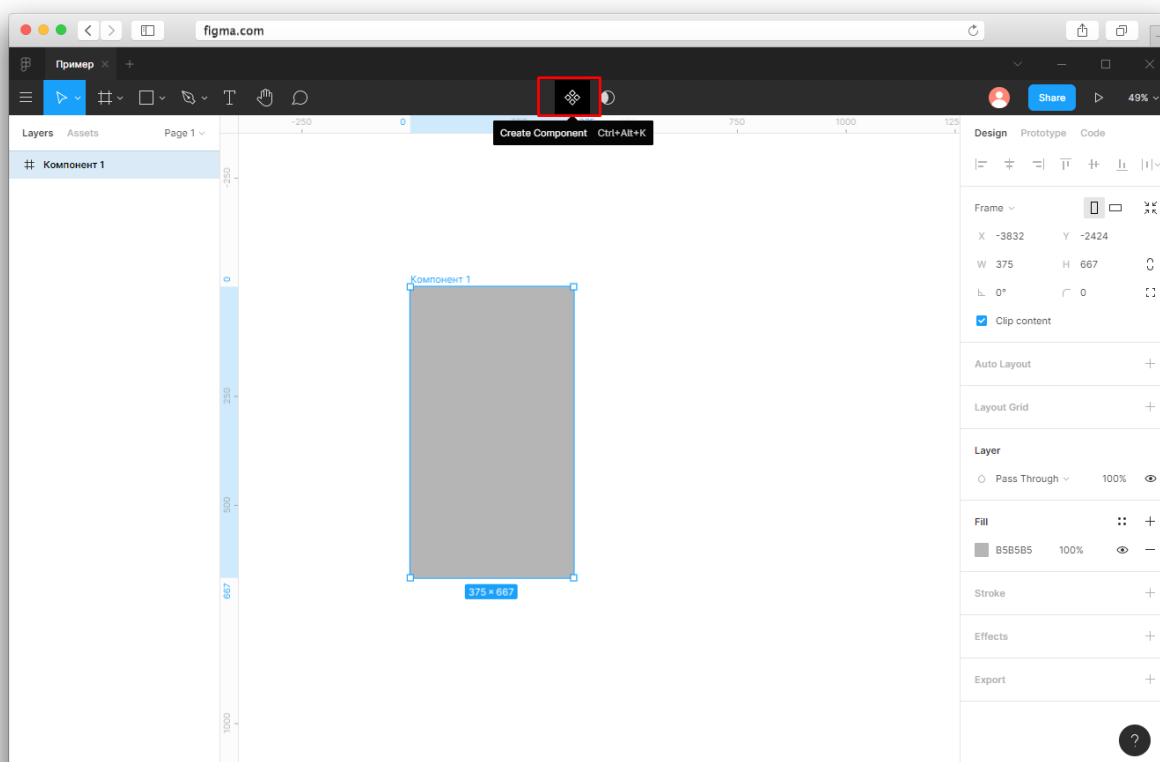


Рисунок 4.104 — Текст

Побачити компоненти можна за кольором шару Фігма позначає їх фіолетовим. Крім кольору, батьківський компонент позначається іконкою із 4 ромбами, а дочірні компоненти – іконкою з одним ромбом.

Тепер ваш кадр - батьківський компонент. Створіть копії, щоб отримати дочірні компоненти. Усі зміни батьківського компонента відбиватимуться на дочірніх компонентах.

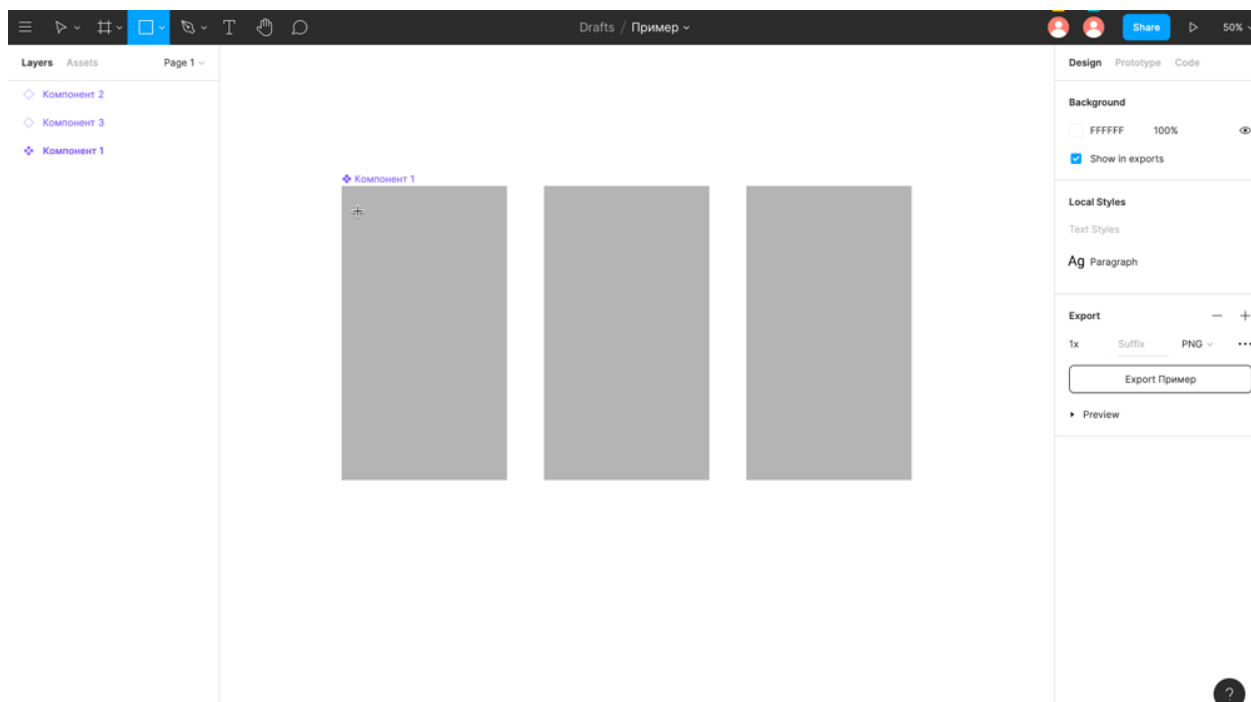


Рисунок 4.105 — Текст

Ви можете змінювати налаштування дочірнього компонента, роблячи його відносно самостійним. Наприклад, можна вручну задати компоненту розмір, колір, обведення, відмінні від батьківського компонента.

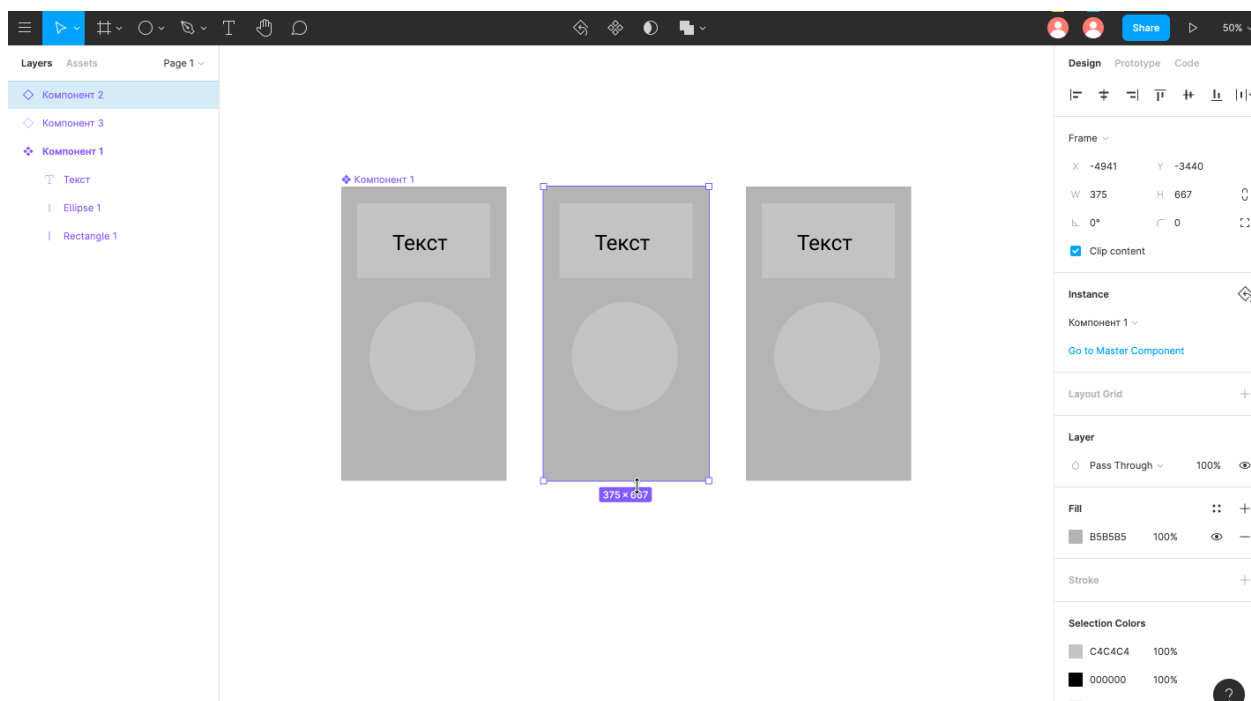


Рисунок 4.106 — Текст

У цьому зв'язок між компонентами не загубиться. Внесіть новий об'єкт у батьківський компонент, і він відобразиться у дочірніх компонентах, навіть змінених раніше.

Щоб відновити порушений зв'язок між батьківським та дочірнім компонентом, виділіть дочірній компонент, натисніть іконку повернення дії та скиньте параметри, які змінювали в ньому вручну.

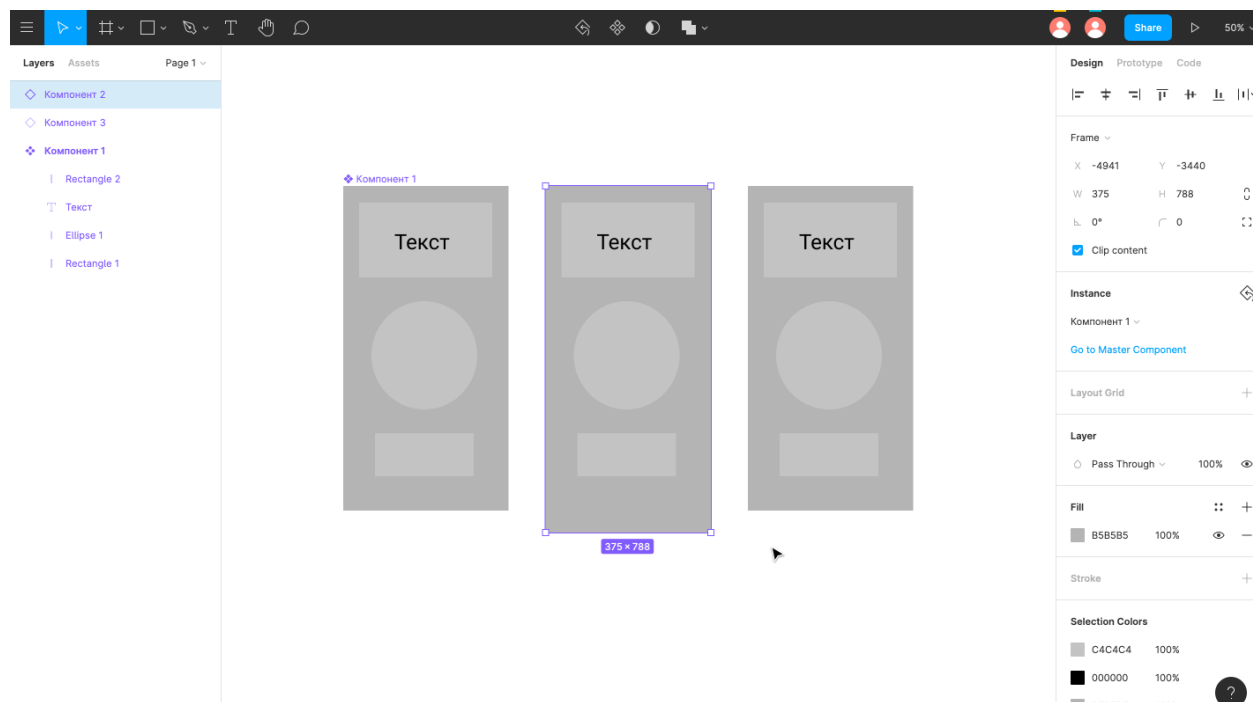


Рисунок 4.107 — Текст

Коли ви працюєте з компонентами, зручно тримати їх в одному місці, щоб вони не губилися на полотні. Створіть новий кадр, назвіть його Components і вкладіть у нього батьківські компоненти.

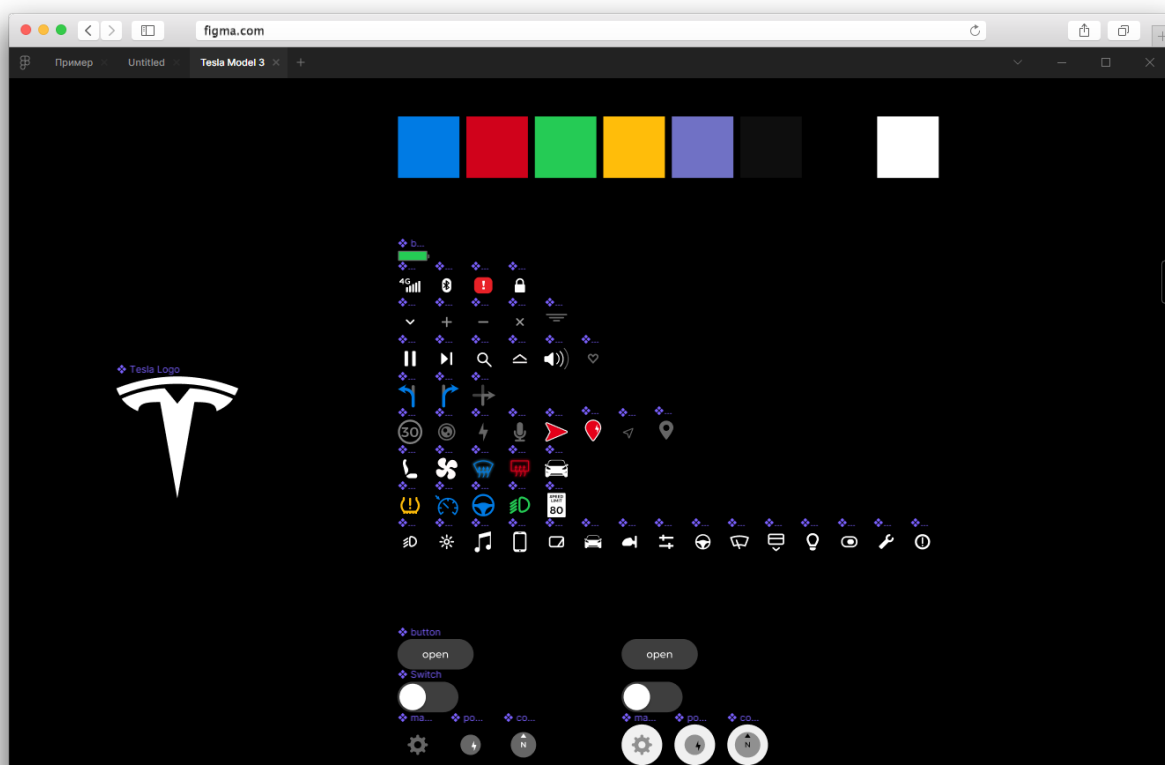


Рисунок 4.108 — Components

Поряд із компонентами у Фігмі можна створювати стилі. Це колекції тексту, кольорів та ефектів, які ви зможете застосовувати до елементів дизайну. Коли ви змінюєте властивості стилю, вони змінюються у всіх об'єктах, які його використовують.

Збережіть шаблон з компонентів та стилів, щоб створити власну бібліотеку для майбутніх проектів.

Створіть стилі, які застосовуватимете до всіх можливих текстів вашого дизайну: заголовків, параграфів, підписів, тегів.

Використовуйте панель властивостей праворуч.



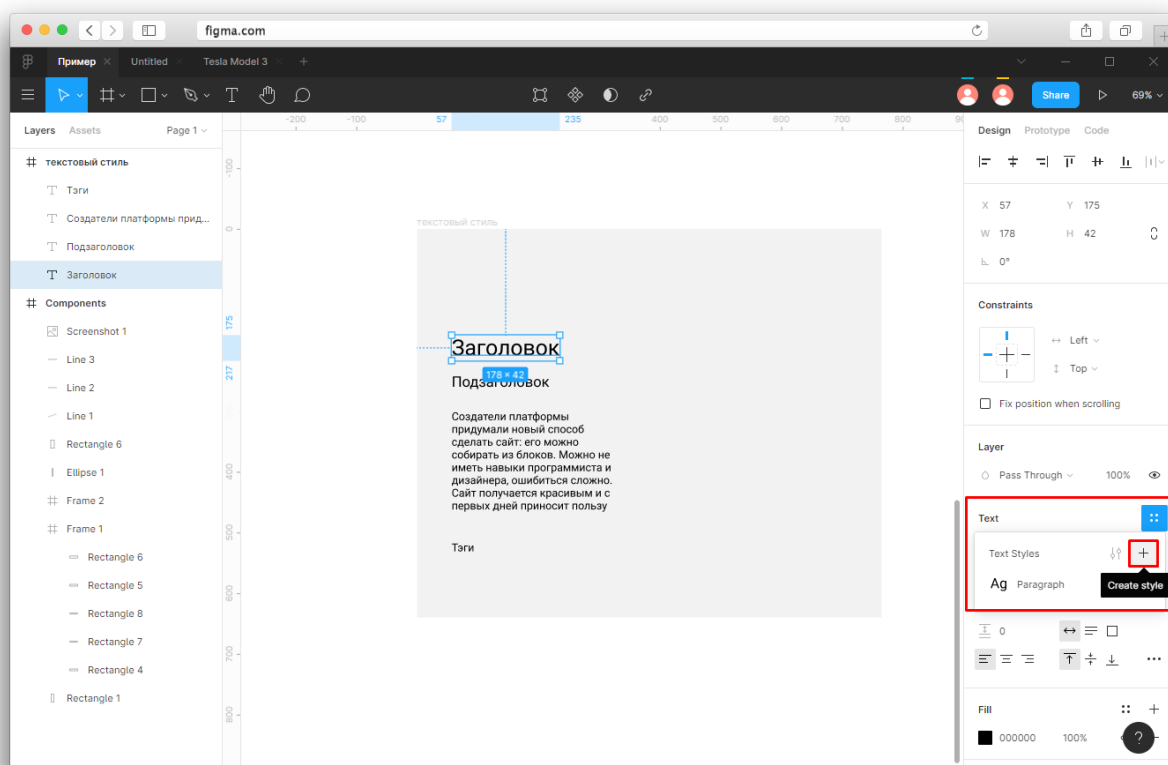


Рисунок 4.109 — Стилї

Збережіть усі варіанти текстових стилів в окремому кадрі та застосовуйте під час роботи з текстами.

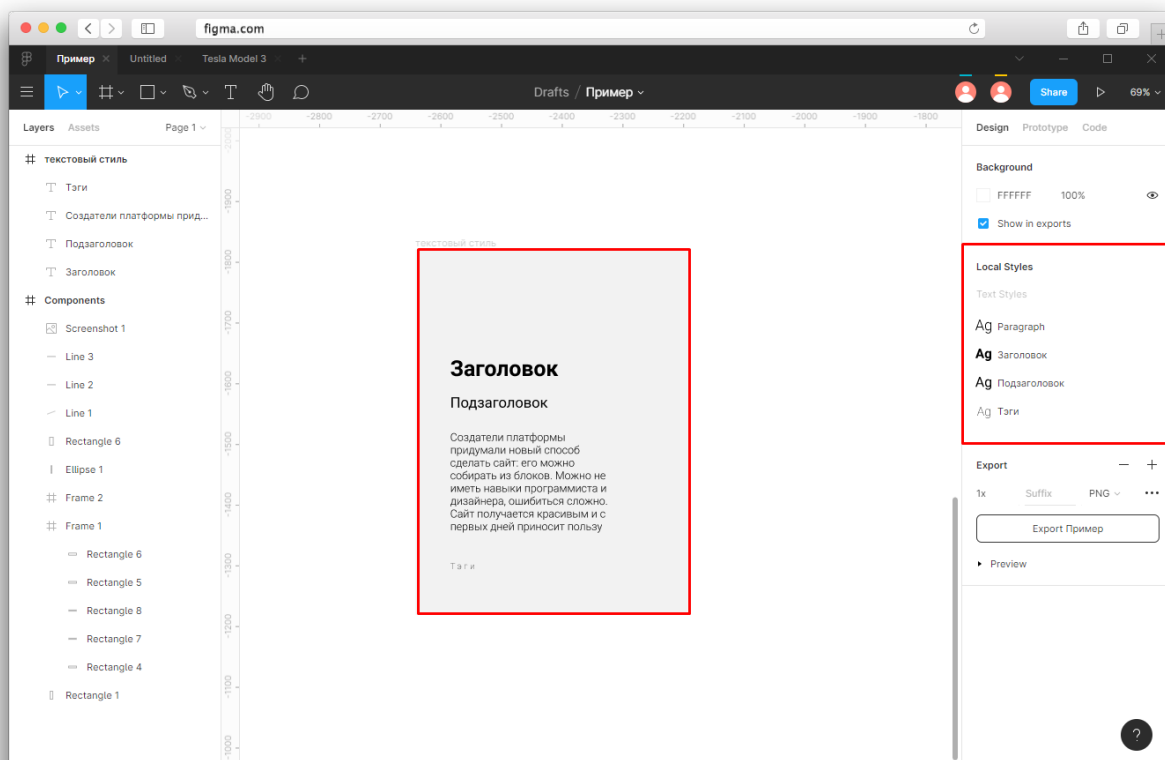


Рисунок 4.110 — Стилi

За аналогією до текстових стилів, створіть заливки з основними кольорами вашого дизайну. Підпишіть кожен колір та вкажіть його код.

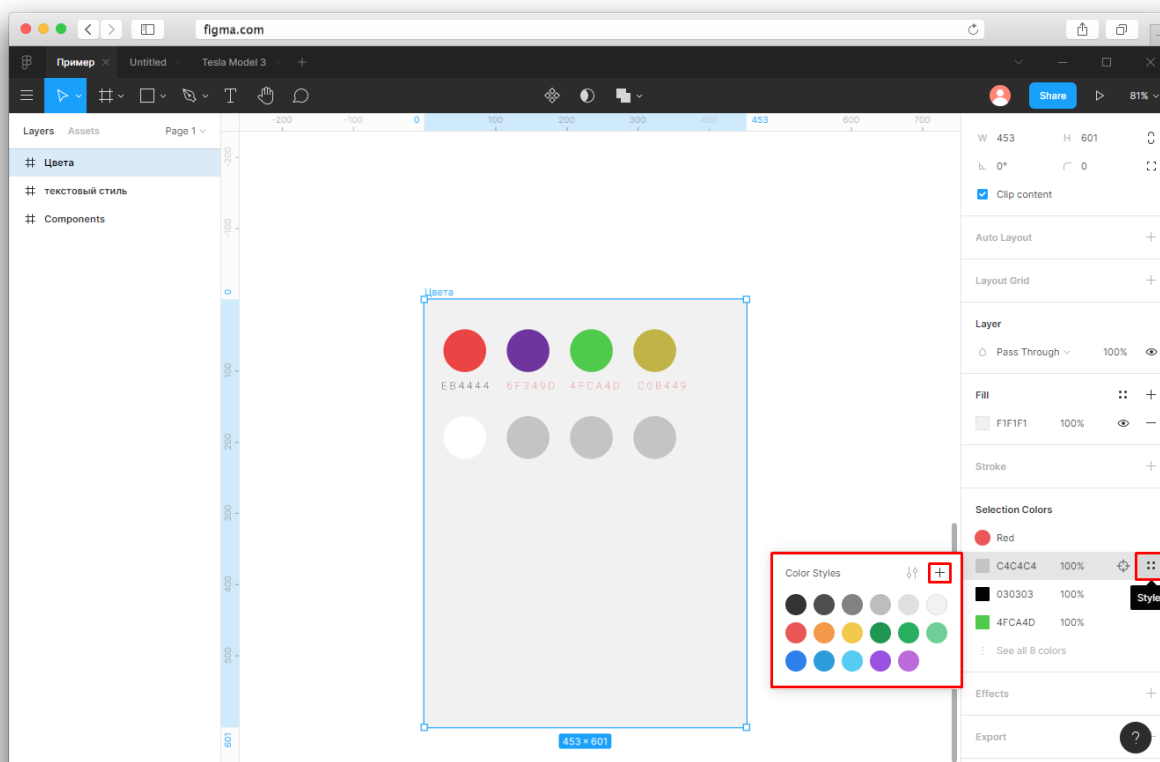


Рисунок 4.111 — Колір

Створіть стилі розмиття шару, фону та тіні.

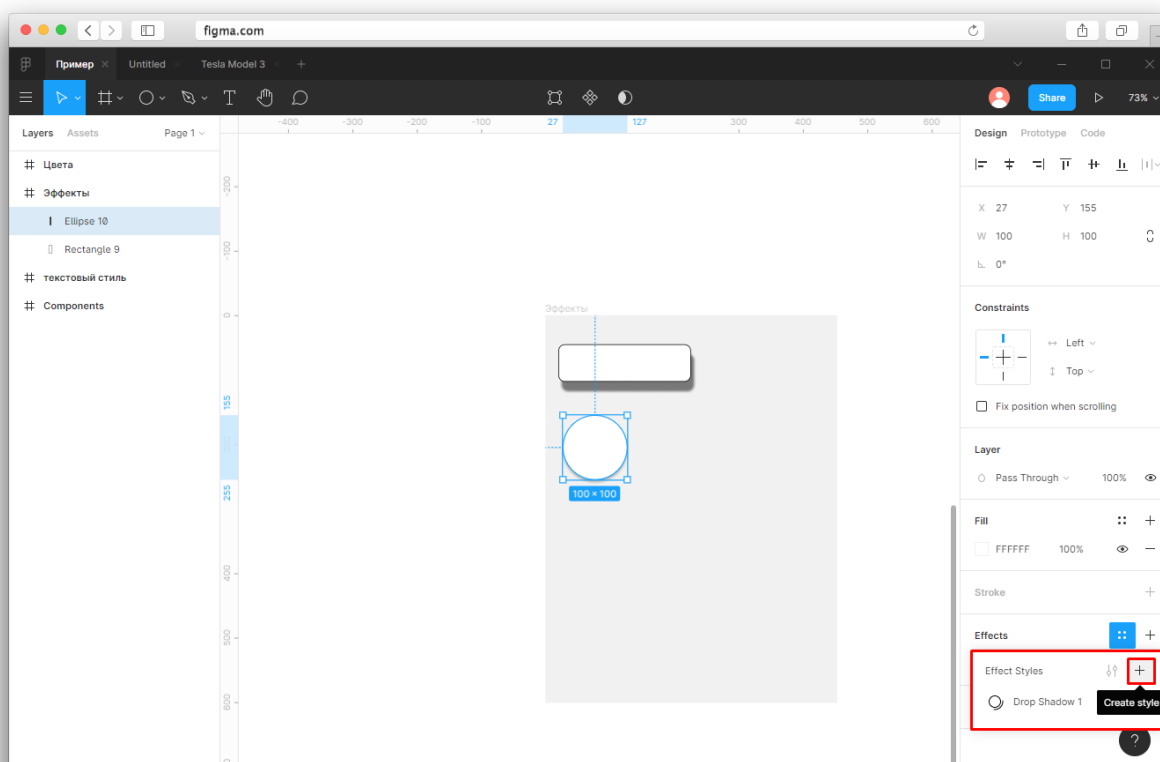


Рисунок 4.112 — Эффекты

Для роботи з компонентами Фігма розробила функцію Auto Layout. Вона допомагає уникнути проблем із розміром контенту в нових компонентах. Допустимо, ви створили дочірній компонент кнопки і вписали новий підпис. Розміри кнопки при цьому не змінилися, і ваш дизайн виглядає непрофесійно.

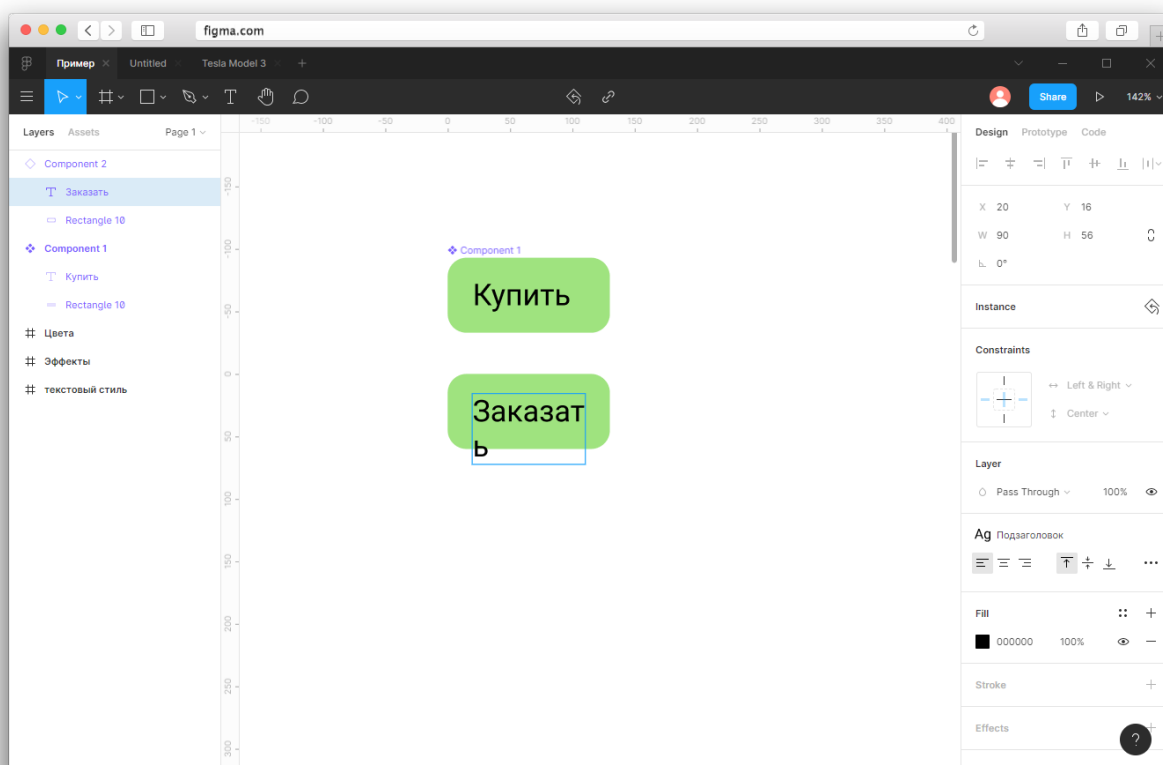


Рисунок 4.113 — Auto Layout

Приберіть зв'язок дочірнього елемента з батьківським елементом командою Detach Instance і застосуйте Auto Layout.

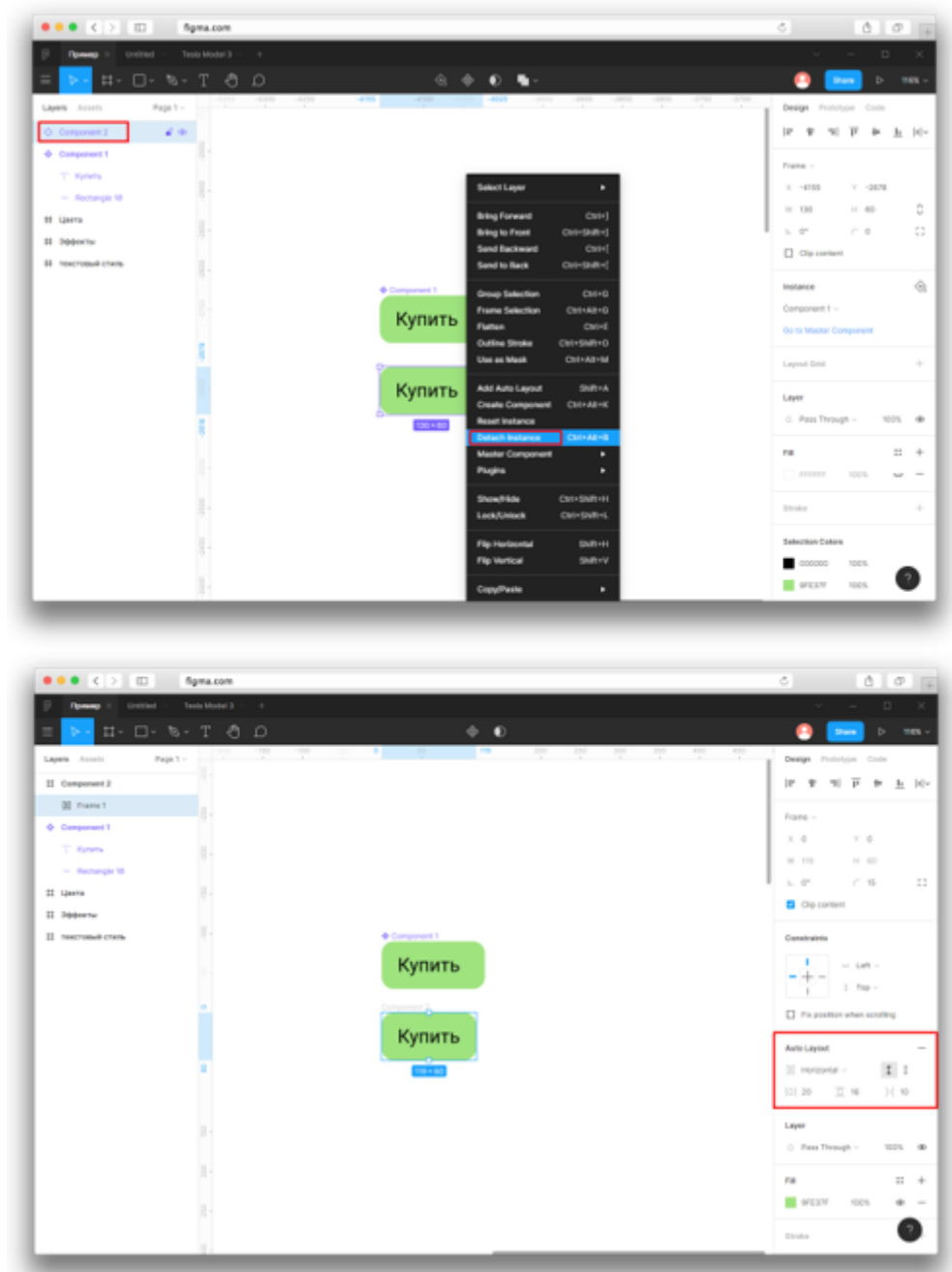


Рисунок 4.114 — Auto Layout

Тепер розміри кнопки динамічно змінюються під її вміст.

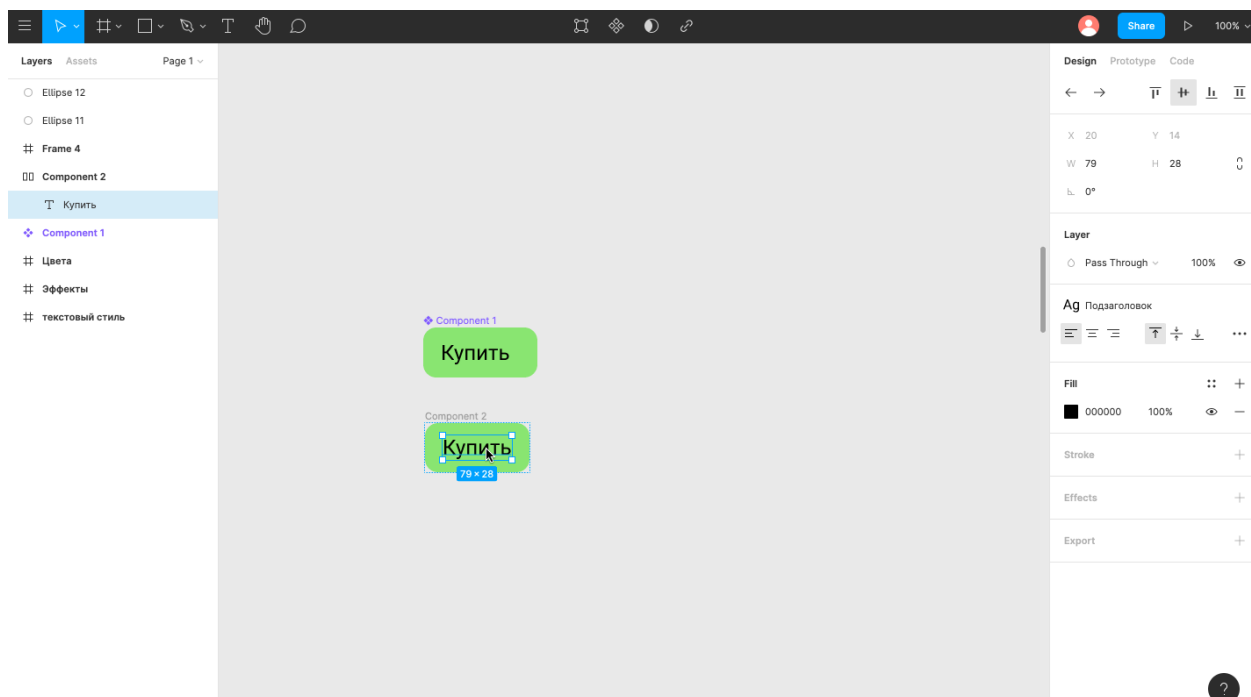


Рисунок 4.115 — Auto Layout

У Auto Layout можна налаштувати, як змінюватиметься його розмір залежно від контенту.

Є три параметри Resizing, які окремо задаються для вертикальної та горизонтальної осі:

- Fixed – фіксована ширина, задана вручну. Auto Layout не реагуватиме на зміну розмірів контенту;
- Hug — розмір за контентом, тобто Auto Layout збільшуватиметься чи зменшуватиметься залежно від вмісту;
- Fill – налаштування доступне для об'єктів усередині Auto Layout. Робить так, щоб елемент заповнив батьківський контейнер за висотою або шириною.

Для зручної роботи з кадрами та адаптивами через Auto Layout також доступні функції:

- Wrap – рівномірно переносить всі елементи при зміні розміру кадру. Наприклад, якщо зменшити область, елементи автоматично перенесуться на новий рядок. Функція може бути корисною при розробці макетів для різних роздільних здатностей екранів.

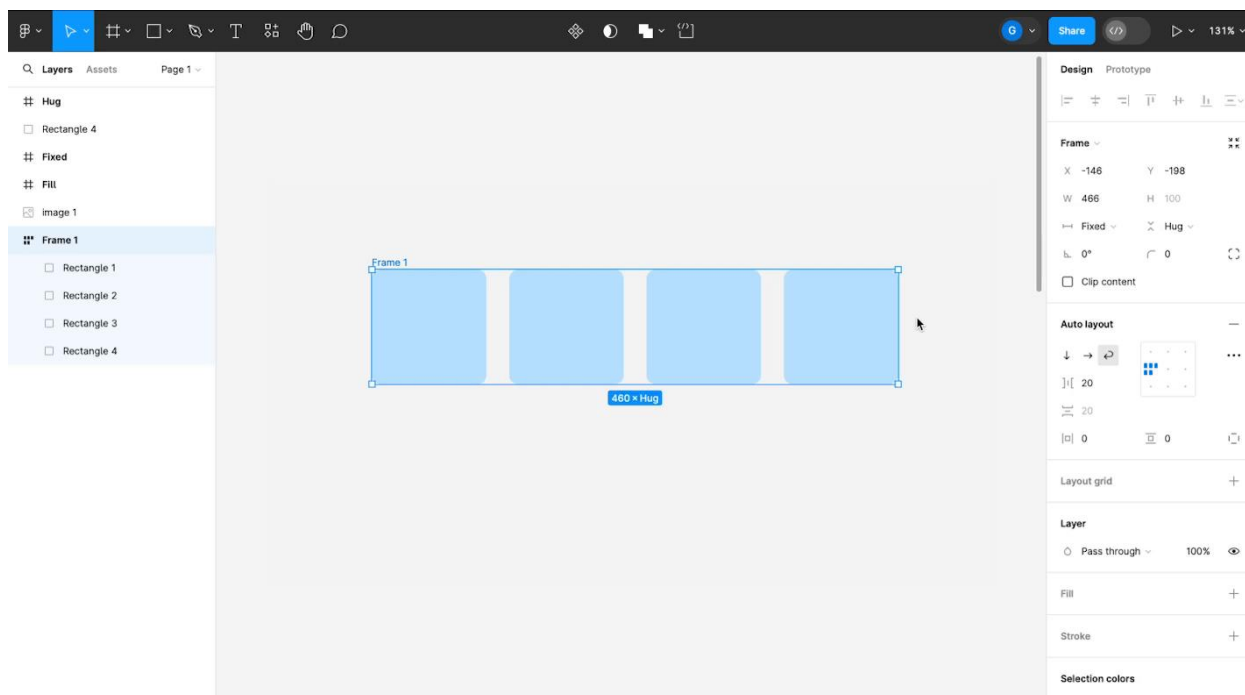


Рисунок 4.116 — Auto Layout

- Min/max — задає максимальну та мінімальну висоту та ширину для кадру та його дочірніх елементів. Тобто, якщо встановити елемент мінімальну ширину, при зменшенні розміру кадру цей елемент буде зменшуватися до встановленого значення, але не менше. Аналогічно працює з максимальним значенням лише при збільшенні кадру. Ця функція стане в нагоді, коли ви захочете подивитися, як дизайн виглядає на різних пристроях. Задані обмеження запобігають спотворенню елементів і вони не стануть надто великими чи дрібними.

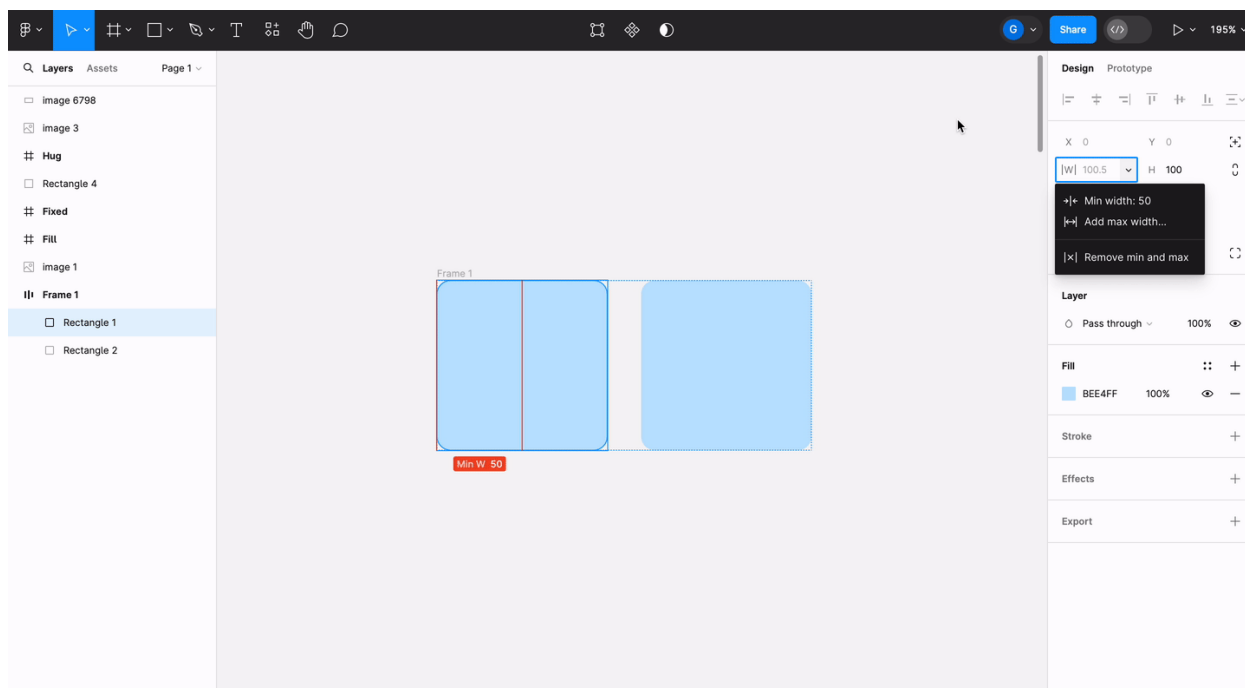


Рисунок 4.117 — Auto Layout

### 4.3.13 Додаткові можливості

Поза основним функціоналом дизайн-редактора, Figma пропонує широкий спектр додаткових можливостей, які значно розширюють сферу її застосування та підвищують ефективність робочого процесу. Ці функції часто стають визначальними факторами при виборі платформи для професійної діяльності.

Система плагінів є однією з найпотужніших особливостей Figma, яка дозволяє розширювати функціональність платформи практично без обмежень. Плагіни можуть автоматизувати рутинні завдання, інтегруватися з зовнішніми сервісами, генерувати контент, перевіряти доступність дизайну та виконувати безліч інших корисних функцій.

Популярні плагіни включають генератори випадкових даних для заповнення макетів реалістичним контентом, інструменти для оптимізації зображень, конвертери одиниць вимірювання, валідатори дизайн-систем та багато інших. Можливість створення власних плагінів за допомогою API



дозволяє адаптувати Figma під специфічні потреби конкретних проектів або організацій.

Інтерактивні прототипи в Figma створюються за допомогою системи з'єднань між артбордами та налаштування переходів. Це дозволяє моделювати користувацькі сценарії, тестувати навігацію та демонструвати функціональність додатку без написання коду.

Анімації та мікроінтеракції можуть бути створені за допомогою інструментів Smart Animate, які автоматично генерують плавні переходи між станами інтерфейсу. Це особливо корисно для демонстрації того, як елементи інтерфейсу повинні поводитися при взаємодії з користувачем.

Інтеграція з іншими інструментами розробки включає можливості експорту CSS-коду, інтеграцію з Slack для сповіщень про зміни в проектах, з'єднання з Jira для створення завдань та багато інших корисних інтеграцій, які роблять Figma частиною більшої екосистеми інструментів розробки.

Система бранчіну дозволяє створювати гілки проекту для експериментів та А/Б тестування дизайнерських рішень без ризику пошкодження основної версії. Це особливо важливо для великих проектів з множинними учасниками, де різні дизайнери можуть працювати над альтернативними рішеннями паралельно.

Автоматичне генерування специфікацій для розробників є революційною функцією, яка автоматично створює детальну технічну документацію на основі дизайн-макетів. Розробники можуть отримувати точні значення відступів, розмірів, кольорів, шрифтів та інших параметрів безпосередньо з інтерфейсу Figma, що значно зменшує кількість запитань та непорозумінь між дизайнерами та програмістами.

Інструменти для тестування доступності включають перевірку контрастності кольорів, аналіз розмірів інтерактивних елементів та інші функції, які допомагають створювати інклюзивні дизайни, доступні для користувачів з різними потребами та обмеженнями.

Система FigJam - інтегрований інструмент для створення діаграм, мозкового штурму та планування проектів, який дозволяє командам проводити дизайн-спринти, створювати карти користувацьких подорожей та структурувати ідеї ще до початку безпосередньо дизайнерської роботи.



Рисунок 4.118 — Розширені можливості Figma

Поєднання гарячих клавіш вказано в Keyboard Shortcuts. Викличте його з меню Help and Account, коли працюєте з файлом, або натисніть "Cmd + Shift +?" в Mac OS і "Ctrl + Shift +?" у Windows.

## Висновки

У цьому розділі було розглянуто ключові принципи графічного дизайну та UX-проектування, які формують основу сучасного веб-дизайну. Було підкреслено, що ефективний дизайн поєднує естетику з функціональністю, забезпечуючи не лише привабливий зовнішній вигляд сайту, але й зручність користування.

Основні положення, які варто виділити:

1) Генерація ідей та концепцій — процес створення дизайну починається з аналізу потреб аудиторії, використання мудбордів, ескізів та дослідження конкурентів. Це дозволяє сформувати унікальний стиль та цілісну концепцію майбутнього сайту.

2) Композиція та візуальні принципи — дизайн будується на балансі, контрасті, акцентах, ритмі та ієрархії, що забезпечують гармонійність і логічність сприйняття контенту.

3) Макетування та структура сторінок — головна сторінка виконує роль візитівки ресурсу, а її макет має бути простим, зрозумілим і водночас інформативним. Використання модульних сіток спрощує побудову адаптивних інтерфейсів.

4) Адаптивність і мобільний дизайн — у сучасних умовах обов'язковим є підхід mobile-first та реалізація responsive або adaptive web design для забезпечення коректного відображення сайту на різних пристроях.

5) Елементи взаємодії — кнопки, меню, візуальні напрямки й заклики до дії повинні бути помітними, інтуїтивними та сприяти досягненню цілей користувача і власника ресурсу.

6) Типографіка та колірні рішення — грамотне використання шрифтів і кольорових схем формує візуальну ідентичність сайту, підвищує читабельність і викликає потрібні емоції.

7) Організація дизайн-макету та робочий процес — упорядкований макет, стайлгайди, системи іменування шарів та підготовка асетів значно полегшують перехід від етапу дизайну до верстки й розробки.

Загалом, розділ доводить, що веб-дизайн є комплексним процесом, де важливе місце займає як творчий підхід, так і технічна грамотність. Якісний результат досягається лише тоді, коли естетичні рішення підкріплюються принципами юзабіліті та сучасними технологіями.

### **Питання для самоперевірки**

- 1) Що таке графічний дизайн сайту та яку роль він відіграє у створенні веб-ресурсів?
- 2) Які методи генерації ідей дизайну використовуються найчастіше?
- 3) У чому полягає відмінність між симетричним та асиметричним балансом у веб-дизайні?
- 4) Як працює принцип AIDA (Attention – Interest – Desire – Action) у веб-дизайні?
- 5) Які основні зони структури головної сторінки сайту можна виділити?
- 6) Поясніть різницю між фіксованим та еластичним макетом.
- 7) У чому полягає відмінність між responsive (чуйним) та adaptive (адаптивним) дизайном?
- 8) Що таке модульна сітка і які її переваги у веб-дизайні?
- 9) Які основні типи навігаційних меню існують? Які їхні сильні та слабкі сторони?
- 10) Яку роль відіграє типографіка у веб-дизайні?
- 11) Які вимоги слід враховувати при підборі шрифтів для веб-сайтів?
- 12) Що таке «білий простір» і чому він важливий у композиції сайту?
- 13) Які принципи допомагають створити ефективну ієрархію контенту?
- 14) Чим відрізняються wireframe, mockup і готовий макет дизайну?
- 15) Що таке стайлгайд і для чого він використовується?

16) Які основні типи колірних схем ви знаєте? Наведіть приклади їх застосування.

17) Як працює правило «60-30-10» у підборі кольорів?

18) Чому важливим є тестування дизайну на різних пристроях і в різних браузерах?

19) Які переваги дає розробка дизайну безпосередньо у браузері?

20) Чому сучасний веб-дизайн неможливий без урахування UX-принципів?

## РОЗДІЛ 5. ОПТИМІЗАЦІЯ САЙТУ

Оптимізація сайту є одним з найважливіших аспектів сучасної веб-розробки, який безпосередньо впливає на користувацький досвід, швидкість завантаження сторінок та позиції веб-ресурсу в пошукових системах. У сьогоdnішньому висококонкурентному цифровому середовищі недостатньо просто створити функціональний веб-сайт – необхідно забезпечити його максимальну ефективність та продуктивність для досягнення бізнес-цілей.

Процес оптимізації охоплює широкий спектр технічних та стратегічних заходів, починаючи від мінімізації розміру файлів та закінчуючи складними алгоритмами покращення видимості в пошукових системах. Правильно оптимізований сайт не лише швидше завантажується та краще працює на різних пристроях, але й забезпечує вищі позиції в результатах пошуку, що призводить до збільшення трафіку та конверсій.

### 5.1 Види оптимізації

#### 5.1.1 Технічна оптимізація

Технічна оптимізація становить фундаментальну основу для створення швидкого та ефективного веб-сайту, адже саме від якості коду, правильності структури даних та налаштувань сервера залежить те, наскільки швидко користувачі зможуть отримати доступ до контенту та наскільки комфортно буде їхня взаємодія з веб-ресурсом. Цей напрямок оптимізації охоплює широкий спектр технічних аспектів, починаючи від написання чистого HTML-коду та закінчуючи складними серверними налаштуваннями, які впливають на швидкість обробки запитів та ефективність використання ресурсів.

#### Оптимізація HTML-коду

Створення чистого та семантично правильного HTML-коду є одним з найважливіших кроків у процесі технічної оптимізації, оскільки саме структура розмітки визначає, наскільки ефективно браузер зможе обробити та

відобразити сторінку. Семантичний HTML не лише покращує доступність сайту для людей з обмеженими можливостями та допомагає пошуковим системам краще розуміти зміст сторінки, але й значно прискорює процес рендерингу, адже браузеру не доводиться обробляти зайві елементи та складні структури вкладення.

Основний принцип оптимізації HTML полягає у використанні мінімальної кількості тегів для досягнення бажаного результату, при цьому кожен елемент повинен мати чітке семантичне значення та логічно вписуватися в загальну структуру документа. Надлишкове вкладення div-елементів, яке часто зустрічається в неоптимізованому коді, створює додаткове навантаження на браузер та ускладнює підтримку коду розробниками.

```
<!-- Неоптимізований підхід з надмірним вкладенням -->
<div class="wrapper">
  <div class="container">
    <div class="content-wrapper">
      <div class="text-container">
        <div class="paragraph-wrapper">
          <p>Текст новини про університет</p>
        </div>
      </div>
    </div>
  </div>
</div>

<!-- Оптимізований семантичний код -->
<article class="news-article">
  <p>Текст новини про університет</p>
</article>

<!-- Ще один приклад: навігаційне меню -->
```

```

<!-- Неоптимізовано -->
<div class="menu-wrapper">
  <div class="menu-container">
    <div class="menu-items">
      <div class="menu-item"><a
href="/">Головна</a></div>
      <div class="menu-item"><a href="/about">Про
нас</a></div>
    </div>
  </div>
</div>

<!-- Оптимізовано -->
<nav class="main-menu">
  <a href="/">Головна</a>
  <a href="/about">Про нас</a>
</nav>

```

### Мінімізація CSS та JavaScript файлів

Процес мінімізації коду є технічним прийомом, який дозволяє значно зменшити розмір файлів стилів та скриптів шляхом видалення всіх непотрібних символів, які використовуються для покращення читабельності коду розробниками, але не несуть функціонального навантаження для браузера. До таких символів належать пробіли між словами та рядками, табуляції для відступів, коментарі, а також зайві символи нового рядка, які в сукупності можуть становити значну частину від загального розміру файлу.

Мінімізація особливо ефективна для великих проектів, де розмір CSS та JavaScript файлів може сягати сотень кілобайт або навіть мегабайт, адже навіть незначне процентне зменшення розміру таких файлів призводить до помітного прискорення завантаження сторінок, особливо для користувачів з повільним інтернет-з'єднанням або мобільних пристроїв.



```

/* Оригінальний CSS з форматуванням для розробників */
.university-header {
    background-color: #ffffff;
    padding: 25px 20px;
    margin-bottom: 40px;
    border-radius: 12px;
    box-shadow: 0 2px 10px rgba(0, 0, 0, 0.1);
}

.course-card {
    background: linear-gradient(135deg, #667eea 0%,
#764ba2 100%);
    border: none;
    color: white;
}

/* Мінімізований CSS (економія ~65% розміру) */
.university-header{background-color:#fff;padding:25px
20px;margin-bottom:40px;border-radius:12px;box-shadow:0
2px 10px rgba(0,0,0,.1)}.course-card{background:linear-
gradient(135deg,#667eea 0%,#764ba2
100%);border:none;color:#fff}
javascript
// Оригінальний JavaScript код
function calculateStudentGrade(homeworkScore,
examScore, participationScore) {
    // Перевіряємо коректність вхідних параметрів
    if (homeworkScore < 0 || examScore < 0 ||
participationScore < 0) {

```

```

        return "Помилка: негативні оцінки
неприпустимі";
    }

    // Розраховуємо загальну оцінку з урахуванням ваги
кожного компонента
    const finalGrade = (homeworkScore * 0.3) +
(examScore * 0.6) + (participationScore * 0.1);
    return Math.round(finalGrade);
}

// Мінімізований JavaScript
function calculateStudentGrade(a,b,c){return
a<0||b<0||c<0?"Помилка: негативні оцінки
неприпустимі":Math.round(.3*a+.6*b+.1*c) }

```

### Оптимізація зображень та мультимедіа контенту

Зображення зазвичай становлять найбільшу частину від загального розміру веб-сторінки, тому їх оптимізація може принести найбільший ефект у плані прискорення завантаження сайту. Сучасні технології надають розробникам потужні інструменти для значного зменшення розміру графічних файлів без втрати візуальної якості, що особливо важливо в епоху мобільного інтернету, коли користувачі часто мають обмежений трафік або повільне з'єднання.

Використання сучасних форматів зображень, таких як WebP та AVIF, дозволяє досягти компресії на 25-50% кращої порівняно з традиційними JPEG та PNG форматами, при цьому зберігаючи ту ж візуальну якість. Крім того, важливим аспектом оптимізації є впровадження технології ледачого завантаження (lazy loading), яка дозволяє завантажувати зображення лише тоді, коли користувач наближається до них під час прокручування сторінки.

```
<!-- Базова оптимізація зображення -->
```

```


<!-- Прогресивна оптимізація з підтримкою різних
форматів -->
<picture>
    <source srcset="student-photo.avif"
type="image/avif">
    <source srcset="student-photo.webp"
type="image/webp">
    
</picture>

<!-- Адаптивні зображення для різних розмірів екрану --
>
<picture>
    <source media="(max-width: 480px)" srcset="student-
small.webp" type="image/webp">
    <source media="(max-width: 768px)" srcset="student-
medium.webp" type="image/webp">
    <source srcset="student-large.webp"
type="image/webp">
    
</picture>

<!-- Оптимізація фонового зображення через CSS -->
<div class="hero-section" style="background-image:
url('university-hero.webp');">
```

```
<h1>Навчайся в кращому технічному університеті</h1>
</div>
```

### 5.1.2 SEO-оптимізація (пошукова оптимізація)

SEO-оптимізація, або пошукова оптимізація, являє собою комплекс технічних і контентних заходів, спрямованих на підвищення позицій веб-сайту в результатах видачі пошукових систем за релевантними запитами користувачів. Основною метою SEO є забезпечення максимальної видимості веб-ресурсу для цільової аудиторії через органічні, тобто неоплачені, результати пошуку.



Рисунок 5.1 — SEO-оптимізація

Фундаментом успішної SEO-стратегії є розуміння того, як працюють пошукові системи, зокрема Google, який домінує на українському ринку пошукових послуг. Пошукові роботи постійно сканують мільярди веб-сторінок, індексують їх контент і ранжують за складними алгоритмами, що враховують понад 200 різних факторів. Наприклад, якщо ваш сайт про кулінарію містить якісний рецепт борщу з детальним описом та фотографіями, пошукова система може показати його на першій сторінці результатів за запитом "рецепт борщу".

Ключові слова становлять серцевину пошукової оптимізації, оскільки вони є містком між пошуковими запитами користувачів і контентом вашого сайту. Процес дослідження ключових слів передбачає аналіз популярних пошукових фраз у вашій тематиці, визначення їх частотності та рівня конкуренції. Для інтернет-магазину взуття ключовими словами можуть бути "купити кросівки Київ", "жіноче взуття онлайн" або "зимові черевики з доставкою". Важливо враховувати як високочастотні загальні запити, так і довгохвостові специфічні фрази, які часто мають вищий рівень конверсії.

Внутрішня оптимізація сайту охоплює технічні аспекти, які безпосередньо впливають на сприйняття ресурсу пошуковими системами. Title-теги повинні містити основні ключові слова та бути унікальними для кожної сторінки, наприклад: "Рецепти української кухні - кулінарний блог 'Смачно'". Meta-описи, хоча й не впливають безпосередньо на ранжування, формують перше враження користувача про вашу сторінку в результатах пошуку і можуть значно підвищити клікабельність. Заголовки H1-H6 створюють ієрархічну структуру контенту, допомагаючи пошуковим системам зрозуміти тематичну організацію інформації на сторінці.

Технічна оптимізація включає забезпечення швидкості завантаження сторінок, оскільки користувачі зазвичай залишають сайти, які завантажуються довше трьох секунд. Адаптивний дизайн, який коректно відображається на мобільних пристроях, став критично важливим фактором, адже понад половина пошукових запитів здійснюється зі смартфонів. SSL-сертифікат

забезпечує безпечне з'єднання та є обов'язковим елементом довіри як для користувачів, так і для пошукових систем. Створення XML-карти сайту допомагає пошуковим роботам ефективніше індексувати всі важливі сторінки вашого ресурсу.

Контентна стратегія в SEO базується на принципі створення цінного, унікального та релевантного контенту, який відповідає інформаційним потребам цільової аудиторії. Якісний контент природно залучає зворотні посилання від інших сайтів, що є потужним сигналом авторитетності для пошукових систем. Регулярне оновлення контенту, використання мультимедійних елементів та структурування інформації за допомогою списків, підзаголовків і таблиць покращує користувацький досвід та збільшує час перебування відвідувачів на сайті.

Зовнішня оптимізація зосереджується на побудові профілю зворотних посилань від авторитетних і тематично релевантних ресурсів. Якісне посилання з популярного новинного сайту на статтю вашого блогу може значно підвищити її позиції в пошуковій видачі. Однак важливо уникати маніпулятивних практик, таких як покупка посилань або участь у схемах обміну посиланнями, які можуть призвести до санкцій від пошукових систем. Натомість варто зосередитися на створенні контенту, який природно заслуговує на посилання, та активній участі в професійних спільнотах.

Локальна SEO-оптимізація особливо важлива для бізнесу, який має фізичну присутність або обслуговує конкретний географічний регіон. Оптимізація Google My Business профілю, збір позитивних відгуків клієнтів і використання локальних ключових слів допомагають з'являтися в локальних результатах пошуку. Ресторан у центрі Львова може оптимізувати свій сайт під запити "ресторан у Львові", "де поїсти біля Ратуші" або "доставка піци Галицький район".

Аналітика та моніторинг результатів SEO-діяльності здійснюються за допомогою спеціалізованих інструментів, таких як Google Analytics, Google Search Console, або платформи на кшталт SEMrush. Ці сервіси дозволяють

відстежувати позиції сайту за ключовими словами, аналізувати трафік, виявляти технічні проблеми та оцінювати ефективність впроваджених оптимізаційних заходів. Регулярний моніторинг допомагає своєчасно реагувати на зміни в алгоритмах пошукових систем та коригувати стратегію просування.

### **On-page SEO та його компоненти**

On-page SEO являє собою сукупність оптимізаційних дій, які виконуються безпосередньо на сторінках вашого веб-сайту з метою покращення їх позицій у пошукових результатах. На відміну від зовнішньої оптимізації, ці заходи повністю знаходяться під вашим контролем і можуть бути впроваджені негайно після внесення відповідних змін у код сторінки.

```
<!DOCTYPE html>
<html lang="uk">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <title>Оптимізація сайту - повний гід по покращенню
продуктивності</title>
  <meta name="description" content="Дізнайтеся про
ефективні методи оптимізації сайту: технічні аспекти,
SEO та UX покращення для збільшення трафіку та
конверсій.">
  <meta name="keywords" content="оптимізація сайту,
SEO, швидкість завантаження, веб-продуктивність">
  <link rel="canonical"
href="https://example.com/optimizatsiya-saytu">
</head>
```

### **Аналіз HTML-структури**

Розглянемо детально кожен елемент наведеного HTML-коду:

Title-тег "Оптимізація сайту - повний гід по покращенню продуктивності" є одним з найважливіших елементів SEO, оскільки він відображається як заголовок у результатах пошуку і повинен містити основне ключове слово на початку. Цей заголовок має оптимальну довжину близько 60 символів, що забезпечує його повне відображення в більшості пошукових систем. Додавання конкретизуючих слів як "повний гід" та "покращення продуктивності" робить заголовок більш привабливим для користувачів і допомагає виділитися серед конкурентів.

Meta-опис містить стислу, але інформативну анотацію сторінки довжиною до 160 символів, яка спонукає користувачів перейти на ваш сайт. Фраза "Дізнайтеся про ефективні методи" створює відчуття цінності контенту, а перерахування конкретних аспектів "технічні аспекти, SEO та UX покращення" показує широту охоплення теми. Завершальна частина "для збільшення трафіку та конверсій" апелює до практичних результатів, які цікавлять власників сайтів.

Canonical-тег вказує пошуковим системам на основну версію сторінки, що особливо важливо, коли ідентичний контент може бути доступний за різними URL-адресами. Наприклад, якщо ваша сторінка доступна як за адресою "example.com/page", так і "example.com/page?utm\_source=social", canonical-тег допоможе уникнути проблем з дублюванням контенту.

### **Структурована розмітка Schema.org**

Structured data markup або структурована розмітка являє собою стандартизований спосіб надання додаткової інформації про контент сторінки пошуковим системам. Використання JSON-LD формату, як показано в прикладі, є рекомендованим Google підходом, оскільки він не впливає на візуальне відображення сторінки і легко читається як людиною, так і машинами.

```
<script type="application/ld+json">
{
  "@context": "https://schema.org",
```



```

"@type": "Article",
"headline": "Оптимізація сайту",
"author": {
  "@type": "Person",
  "name": "Викладач технологій інтернет"
},
"datePublished": "2024-01-15",
"description": "Комплексний гід з оптимізації веб-сайтів"
}
</script>

```

У наведеному прикладі розмітка типу "Article" повідомляє пошуковим системам, що дана сторінка містить статтю, а не продукт чи послугу. Поле "headline" дублює основний заголовок статті, що допомагає пошуковим системам ідентифікувати головну тему контенту. Інформація про автора через тип "Person" може призвести до відображення авторства у результатах пошуку, що підвищує довіру користувачів до контенту.

Дата публікації "datePublished" допомагає пошуковим системам визначити актуальність контенту, що особливо важливо для новинних і технічних матеріалів, де свіжість інформації має критичне значення. Наприклад, стаття про останні зміни в алгоритмі Google, опублікована місяць тому, буде більш релевантною, ніж аналогічна стаття річної давності.

### **Практичні переваги структурованої розмітки**

Правильне використання Schema.org може призвести до появи розширених сніпетів у результатах пошуку, які займають більше місця і привертають більше уваги користувачів. Для статей це можуть бути зірочки рейтингу, дата публікації, зображення автора або час читання. Для рецептів можуть відображатися інгредієнти, час приготування та калорійність прямо в результатах пошуку.

Розмітка товарів може включати ціну, наявність на складі та рейтинг, що значно підвищує привабливість вашого інтернет-магазину в очах потенційних покупців. Події можуть відображати дату, місце проведення та ціну квитків, що робить їх більш помітними серед конкурентів.

### **Технічні рекомендації та найкращі практики**

При впровадженні on-page оптимізації важливо дотримуватися принципу природності - всі елементи повинні виглядати органічно і служити користувачам, а не лише пошуковим системам. Надмірне насичення ключовими словами або створення мета-описів, які не відповідають реальному контенту сторінки, може призвести до негативних наслідків у вигляді зниження позицій або навіть санкцій.

Регулярна перевірка та оновлення мета-тегів відповідно до змін у контенті забезпечує актуальність інформації, що відображається в результатах пошуку. Використання інструментів для перевірки структурованої розмітки, таких як Google Rich Results Test, допомагає виявити помилки та переконатися в коректності реалізації markup-коду.

### **5.1.3 UX/UI оптимізація: комплексний підхід до покращення користувацького досвіду**

UX/UI оптимізація представляє собою системний процес удосконалення веб-сайту з точки зору зручності використання та візуальної привабливості, який безпосередньо впливає на успішність онлайн-проекту. User Experience охоплює весь шлях користувача від моменту входу на сайт до досягнення його мети, тоді як User Interface зосереджується на візуальних та інтерактивних елементах, через які відбувається це спілкування.

Статистичні дані підтверджують критичну важливість якісного UX/UI: згідно з дослідженнями, 88% користувачів не повертаються на сайт після негативного досвіду взаємодії, а кожна додаткова секунда завантаження сторінки може зменшити конверсію на 7%. Інвестиції в UX/UI оптимізацію

зазвичай окупаються збільшенням часу перебування користувачів на сайті, зростанням кількості переглядів сторінок та підвищенням рівня конверсії.

### **Адаптивний дизайн: mobile-first стратегія**

```
/* Mobile-first підхід */  
.container {  
    width: 100%;  
    padding: 15px;  
}  
  
/* Планшети */  
@media (min-width: 768px) {  
    .container {  
        max-width: 750px;  
        margin: 0 auto;  
        padding: 20px;  
    }  
}  
  
/* Десктоп */  
@media (min-width: 1200px) {  
    .container {  
        max-width: 1170px;  
        padding: 30px;  
    }  
}
```

Розглянутий CSS-код демонструє mobile-first підхід, який є сучасним стандартом веб-розробки у зв'язку з домінуванням мобільного трафіку. Базові стилі спочатку визначаються для найменших екранів, а потім поступово розширюються для більших пристроїв через media queries.

Початковий стиль `.container` з шириною `100%` та відступами `15px` забезпечує оптимальне відображення на смартфонах, де кожен піксель екрану є цінним. Такий підхід гарантує, що контент буде читабельним навіть на старих моделях телефонів з малими екранами, наприклад iPhone SE з діагоналлю 4 дюйми.

Media query для планшетів `@media (min-width: 768px)` активується на пристроях шириною від 768 пікселів, що включає iPad у портретній орієнтації та більші смартфони. Максимальна ширина контейнера обмежується 750 пікселями з centruванням через `margin: 0 auto`, що забезпечує зручне читання без надто довгих рядків тексту. Збільшення `padding` до `20px` створює більше "повітря" навколо контенту.

Десктопна версія з breakpoint `1200px` розрахована на повноцінні комп'ютери та ноутбуки, де контейнер може розширитися до 1170 пікселів. Додаткові відступи в `30px` використовують переваги більшого простору екрану для створення більш просторого та елегантного макету.

### Оптимізація швидкості завантаження через lazy loading

```
// Ледаче завантаження зображень
const images =
document.querySelectorAll('img[loading="lazy"]');
const imageObserver = new
IntersectionObserver((entries, observer) => {
  entries.forEach(entry => {
    if (entry.isIntersecting) {
      const img = entry.target;
      img.src = img.dataset.src;
      img.classList.remove('lazy');
      observer.unobserve(img);
    }
  });
});
```

```
images.forEach(img => imageObserver.observe(img));
```

Представлений JavaScript-код реалізує технологію ледачого завантаження зображень, яка є однією з найефективніших методик оптимізації продуктивності веб-сайтів. Intersection Observer API дозволяє відстежувати момент, коли зображення з'являється у видимій області екрану користувача, і лише тоді ініціює його завантаження.

Механізм роботи коду базується на спостереженні за елементами `img` з атрибутом `loading="lazy"`. Замість безпосереднього вказування джерела зображення в атрибуті `src`, реальна адреса зберігається в `data-src`, що запобігає автоматичному завантаженню браузером. Коли користувач прокручує сторінку і зображення потрапляє в зону видимості, спрацьовує `callback`-функція `observer'a`.

Практична користь цього підходу особливо помітна на сторінках з великою кількістю зображень, таких як галереї, каталоги товарів або блоги з фотографіями. Наприклад, інтернет-магазин одягу з 100 фотографіями товарів може заощадити до 80% трафіку та значно прискорити первинне завантаження сторінки, оскільки користувач зазвичай переглядає лише перші 10-15 позицій.

### **Вплив на користувацький досвід та бізнес-показники**

Швидкість завантаження безпосередньо корелює з поведінковими метриками: згідно з дослідженнями Google, сайти, які завантажуються за 1-3 секунди, мають показник відмов на 32% нижчий порівняно з тими, що завантажуються за 5 секунд. Amazon підрахував, що кожні 100 мілісекунд затримки коштують їм 1% від продажів.

Адаптивний дизайн впливає не лише на зручність використання, але й на SEO-позиції сайту, оскільки Google використовує `mobile-first` індексацію. Це означає, що пошукова система оцінює та ранжує сайти в першу чергу на основі їх мобільної версії. Сайт з неадаптивним дизайном може втратити значну частину органічного трафіку незалежно від якості контенту.

### **Додаткові аспекти UX/UI оптимізації**

Кольорова схема та контрастність повинні забезпечувати комфортне читання для користувачів з різним рівнем зору. Рекомендований коефіцієнт контрастності WCAG становить мінімум 4.5:1 для звичайного тексту та 3:1 для великого тексту. Наприклад, темно-сірий текст #333333 на білому фоні забезпечує відмінну читабельність без надмірної контрастності.

Навігаційна структура повинна бути інтуїтивно зрозумілою та послідовною на всіх сторінках сайту. Правило "трьох кліків" рекомендує розташовувати будь-яку інформацію не глибше трьох рівнів від головної сторінки. Хлібні крихти та логічна ієрархія меню допомагають користувачам орієнтуватися в структурі сайту.

Інтерактивні елементи, такі як кнопки та посилання, повинні мати достатній розмір для зручного натискання на сенсорних екранах - мінімум 44x44 пікселі згідно з рекомендаціями Apple та Google. Забезпечення візуального зворотного зв'язку при взаємодії, наприклад зміна кольору кнопки при наведенні курсору, покращує відчуття контролю користувача над інтерфейсом.

## **5.2 Аудит сайту і внутрішня оптимізація**

### **5.2.1 Технічний аудит**

Технічний аудит представляє собою фундаментальний процес всебічної діагностики веб-ресурсу, який дозволяє виявити приховані проблеми та вузькі місця, що можуть суттєво впливати на продуктивність сайту, його позиції в пошукових системах та загальний користувацький досвід. Цей процес є критично важливим для будь-якого серйозного веб-проекту, оскільки навіть незначні технічні недоліки можуть призвести до втрати значної частки трафіку та потенційних клієнтів.

Професійний технічний аудит охоплює множину аспектів: від аналізу серверних налаштувань і безпеки до перевірки коректності HTML-розмітки та оптимізації ресурсів. Регулярне проведення таких аудитів дозволяє

підтримувати веб-сайт у оптимальному стані та своєчасно реагувати на зміни в технологічному ландшафті інтернету.

### **Детальний аналіз швидкості завантаження**

```
// Вимірювання продуктивності за допомогою Performance
API
window.addEventListener('load', () => {
    const navigation =
performance.getEntriesByType('navigation')[0];

    console.log('Час завантаження DOM:',
navigation.domContentLoadedEventEnd -
navigation.domContentLoadedEventStart);
    console.log('Повний час завантаження:',
navigation.loadEventEnd - navigation.loadEventStart);
    console.log('Час відповіді сервера:',
navigation.responseEnd - navigation.requestStart);

    // Надсилання метрик до аналітики
    if ('sendBeacon' in navigator) {
        const data = {
            domLoad:
navigation.domContentLoadedEventEnd -
navigation.domContentLoadedEventStart,
            fullLoad: navigation.loadEventEnd -
navigation.loadEventStart,
            serverResponse: navigation.responseEnd -
navigation.requestStart
        };
        navigator.sendBeacon('/analytics',
JSON.stringify(data));
    }
});
```

```
    }  
  } ) ;
```

Швидкість завантаження сторінок є одним з найкритичніших факторів успіху веб-проекту, який впливає на все: від користувацького досвіду до SEO-ранжування. Сучасні користувачі очікують миттєвої реакції від веб-сайтів, і навіть затримка в кілька секунд може призвести до втрати потенційного клієнта або читача.

Представлений JavaScript-код демонструє використання Performance API для отримання точних метрик завантаження безпосередньо в браузері користувача. Navigation Timing API надає детальну інформацію про кожен етап завантаження сторінки, від ініціації запиту до повного завершення рендерингу.

Метрика `domContentLoadedEventEnd` - `domContentLoadedEventStart` показує час, необхідний для обробки події `DOMContentLoaded`, яка сигналізує про завершення парсингу HTML та готовність DOM до взаємодії. Цей показник критично важливий для оцінки швидкості інтерактивності сторінки, оскільки саме після цієї події користувач може почати взаємодіяти з елементами інтерфейсу.

Повний час завантаження, розрахований як `loadEventEnd` - `loadEventStart`, включає завантаження всіх ресурсів сторінки: зображень, стилів, скриптів та інших файлів. Цей показник відображає момент, коли сторінка повністю готова до використання, включаючи всі мультимедійні елементи.

Час відповіді сервера `responseEnd` - `requestStart` є критично важливою метрикою, яка показує ефективність серверної інфраструктури. Повільна відповідь сервера може вказувати на проблеми з хостингом, неоптимізовані запити до бази даних або необхідність впровадження кешування.

Використання `navigator.sendBeacon()` для відправки метрик до аналітичної системи є оптимальним рішенням, оскільки цей метод не блокує закриття сторінки і гарантує доставку даних навіть якщо користувач швидко



покидає сайт. Це дозволяє збирати реальні дані про продуктивність від усіх відвідувачів.

### **Семантичний аналіз HTML-структури**

Правильна HTML-структура є основою доступного та SEO-оптимізованого веб-сайту. Семантична розмітка не лише покращує розуміння контенту пошуковими системами, але й забезпечує коректну роботу допоміжних технологій для користувачів з обмеженими можливостями.

```
<!-- Правильна семантична структура -->
<!DOCTYPE html>
<html lang="uk">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <title>Заголовок сторінки</title>
</head>
<body>
  <header role="banner">
    <nav role="navigation" aria-label="Головне
меню">
      <ul>
        <li><a href="/">Головна</a></li>
        <li><a
href="/products">Продукти</a></li>
        <li><a
href="/contact">Контакти</a></li>
      </ul>
    </nav>
  </header>
```

```

<main role="main">
  <article>
    <header>
      <h1>Заголовок статті</h1>
      <time datetime="2024-01-15">15 січня
2024</time>
    </header>
    <section>
      <h2>Підзаголовок секції</h2>
      <p>Текст контенту...</p>
    </section>
  </article>
</main>

<aside role="complementary">
  <h2>Бічна панель</h2>
  <p>Додатковий контент...</p>
</aside>

<footer role="contentinfo">
  <p>© 2024 Назва компанії</p>
</footer>
</body>
</html>

```

Представлена HTML-структура демонструє сучасні найкращі практики веб-розробки. Атрибут `lang="uk"` в тезі `<html>` повідомляє браузерам і скрін-рідерам про мову контенту, що забезпечує коректну вимову тексту синтезаторами мовлення та правильну роботу автоматичних перекладачів.

Використання ролей ARIA, таких як `role="banner"`, `role="navigation"`, `role="main"`, значно покращує доступність сайту для користувачів допоміжних

технологій. Ці ролі створюють семантичні орієнтири, які допомагають людям з порушеннями зору швидко навігувати по сторінці та знаходити потрібний контент.

Елемент `<nav>` з атрибутом `aria-label="Головне меню"` чітко ідентифікує навігаційну область та надає їй зрозумілу назву. Це особливо важливо на сторінках з кількома навігаційними блоками, наприклад основним меню, хлібними крихтами та футерними посиланнями.

Семантичний тег `<main>` визначає основний контент сторінки і повинен бути унікальним для кожної сторінки сайту. Всередині `main` розташовується `<article>`, який представляє самостійний контентний блок, що може існувати незалежно від решти сторінки, наприклад новинна стаття або запис у блозі.

Використання тегу `<time>` з атрибутом `datetime` забезпечує машиночитний формат дати, що дозволяє пошуковим системам та браузерам коректно інтерпретувати темпоральну інформацію. Це може призвести до відображення дати публікації безпосередньо в результатах пошуку.

### **Практичні рекомендації для технічного аудиту**

При проведенні технічного аудиту важливо використовувати комплексний підхід, який включає автоматизовані інструменти та ручну перевірку. Google Lighthouse надає загальну оцінку продуктивності, доступності та SEO, але не завжди враховує специфічні особливості конкретного проекту.

GTmetrix дозволяє детально проаналізувати каскадне завантаження ресурсів та ідентифікувати елементи, які найбільше сповільнюють сторінку. WebPageTest надає можливість тестування з різних географічних локацій та типів з'єднань, що особливо важливо для глобальних проектів.

Валідація HTML-коду через W3C Markup Validator допомагає виявити синтаксичні помилки, які можуть призвести до непередбачуваного відображення сторінки в різних браузерах. Крос-браузерне тестування залишається критично важливим, оскільки різні браузери можуть по-різному інтерпретувати навіть валідний код.

Аналіз мобільної версії сайту повинен включати тестування на реальних пристроях з різними розмірами екранів та швидкістю інтернет-з'єднання. Емуляція мобільних пристроїв у браузерних інструментах розробника не завжди точно відображає реальний користувацький досвід.

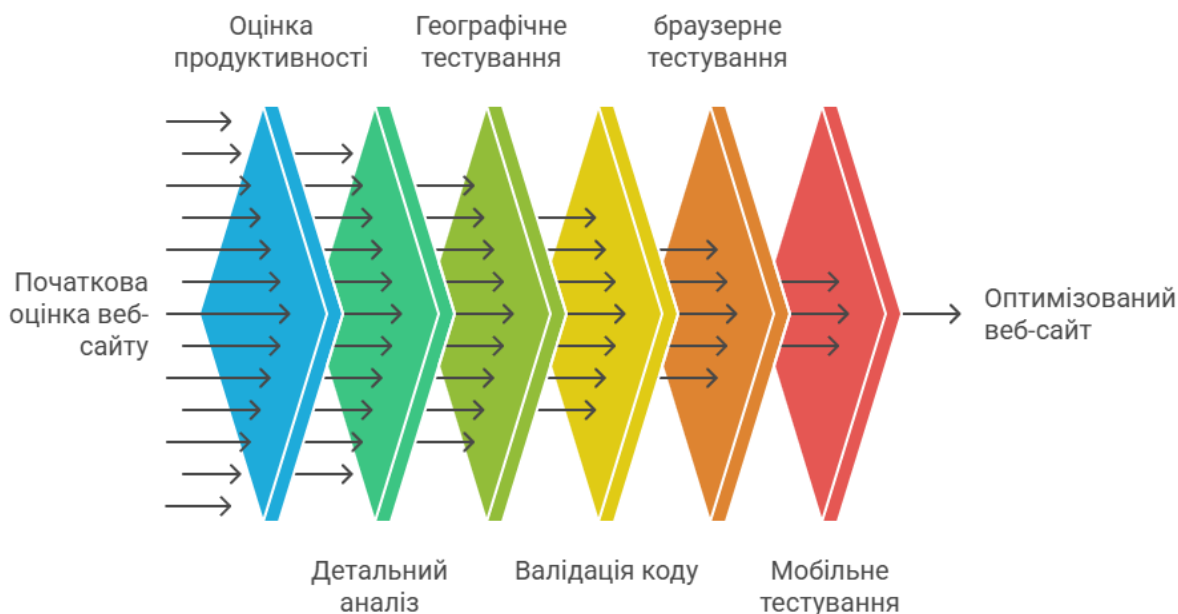


Рисунок 5.2 — Технічний аудит веб-сайту

### 5.2.2 Оптимізація контенту

Оптимізація контенту є багатогранним процесом, який поєднує мистецтво створення захоплюючого та корисного контенту з наукою технічної оптимізації для пошукових систем. Сучасний підхід до контентної стратегії вимагає розуміння не лише потреб цільової аудиторії, але й алгоритмів пошукових систем, принципів доступності та технічних аспектів веб-розробки.

Якісний контент служить кількома цілями одночасно: він задовольняє інформаційні потреби користувачів, підвищує авторитет веб-ресурсу в очах пошукових систем, сприяє природному залученню зворотних посилань та збільшує час перебування відвідувачів на сайті. Інвестиції в контентну

оптимізацію зазвичай дають довготривалі результати, на відміну від платної реклами, ефект якої припиняється після закінчення бюджету.

### Структурування та оптимізація текстового контенту

```
<article class="content-article">
  <header>
    <h1>Головний заголовок статті з основним
ключовим словом</h1>
    <p class="article-meta">
      <time datetime="2024-01-15">15 січня
2024</time>
      <span>Автор: Викладач</span>
    </p>
  </header>

  <div class="article-content">
    <h2>Підзаголовок з другорядним ключовим
словом</h2>
    <p>Перший абзац тексту, який містить природне
включення ключових слів та надає корисну інформацію для
читачів. Важливо підтримувати баланс між SEO-
оптимізацією та читабельністю.</p>

    <h3>Більш специфічний підзаголовок</h3>
    <p>Детальний опис або пояснення, яке розкриває
тему більш глибоко. Використання синонімів та
пов'язаних термінів допомагає покращити семантичне
розуміння контенту пошуковими системами.</p>

    <ul>
      <li>Перший важливий пункт списку</li>
```

```

        <li>Другий пункт з додатковою
інформацією</li>
        <li>Третій пункт, який підсумовує основні
ідеї</li>
    </ul>
</div>
</article>

```

Представлена HTML-структура демонструє оптимальну організацію текстового контенту, яка враховує як SEO-вимоги, так і потреби користувачів. Використання семантичних тегів створює логічну ієрархію інформації, яку легко сприймають як люди, так і пошукові роботи.

Заголовок H1 "Головний заголовок статті з основним ключовим словом" виконує подвійну функцію: він чітко повідомляє читачеві про тему статті та містить основне ключове слово, за яким сторінка має ранжуватися в пошукових результатах. Критично важливо, щоб на кожній сторінці був лише один H1, який точно відображає основну тематику контенту.

Мета-інформація в елементі `<p class="article-meta">` включає дату публікації в машиночитному форматі та інформацію про автора. Ця інформація не лише корисна для користувачів, але й може відображатися в розширених сніпетах пошукових результатів, підвищуючи клікабельність сторінки.

Ієрархія заголовків H2 та H3 створює логічну структуру контенту, яка допомагає користувачам швидко сканувати текст та знаходити потрібну інформацію. Пошукові системи використовують цю ієрархію для кращого розуміння тематичної структури сторінки та виявлення найважливіших розділів контенту.

Текстові абзаци демонструють принцип природної інтеграції ключових слів без надмірного насичення. Фраза "Важливо підтримувати баланс між SEO-оптимізацією та читабельністю" підкреслює ключовий принцип сучасної контентної оптимізації: контент повинен в першу чергу служити користувачам, а не пошуковим системам.

Використання маркованого списку покращує сприйняття інформації та збільшує ймовірність потрапляння в featured snippets пошукових систем. Структуровані списки особливо ефективні для інструкцій, переліків переваг або характеристик продуктів.

### Прогресивна оптимізація мультимедіа

```
<!-- Оптимізоване зображення з різними форматами -->
<figure class="optimized-image">
  <picture>
    <source media="(min-width: 800px)"
      srcset="large-image.avif 1x, large-
image@2x.avif 2x"
      type="image/avif">
    <source media="(min-width: 800px)"
      srcset="large-image.webp 1x, large-
image@2x.webp 2x"
      type="image/webp">
    <source media="(max-width: 799px)"
      srcset="small-image.avif 1x, small-
image@2x.avif 2x"
      type="image/avif">
    <source media="(max-width: 799px)"
      srcset="small-image.webp 1x, small-
image@2x.webp 2x"
      type="image/webp">
    
```

```

</picture>
<figcaption>Підпис до зображення з додатковим
контекстом</figcaption>
</figure>

```

Код оптимізації зображень демонструє сучасний підхід до адаптивного завантаження мультимедійного контенту з урахуванням різних типів пристроїв та швидкості інтернет-з'єднання. Використання елемента `<picture>` дозволяє браузеру вибрати найкращий варіант зображення залежно від контексту відображення.

Формат AVIF, розроблений альянсом Alliance for Open Media, забезпечує на 50% кращу компресію порівняно з JPEG при збереженні якості зображення. Однак підтримка цього формату ще не універсальна, тому код включає fallback через WebP та традиційний JPEG. Така градація забезпечує оптимальний баланс між якістю, розміром файлів та сумісністю.

Media queries в атрибутах media дозволяють завантажувати різні версії зображень залежно від розміру екрану. Для мобільних пристроїв завантажуються менші зображення, що економить трафік та прискорює завантаження, особливо критично для користувачів з обмеженими тарифними планами.

Атрибут srcset з дескрипторами 1x та 2x забезпечує підтримку екранів з високою щільністю пікселів, таких як Retina дисплеї. Зображення @2x автоматично завантажується на пристроях з pixel ratio 2.0 і вище, гарантуючи чіткість відображення навіть на найсучасніших екранах.

### **Доступність та SEO-оптимізація мультимедіа**

Атрибут alt з детальним описом зображення служить кільком цілям одночасно: він забезпечує доступність контенту для користувачів з порушеннями зору, надає контекст пошуковим системам для розуміння змісту зображення та відображається в браузері, якщо зображення не завантажилося з технічних причин.



Ледаче завантаження через `loading="lazy"` відкладає завантаження зображень до моменту, коли вони потрапляють у видиму область екрану. Це значно прискорює первинне завантаження сторінки, особливо критично для довгих сторінок з великою кількістю зображень.

Явне вказування розмірів через атрибути `width` та `height` запобігає `layout shift` - неприємному стрибку контенту під час завантаження зображень. Браузер резервує необхідний простір ще до завантаження зображення, що покращує Core Web Vitals показники, які Google враховує при ранжуванні.

Елемент `<figcaption>` надає додатковий контекст зображенню, який може бути корисним як для користувачів, так і для пошукових систем. Підписи до зображень часто читаються більш уважно, ніж основний текст, тому вони є ефективним місцем для розміщення важливої інформації.

### **Стратегічні рекомендації з контентної оптимізації**

Регулярний аудит існуючого контенту допомагає виявити сторінки з низькою ефективністю та можливості для їх покращення. Аналіз поведінкових метрик, таких як час на сторінці, глибина перегляду та показник відмов, надає цінну інформацію про якість та релевантність контенту.

Оновлення старого контенту часто більш ефективне, ніж створення нового, оскільки існуючі сторінки вже мають певний авторитет у пошукових системах. Додавання нової інформації, оновлення статистичних даних та покращення структури може значно підвищити позиції в пошукових результатах.

Кластеризація контенту навколо основних тем створює семантичне ядро сайту та допомагає пошуковим системам краще розуміти експертизу ресурсу в конкретних галузях. Внутрішня перелінковка між тематично пов'язаними сторінками посилює цей ефект та покращує користувацький досвід.

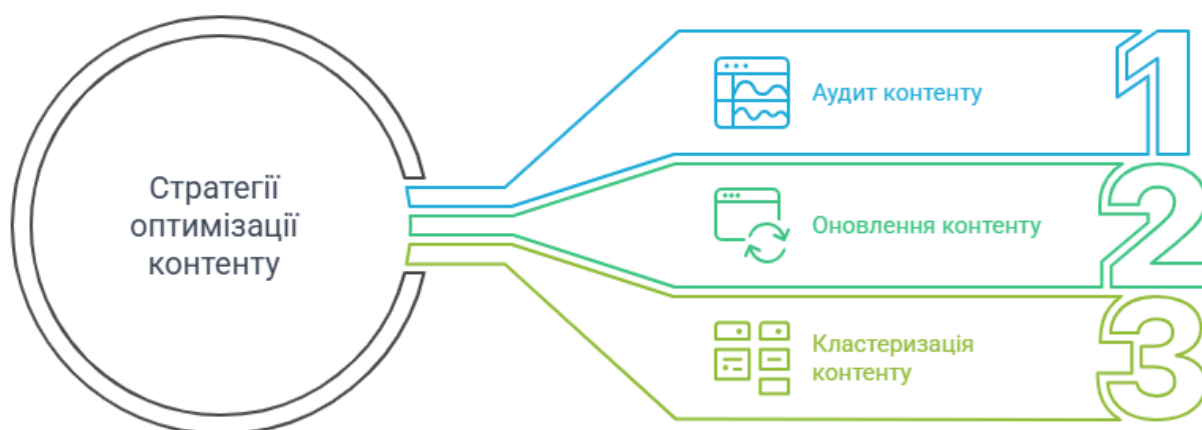


Рисунок 5.3 — Розкриття стратегій оптимізації контенту

### 5.2.3 Внутрішня перелінковка

Внутрішня перелінковка являє собою стратегічну систему з'єднання сторінок веб-сайту через гіперпосилання, яка служить навігаційним каркасом для користувачів і сигнальною мережею для пошукових систем. Ця технологія є одним з найпотужніших та найбільш контрольованих інструментів SEO-оптимізації, оскільки вебмастер має повний контроль над структурою та розподілом посилань у межах свого ресурсу.

Ефективна система внутрішньої перелінковки вирішує кілька критично важливих завдань одночасно: вона направляє користувачів до релевантного контенту, розподіляє авторитет сторінок за принципом PageRank, допомагає пошуковим роботам виявляти та індексувати новий контент, а також створює семантичні зв'язки між тематично пов'язаними матеріалами. Дослідження показують, що сайти з добре організованою внутрішньою перелінковкою мають на 40% більший органічний трафік порівняно з ресурсами з хаотичною структурою посилань.

```
<!-- Приклад стратегічної внутрішньої перелінковки -->
<nav class="breadcrumb" aria-label="Хлібні крихти">
  <ol>
    <li><a href="/">Головна</a></li>
```

```

        <li><a href="/tehnologii-internet">Технології
Інтернет</a></li>
        <li aria-current="page">Оптимізація сайту</li>
    </ol>
</nav>

<article>
    <p>Оптимізація сайту включає багато аспектів,
включаючи
        <a href="/seo-osnovy" title="Дізнайтеся про основи
SEO">пошукову оптимізацію</a>,
        <a href="/web-produktyvnist" title="Поради щодо
покращення продуктивності">підвищення швидкості</a>
        та <a href="/ux-design" title="Принципи хорошого UX
дизайну">покращення користувацького досвіду</a>.</p>
</article>

<aside class="related-articles">
    <h3>Пов'язані статті</h3>
    <ul>
        <li><a href="/html-semantyka">Семантична
розмітка HTML</a></li>
        <li><a href="/css-optimizatsiya">Оптимізація
CSS для продуктивності</a></li>
        <li><a href="/javascript-
performance">JavaScript та продуктивність</a></li>
    </ul>
</aside>

```

**Хлібні крихти: навігаційна основа користувацького досвіду**

Представлена навігаційна структура хлібних крихт демонструє класичну ієрархічну організацію, яка відображає положення поточної сторінки в загальній структурі сайту. Використання семантичного елементу `<nav>` з атрибутом `aria-label="Хлібні крихти"` забезпечує доступність для користувачів допоміжних технологій, чітко ідентифікуючи цей блок як навігаційний елемент.

Упорядкований список `<ol>` логічно підходить для хлібних крихт, оскільки порядок елементів має семантичне значення - він відображає ієрархію від загального до конкретного. Кожен рівень ієрархії представляє більш конкретну категорію: "Головна" → "Технології Інтернет" → "Оптимізація сайту".

Атрибут `aria-current="page"` на останньому елементі повідомляє скрін-рідерам, що це поточна сторінка, на якій знаходиться користувач. Цей елемент зазвичай не є посиланням, оскільки немає сенсу створювати посилання на ту ж саму сторінку, на якій вже перебуває користувач.

Хлібні крихти особливо корисні на великих сайтах з глибокою ієрархією, таких як інтернет-магазини з множинними категоріями товарів або освітні ресурси з багаторівневою структурою курсів. Вони зменшують когнітивне навантаження на користувачів, дозволяючи швидко зрозуміти своє місцезнаходження в структурі сайту.

### **Контекстуальна перелінковка: створення семантичної мережі**

Внутрішньотекстові посилання в основному контенті представляють найбільш природний і ефективний спосіб перелінковки. Фрази "пошукову оптимізацію", "підвищення швидкості" та "покращення користувацького досвіду" органічно вплетені в текст і ведуть до тематично релевантних сторінок, створюючи логічну мережу взаємопов'язаного контенту.

Використання атрибута `title` в посиланнях надає додаткову інформацію про цільову сторінку, що корисно як для користувачів, так і для пошукових систем. Описи на кшталт "Дізнайтеся про основи SEO" або "Поради щодо

покращення продуктивності" створюють очікування щодо контенту, який користувач знайде після переходу.

Анкорні тексти посилань повинні бути описовими та релевантними до контенту цільової сторінки. Замість загальних фраз типу "дізнайтеся більше" або "читати далі" краще використовувати конкретні терміни, які точно описують тематику пов'язаної сторінки. Це покращує як користувацький досвід, так і SEO-сигнали для пошукових систем.

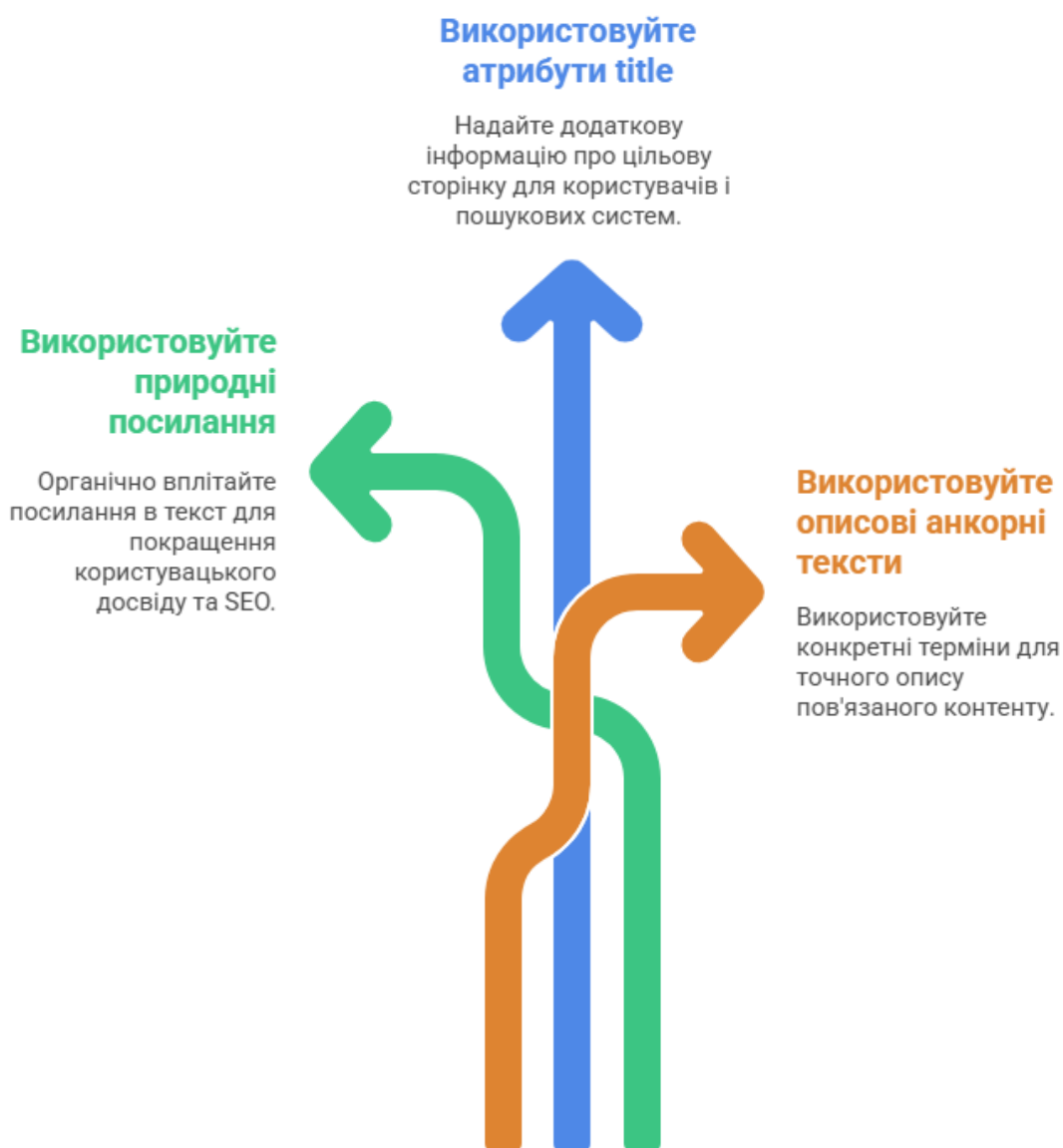


Рисунок 5.4 — Як ефективно використовувати внутрішні посилання?

### **Блок пов'язаних статей: розширення горизонтів користувача**

Секція "Пов'язані статті" в елементі <aside> служить потужним інструментом утримання користувачів на сайті та розповсюдження авторитету

серед тематично близьких сторінок. Розміщення цього блоку в семантичному тезі `<aside>` правильно позначає його як додатковий, але релевантний контент.

Список пов'язаних матеріалів "Семантична розмітка HTML", "Оптимізація CSS для продуктивності" та "JavaScript та продуктивність" створює тематичний кластер навколо основної теми веб-розробки та оптимізації. Така організація допомагає пошуковим системам краще розуміти експертизу сайту в конкретній галузі.

Алгоритми відбору пов'язаних статей можуть базуватися на різних критеріях: семантичній близькості контенту, поведінці користувачів, які переглядали схожі матеріали, або ручному кураторському відборі. Найефективніший підхід часто поєднує автоматизацію з людським контролем якості.

## Процес відбору пов'язаних статей



Рисунок 5.5 — Процес відбору пов'язаних статей

### Стратегічні принципи внутрішньої перелінковки

Розподіл внутрішніх посилань повинен відображати важливість сторінок у загальній структурі сайту. Найважливіші сторінки, які ви хочете просувати в

пошукових результатах, повинні отримувати більше внутрішніх посилань з різних розділів сайту. Це створює сигнал для пошукових систем про пріоритетність цих сторінок.

Глибина переходів від головної сторінки до будь-якої іншої сторінки сайту не повинна перевищувати 3-4 кліки для більшості ресурсів. Це правило "трьох кліків" забезпечує доступність контенту як для користувачів, так і для пошукових роботів, які можуть мати обмеження на глибину сканування.

Регулярний аудит внутрішніх посилань допомагає виявити "сирітські" сторінки, які не мають жодних вхідних внутрішніх посилань, а також битві посилання, які ведуть на неіснуючі сторінки. Такі проблеми можуть негативно впливати на користувацький досвід та SEO-показники.

Анкорні тексти внутрішніх посилань повинні бути різноманітними та природними. Надмірне використання точних ключових слів в якості анкорів може сприйматися пошуковими системами як спроба маніпуляції та призводити до санкцій. Краще використовувати синоніми, пов'язані терміни та природні фрази.

### Стратегія внутрішньої перелінковки

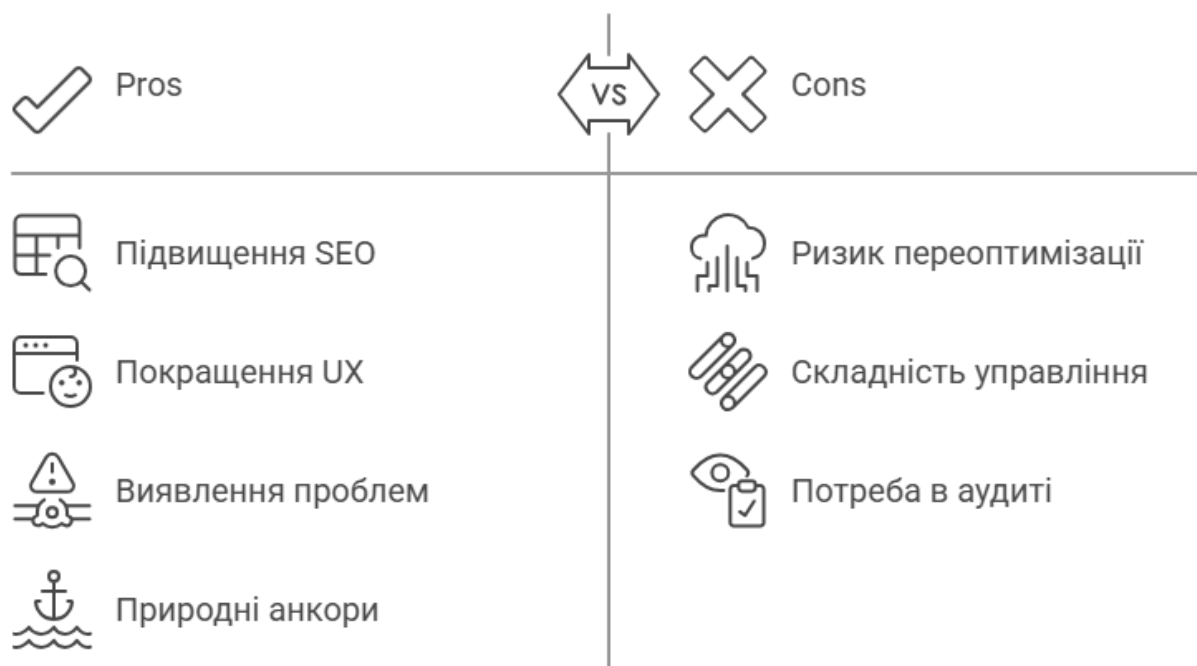


Рисунок 5.6 — Стратегія внутрішньої перелінковки



## Технічні аспекти та найкращі практики

Структура URL повинна відображати ієрархію сайту та бути зрозумілою як для користувачів, так і для пошукових систем. Логічна організація на кшталт `"/category/subcategory/page"` полегшує створення природної системи внутрішніх посилань.

JavaScript-навігація може створювати проблеми для пошукових роботів, тому критично важливі навігаційні елементи повинні бути доступними через звичайні HTML-посилання. Progressive enhancement дозволяє додавати JavaScript-функціональність поверх базової HTML-структури.

Моніторинг кліків по внутрішніх посиланнях через веб-аналітику надає цінні дані про поведінку користувачів і ефективність навігаційної структури. Посилання з низьким CTR можуть потребувати переформулювання або зміни позиціонування на сторінці.

### Структура сайту

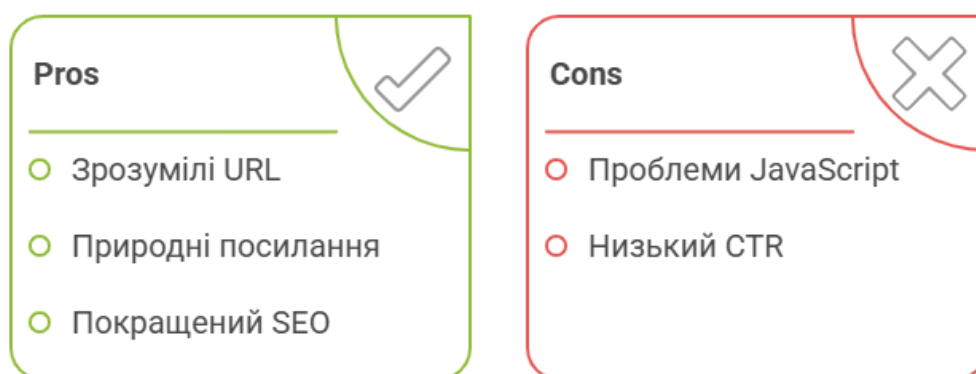


Рисунок 5.7 — Структура сайту

## 5.3 Зовнішня оптимізація

### 5.3.1 Link Building (побудова посилань)

**Link Building:** мистецтво побудови авторитету через якісні зв'язки

Link building являє собою стратегічний процес отримання зворотних посилань з інших веб-ресурсів, який є одним з найпотужніших факторів

ранжування в алгоритмах пошукових систем. Цей процес базується на фундаментальному принципі інтернету: посилання служать рекомендаціями та свідчать про цінність контенту. Пошукові системи інтерпретують якісні зворотні посилання як сигнали довіри і авторитетності, що може суттєво покращити позиції сайту в результатах пошуку.

Сучасний підхід до link building кардинально відрізняється від методів початку 2000-х, коли акцент робився на кількості посилань незалежно від їх якості. Сьогодні пошукові системи використовують складні алгоритми для оцінки релевантності, авторитетності та природності профілю зворотних посилань. Один якісний лінк з авторитетного тематичного ресурсу може мати більшу цінність, ніж сотні посилань з низькоякісних сайтів.

```
<!-- Приклад правильного оформлення зовнішніх посилань
-->
<article>
  <p>Згідно з дослідженням
  <a href="https://authoritative-site.com/study"
    target="_blank"
    rel="nofollow noopener"
    title="Дослідження впливу швидкості сайту на
конверсії">
    авторитетного джерела
  </a>, підвищення швидкості завантаження сторінки на
1 секунду може збільшити конверсії до 7%.</p>

  <!-- Внутрішнє посилання (без nofollow) -->
  <p>Детальніше про методи оптимізації читайте в
нашому
  <a href="/guide-optimization" title="Повний гайд з
оптимізації сайту">
    комплексному гайді
```

```

    </a>.</p>
</article>

<!-- Приклад структурованих даних для статті -->
<script type="application/ld+json">
{
  "@context": "https://schema.org",
  "@type": "Article",
  "headline": "Побудова якісних зворотних посилань",
  "author": {
    "@type": "Organization",
    "name": "Університет технологій"
  },
  "publisher": {
    "@type": "Organization",
    "name": "Університет технологій",
    "logo": {
      "@type": "ImageObject",
      "url": "https://university.com/logo.png"
    }
  },
  "datePublished": "2024-01-15",
  "citation": [
    {
      "@type": "CreativeWork",
      "url": "https://authoritative-site.com/study",
      "name": "Дослідження впливу швидкості сайту"
    }
  ]
}

```

</script>

### **Етичні стратегії отримання якісних посилань**

Найефективнішою довгостроковою стратегією link building є створення контенту, який природним чином заслуговує на посилання. Це включає оригінальні дослідження з унікальними даними, які можуть цитувати журналісти та блогери. Наприклад, дослідження користувацької поведінки на вашому сайті може стати цінним ресурсом для інших фахівців у галузі.

Створення корисних інструментів та калькуляторів також генерує природні посилання, оскільки інші веб-майстри посилаються на них як на корисні ресурси для своєї аудиторії. Безкоштовний інструмент для перевірки швидкості сайту або SEO-аудиту може залучити посилання з десятків релевантних ресурсів.

Гостьові публікації на авторитетних сайтах у вашій тематиці дозволяють не лише отримати якісне посилання, але й продемонструвати експертизу ширшій аудиторії. Важливо, щоб такі статті надавали реальну цінність читачам, а не слугували лише для розміщення посилань.

### **Технічні аспекти оформлення посилань**

Представлений HTML-код демонструє правильне оформлення зовнішніх та внутрішніх посилань з урахуванням сучасних SEO-вимог. Атрибут `rel="nofollow noopener"` на зовнішньому посиланні виконує кілька важливих функцій: `nofollow` повідомляє пошуковим системам не передавати авторитет через це посилання, а `noopener` забезпечує безпеку, запобігаючи доступу цільової сторінки до об'єкта `window.opener`.

Використання атрибута `target="_blank"` змушує посилання відкриватися в новій вкладці, що дозволяє користувачам залишатися на вашому сайті навіть після переходу за зовнішнім посиланням. Це особливо важливо для утримання аудиторії та зменшення показника відмов.

Атрибут `title` надає додатковий контекст про цільову сторінку, що покращує доступність для користувачів допоміжних технологій та може

використовуватися браузером для відображення підказок при наведенні курсору.

Внутрішні посилання оформлюються без атрибута `nofollow`, що дозволяє вільно передавати авторитет між сторінками вашого власного сайту. Це критично важливо для ефективної внутрішньої перелінковки та розподілу PageRank.

### Структуровані дані для цитувань

JSON-LD розмітка в прикладі демонструє, як правильно оформлювати структуровані дані для статей, що містять цитування зовнішніх джерел. Поле `citation` дозволяє явно вказати джерела, на які посилається стаття, що може покращити довіру пошукових систем до контенту.

Інформація про автора та публікатора через типи `Organization` замість `Person` підходить для корпоративних та освітніх ресурсів, де контент створюється від імені інституції. Логотип організації в структурованих даних може відображатися в результатах пошуку, покращуючи розпізнаваність бренду.

Дата публікації `datePublished` допомагає пошуковим системам оцінити актуальність контенту, що особливо важливо для тем, які швидко змінюються, таких як технології або законодавство.

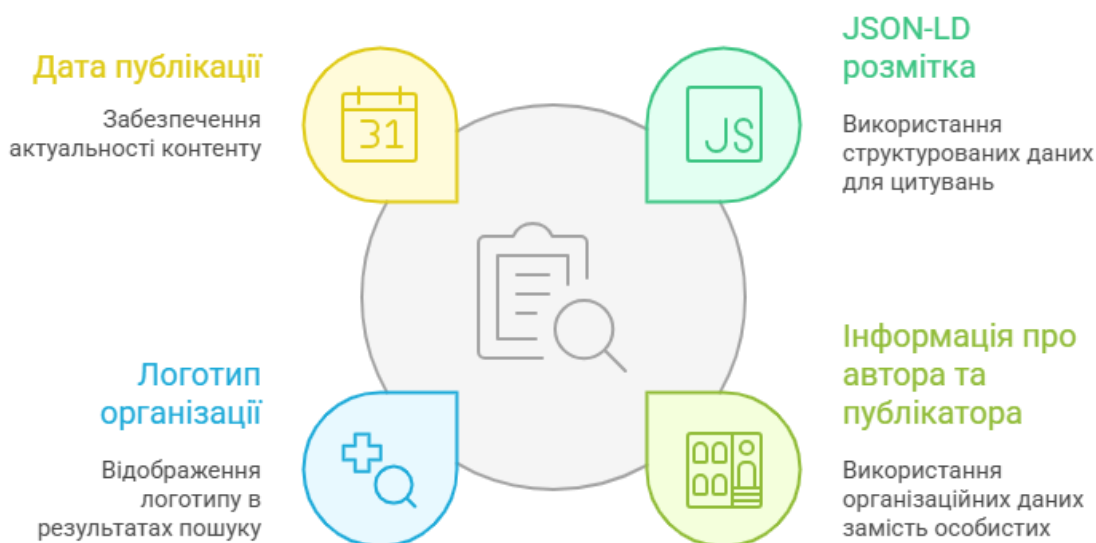


Рисунок 5.8 — Покращення довіри до контенту та розпізнаваності бренду



```

        'event_label': this.href,
        'value': 1
    });
}

// Додавання затримки для завантаження
аналітики
if (!this.target || this.target ===
'_self') {
    e.preventDefault();
    setTimeout(() => {
        window.location.href = this.href;
    }, 100);
}
});
});
});

```

Регулярний аудит профілю зворотних посилань за допомогою інструментів типу Ahrefs, SEMrush або безкоштовного Google Search Console допомагає відстежувати зростання авторитету сайту та виявляти потенційні проблеми. Різкий приріст низькоякісних посилань може сигналізувати про негативну SEO-атаку конкурентів.

Аналіз анкорних текстів зворотних посилань допомагає забезпечити їх природність. Надмірна концентрація точних комерційних анкорів може викликати підозри пошукових систем і призвести до санкцій. Здоровий профіль посилань включає різноманітні анкори: брендові, URL, загальні фрази та природні варіації ключових слів.

### **Етичні принципи та довгострокова стратегія**

Сучасний link building повинен базуватися на принципах створення справжньої цінності для користувачів та веб-спільноти. Покупка посилань,

участь у схемах обміну лінками або використання приватних блог-мереж може дати короткострокові результати, але несе високі ризики санкцій від пошукових систем.

Фокус на якості замість кількості означає концентрацію зусиль на отриманні посилань з дійсно авторитетних та релевантних ресурсів. Одне посилання з популярного галузевого видання може принести більше трафіку та авторитету, ніж десятки посилань з малозначних блогів.

Довгострокова стратегія link building включає постійне створення контенту, гідного цитування, налагодження відносин з впливовими особами в галузі та участь у професійних спільнотах. Такий підхід забезпечує стійкий ріст авторитету сайту та стійкість до змін в алгоритмах пошукових систем.

#### Стратегії побудови посилань

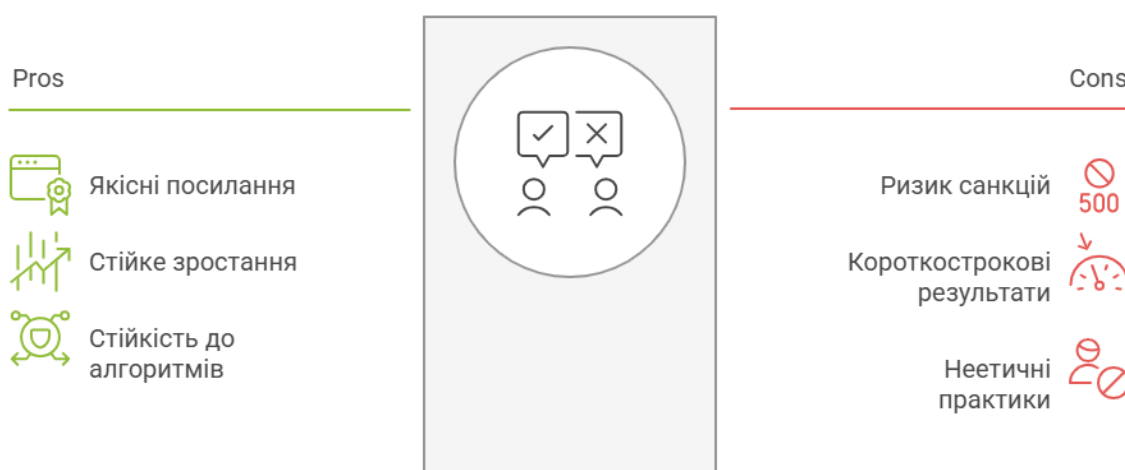


Рисунок 5.9 — Стратегії побудови посилань

### 5.3.2 Соціальні сигнали

Соціальні сигнали представляють собою метрики активності користувачів у соціальних мережах відносно контенту вашого веб-сайту, включаючи лайки, репости, коментарі та згадування бренду. Хоча представники пошукових систем офіційно заперечують прямий вплив соціальних сигналів на ранжування, існує безліч непрямих механізмів, через які соціальна активність може покращувати SEO-показники.



Соціальні медіа служать потужними каналами розповсюдження контенту, які можуть генерувати значний реферальний трафік, збільшувати впізнаваність бренду та стимулювати природні згадування і посилання з інших веб-ресурсів. Вірусний контент у соціальних мережах часто привертає увагу журналістів та блогерів, що може призвести до отримання якісних зворотних посилань та додаткового медіа-покриття.

Дослідження показують кореляцію між високою соціальною активністю та кращими позиціями в пошукових результатах. Хоча це може бути результатом того, що якісний контент одночасно добре ранжується і активно поширюється в соціальних мережах, а не прямої причинно-наслідкової залежності.

```
<!-- Open Graph мета-теги для соціальних мереж -->
<head>
    <meta property="og:type" content="article">
    <meta property="og:title" content="Оптимізація
сайту - повний гайд">
    <meta property="og:description" content="Дізнайтеся
про ефективні методи оптимізації сайту для покращення
продуктивності та SEO">
    <meta property="og:url"
content="https://university.com/optimization-guide">
    <meta property="og:image"
content="https://university.com/images/optimization-
guide-social.jpg">
    <meta property="og:image:width" content="1200">
    <meta property="og:image:height" content="630">
    <meta property="og:locale" content="uk_UA">
    <meta property="og:site_name" content="Університет
технологій">
```

```

<!-- Twitter Card мета-теги -->
<meta name="twitter:card"
content="summary_large_image">
<meta name="twitter:title" content="Оптимізація
сайту - повний гайд">
<meta name="twitter:description"
content="Дізнайтеся про ефективні методи оптимізації
сайту">
<meta name="twitter:image"
content="https://university.com/images/optimization-
guide-twitter.jpg">

<!-- Додаткові мета-теги для інших платформ -->
<meta property="article:author"
content="https://university.com/author/teacher">
<meta property="article:published_time"
content="2024-01-15T10:00:00+00:00">
<meta property="article:tag" content="SEO">
<meta property="article:tag" content="Оптимізація">
<meta property="article:tag" content="Веб-
розробка">
</head>

```

### **Open Graph та Twitter Cards: оптимізація для соціального поширення**

Open Graph протокол, розроблений Facebook, стандартизував спосіб представлення веб-контенту в соціальних мережах. Правильне налаштування Open Graph мета-тегів забезпечує контрольоване та привабливе відображення ваших сторінок при поширенні в соціальних мережах.

Тег `og:type` зі значенням `"article"` повідомляє соціальним платформам, що контент є статтею, що може активувати додаткові функції відображення, такі

як показ дати публікації або інформації про автора. Для різних типів контенту існують специфічні типи: "website" для головних сторінок, "product" для товарів у інтернет-магазинах, "video" для відеоматеріалів.

Заголовок `og:title` та опис `og:description` повинні бути оптимізованими для соціальних мереж і можуть відрізнятися від SEO-заголовків. Соціальні мережі мають інші обмеження по довжині: Facebook відображає близько 100 символів заголовка та 300 символів опису, тому текст повинен бути лаконічним та привабливим.

Зображення `og:image` є критично важливим елементом, оскільки візуальний контент отримує значно більше уваги в соціальних мережах. Рекомендований розмір 1200x630 пікселів забезпечує оптимальне відображення на всіх платформах. Зображення повинно бути релевантним до контенту та включати мінімальну кількість тексту, щоб залишатися читабельним навіть у зменшеному вигляді.

Локалізація через `og:locale` допомагає соціальним платформам правильно формувати дати, числа та інші локалізовані елементи. Значення "uk\_UA" вказує на українську мову та українські регіональні налаштування.

### **Twitter Cards: специфічна оптимізація для Twitter**

Twitter використовує власну систему мета-тегів для оптимального відображення контенту. Тип карти "summary\_large\_image" забезпечує показ великого зображення, що значно підвищує привабливість посту в стрічці Twitter.

Twitter накладає більш жорсткі обмеження на довжину тексту: заголовок обмежений 70 символами, а опис - 200. Це вимагає ще більш лаконічного формулювання, яке має миттєво привертати увагу користувачів.

Додаткові Article-теги надають детальну інформацію про контент для платформ, які підтримують розширені мета-дані. Інформація про автора через `article:author` може відображатися в деяких соціальних мережах, підвищуючи довіру до контенту.

Теги `article:tag` допомагають категоризувати контент і можуть використовуватися для створення тематичних кластерів на соціальних платформах. Це особливо корисно для блогів і новинних сайтів з різноманітним контентом.

### Оптимізація кнопок соціального поширення

```
<!-- Оптимізовані кнопки соціального поширення -->
<div class="social-share" role="group" aria-
label="Поділитися в соціальних мережах">
    <button class="share-button facebook"
        onclick="shareOnFacebook()"
        aria-label="Поділитися у Facebook">
        <svg width="24" height="24" viewBox="0 0 24 24"
fill="currentColor">
            <path d="M24 12.073c0-6.627-5.373-12-12-12-
12s-12 5.373-12 12c0 5.99 4.388 10.954 10.125 11.854v-
8.385H7.078v-3.47h3.047V9.43c0-3.007 1.792-4.669 4.533-
4.669 1.312 0 2.686.235 2.686.235v2.953H15.83c-1.491 0-
1.956.925-1.956 1.874v2.25h3.328l-.532 3.47h-
2.796v8.385C19.612 23.027 24 18.062 24 12.073z"/>
        </svg>
        Facebook
    </button>

    <button class="share-button twitter"
        onclick="shareOnTwitter()"
        aria-label="Поділитися у Twitter">
        <svg width="24" height="24" viewBox="0 0 24 24"
fill="currentColor">
            <path d="M23.953 4.57a10 10 0 01-2.825.775
4.958 4.958 0 002.163-2.723c-.951.555-2.005.959-3.127
```

```

1.184a4.92 4.92 0 00-8.384 4.482C7.69 8.095 4.067 6.13
1.64 3.162a4.822 4.822 0 00-.666 2.475c0 1.71.87 3.213
2.188 4.096a4.904 4.904 0 01-2.228-.616v.06a4.923 4.923
0 003.946 4.827 4.996 4.996 0 01-2.212.085 4.936 4.936
0 004.604 3.417 9.867 9.867 0 01-6.102 2.105c-.39 0-
.779-.023-1.17-.067a13.995 13.995 0 007.557 2.209c9.053
0 13.998-7.496 13.998-13.985 0-.21 0-.42-.015-.63A9.935
9.935 0 0024 4.59z"/>
    </svg>
    Twitter
</button>

<button class="share-button linkedin"
    onclick="shareOnLinkedIn()"
    aria-label="Поділитися у LinkedIn">
    <svg width="24" height="24" viewBox="0 0 24 24"
fill="currentColor">
        <path d="M20.447 20.452h-3.554v-5.569c0-
1.328-.027-3.037-1.852-3.037-1.853 0-2.136 1.445-2.136
2.939v5.667H9.351V9h3.414v1.561h.046c.477-.9 1.637-1.85
3.37-1.85 3.601 0 4.267 2.37 4.267 5.455v6.286zM5.337
7.433c-1.144 0-2.063-.926-2.063-2.065 0-1.138.92-2.063
2.063-2.063 1.14 0 2.064.925 2.064 2.063 0 1.139-.925
2.065-2.064 2.065zm1.782
13.019H3.555V9h3.564v11.452zM22.225 0H1.771C.792 0 0
.774 0 1.729v20.542C0 23.227.792 24 1.771
24h20.451C23.2 24 24 23.227 24 22.271V1.729C24 .774
23.2 0 22.222 0h.003z"/>
    </svg>
    LinkedIn

```

```
        </button>
    </div>

    <script>
    function shareOnFacebook() {
        const url =
    encodeURIComponent(window.location.href);
        const title = encodeURIComponent(document.title);

    window.open(`https://www.facebook.com/sharer/sharer.php
    ?u=${url}`, 'facebook-share', 'width=580,height=296');
    }

    function shareOnTwitter() {
        const url =
    encodeURIComponent(window.location.href);
        const title = encodeURIComponent(document.title);
        const text =
    encodeURIComponent(document.querySelector('meta[name="d
    escription"]').content);

    window.open(`https://twitter.com/intent/tweet?url=${url
    }&text=${title}&via=university_tech`, 'twitter-share',
    'width=550,height=235');
    }

    function shareOnLinkedIn() {
        const url =
    encodeURIComponent(window.location.href);
        const title = encodeURIComponent(document.title);
```

```

    const summary =
encodeURIComponent(document.querySelector('meta[name="description"]').content);

window.open(`https://www.linkedin.com/sharing/share-offsite/?url=${url}`, 'linkedin-share',
'width=490,height=530');
}
</script>

```

Представлені кнопки соціального поширення демонструють сучасний підхід до інтеграції соціальних функцій без використання важких зовнішніх SDK, які можуть сповільнювати завантаження сторінки. Використання inline SVG-іконок замість растрових зображень забезпечує чіткість відображення на всіх типах екранів і знижує кількість HTTP-запитів.

Атрибути доступності `aria-label` та `role="group"` роблять кнопки зручними для користувачів скрін-рідерів. Описовий текст на кшталт "Поділитися у Facebook" надає чіткий контекст про функціональність кнопки.

Функція `shareOnFacebook()` відкриває нативний діалог поширення Facebook у спливаючому вікні з оптимальними розмірами. Передача URL через `encodeURIComponent()` забезпечує коректну обробку спеціальних символів у адресах сторінок.

Twitter-функція включає додатковий параметр `via` з іменем акаунту організації, що може збільшити кількість підписників та створити зв'язок між поширеним контентом та брендом. Параметр `text` автоматично заповнює текст твіта описом сторінки.

LinkedIn використовує найпростіший API поширення, який автоматично витягає всю необхідну інформацію з Open Graph мета-тегів сторінки. Це зменшує складність коду та забезпечує консистентність представлення контенту.

### **Стратегічні аспекти використання соціальних сигналів**

Ефективна стратегія соціальних медіа для SEO включає створення контенту, оптимізованого для конкретних платформ. Візуальний контент краще працює в Instagram та Pinterest, довгі аналітичні матеріали - у LinkedIn, а швидкі оновлення та новини - у Twitter.

Тайминг публікацій критично важливий для максимального охоплення. Аналіз аудиторії через інструменти соціальних платформ допомагає визначити оптимальні години для публікації контенту у кожній мережі.

Взаємодія з аудиторією через коментарі та приватні повідомлення підвищує рівень залученості та може стимулювати додаткове поширення контенту. Швидкі та змістовні відповіді на запитання користувачів демонструють професіоналізм і турботу про аудиторію.

### **Моніторинг та аналітика соціальних сигналів**

Відстеження соціальних згадувань бренду через інструменти типу Google Alerts, Mention або Hootsuite допомагає виявляти можливості для взаємодії та управління репутацією. Швидка реакція на негативні відгуки може запобігти ескалації проблем.

Аналіз трафіку з соціальних мереж через Google Analytics показує, які платформи генерують найбільш залучену аудиторію. Користувачі з різних соціальних мереж можуть мати різну поведінку на сайті: Twitter зазвичай дає високий показник відмов, але швидко поширює новини, тоді як LinkedIn генерує більш цільовий трафік для B2B-контенту.

Корелятний аналіз між соціальною активністю та органічним трафіком може виявити закономірності, які допоможуть оптимізувати контентну стратегію. Контент, який добре працює в соціальних мережах, часто отримує кращі позиції в пошукових результатах через збільшений користувацький інтерес і природні посилання.





Рисунок 5.10 — Стратегії соціального моніторингу та аналітики

### 5.3.3 Контент-маркетинг

Контент-маркетинг представляє собою стратегічний підхід до зовнішньої оптимізації, який кардинально відрізняється від традиційної реклами тим, що замість прямого просування продуктів чи послуг зосереджується на створенні дійсно корисного та цікавого контенту, який люди хочуть читати, ділитися ним та посилатися на нього природним чином. Ця довгострокова стратегія будується на розумінні того, що сучасні користувачі втомилися від нав'язливої реклами та шукають справжню цінність у контенті, який вони споживають в інтернеті.

Ефективність контент-маркетингу полягає в його здатності одночасно вирішувати декілька важливих завдань: він покращує позиції веб-сайту в результатах пошуку завдяки створенню релевантного та унікального контенту,

будує довіру та лояльність аудиторії через надання корисної інформації, а також стимулює природне отримання зворотних посилань від інших ресурсів, які хочуть посилатися на якісні матеріали. Більше того, такий підхід дозволяє позиціонувати організацію як експерта в своїй галузі, що особливо важливо для навчальних закладів та інших установ, які працюють у сфері знань.

### **Створення linkable assets (ресурсів, що привертають посилання)**

Linkable assets - це особливий тип контенту, який володіє такою високою цінністю та унікальністю, що інші веб-сайти природним чином хочуть на нього посилатися, використовуючи його як авторитетне джерело інформації або корисний ресурс для своїх читачів. На відміну від звичайного контенту, який може бути цікавим для читання, linkable assets створюються з конкретною метою стати "магнітом для посилань" - тобто контентом, який настільки корисний або унікальний, що блогери, журналісти, дослідники та інші контент-мейкери захочуть його процитувати або рекомендувати своїй аудиторії.

Найефективніші типи linkable assets включають оригінальні дослідження з унікальними даними, які ніде більше не публікувалися, інтерактивні онлайн-інструменти, що допомагають користувачам вирішувати конкретні завдання, всеохоплюючі гайди та посібники, які стають довідковими матеріалами в певній галузі, а також візуальний контент на кшталт інфографік, які представляють складну інформацію в легкому для сприйняття форматі. Кожен з цих типів контенту має свої особливості та переваги, але всі вони об'єднані спільною характеристикою - вони надають справжню цінність користувачам та заохочують інших створювати посилання на них.

### **Приклад: Інтерактивний інструмент для університету**

Розглянемо конкретний приклад створення linkable asset для університету - інтерактивний інструмент аналізу SEO-показників веб-сайтів. Такий інструмент може стати надзвичайно популярним серед веб-розробників, маркетологів та власників сайтів, оскільки він надає безкоштовну можливість швидко оцінити основні параметри оптимізації будь-якого веб-ресурсу.

```

<!-- Простіший приклад калькулятора для студентів -->
<div class="study-calculator">
    <h3>Калькулятор навчального навантаження</h3>
    <p>Дізнайтеся, скільки годин на тиждень вам
    потрібно вчитися</p>

    <form id="studyForm">
        <label>Кількість предметів:
            <input type="number" id="subjects" min="1"
max="10" value="5">
        </label>

        <label>Складність (1-легко, 5-дуже важко):
            <input type="range" id="difficulty" min="1"
max="5" value="3">
        </label>

        <button type="submit">Розрахувати</button>
    </form>

    <div id="result" style="display:none;">
        <h4>Рекомендована кількість годин: <span
id="hours">0</span> год/тиждень</h4>
        <p id="advice"></p>
    </div>
</div>

<script>
document.getElementById('studyForm').addEventListener('
submit', function(e) {

```

```

    e.preventDefault();

    const subjects =
parseInt(document.getElementById('subjects').value);
    const difficulty =
parseInt(document.getElementById('difficulty').value);

    // Простий розрахунок: базових 2 години на предмет
+ коефіцієнт складності
    const hours = subjects * (2 + difficulty * 0.5);

    document.getElementById('hours').textContent =
Math.round(hours);

    document.getElementById('advice').textContent =
        hours > 25 ? 'Високе навантаження - плануйте
час ретельно!' :
        'Помірне навантаження - у вас є час на хобі';

    document.getElementById('result').style.display =
'block';
});
</script>

```

Переваги такого інструменту полягають у тому, що він вирішує конкретну проблему користувачів (планування навчального часу), є інтерактивним та надає персоналізовані результати, а також може бути легко вбудований на інші сайти або блоги, що природним чином генерує зворотні посилання на ваш ресурс.

```

<!-- Приклад структури для інтерактивного інструменту -
->
<div class="interactive-tool" id="seo-analyzer">

```

```

<header class="tool-header">
    <h2>Інструмент аналізу SEO-показників</h2>
    <p class="tool-description">
        Безкоштовний онлайн-інструмент для швидкого
        аналізу основних SEO-метрик вашого сайту
    </p>
</header>

<form class="tool-form" id="analyzerForm">
    <div class="form-group">
        <label for="websiteUrl">URL веб-сайту для
        аналізу:</label>
        <input type="url"
            id="websiteUrl"
            name="websiteUrl"
            placeholder="https://example.com"
            required
            aria-describedby="url-help">
        <small id="url-help">Введіть повний URL
        включаючи https://</small>
    </div>

    <button type="submit" class="analyze-button">
        <span class="button-text">Проаналізувати
        сайт</span>
        <span class="loading-spinner"
        hidden>Аналізуємо...</span>
    </button>
</form>

```

```

    <div class="results-container" id="analysisResults"
hidden>
        <h3>Результати аналізу</h3>
        <div class="metrics-grid">
            <div class="metric-card">
                <h4>Швидкість завантаження</h4>
                <div class="metric-value"
id="speedScore">-</div>
                <div class="metric-description">Page
Speed Score</div>
            </div>

            <div class="metric-card">
                <h4>SEO-оптимізація</h4>
                <div class="metric-value"
id="seoScore">-</div>
                <div class="metric-description">SEO
Score</div>
            </div>

            <div class="metric-card">
                <h4>Мобільна адаптивність</h4>
                <div class="metric-value"
id="mobileScore">-</div>
                <div class="metric-description">Mobile
Friendly Score</div>
            </div>
        </div>
    </div>

```

```

        <div class="recommendations"
id="recommendations">
            <h4>Рекомендації для покращення:</h4>
            <ul id="recommendationsList"></ul>
        </div>

        <div class="share-results">
            <p>Поділіться результатами:</p>
            <button onclick="shareResults()"
class="share-button">
                Отримати посилання на результати
            </button>
        </div>
    </div>
</div>

<script>
document.getElementById('analyzerForm').addEventListener(
    'submit', async function(e) {
        e.preventDefault();

        const url =
document.getElementById('websiteUrl').value;
        const loadingSpinner =
document.querySelector('.loading-spinner');
        const buttonText = document.querySelector('.button-
text');
        const resultsContainer =
document.getElementById('analysisResults');

```

```

// Показати індикатор завантаження
loadingSpinner.hidden = false;
buttonText.textContent = 'Аналізуємо...';

try {
    // Симуляція API-запиту (у реальному додатку
    тут буде виклик до серверного API)
    await simulateAnalysis(url);

    // Показати результати
    resultsContainer.hidden = false;
    resultsContainer.scrollToView({ behavior:
'smooth' });

    } catch (error) {
        console.error('Помилка аналізу:', error);
        alert('Виникла помилка під час аналізу.
Спробуйте ще раз.');
```

```

    } finally {
        // Сховати індикатор завантаження
        loadingSpinner.hidden = true;
        buttonText.textContent = 'Проаналізувати сайт';
    }
});

async function simulateAnalysis(url) {
    // Симуляція затримки API-запиту
    await new Promise(resolve => setTimeout(resolve,
2000));

```



```

// Симуляція результатів аналізу
const mockResults = {
    speedScore: Math.floor(Math.random() * 40) +
60,
    seoScore: Math.floor(Math.random() * 30) + 70,
    mobileScore: Math.floor(Math.random() * 20) +
80,
    recommendations: [
        'Оптимізуйте розмір зображень',
        'Увімкніть компресію gzip',
        'Додайте мета-описи до всіх сторінок',
        'Покращте внутрішню перелінковку',
        'Оптимізуйте заголовки H1-H6'
    ]
};

// Відображення результатів
document.getElementById('speedScore').textContent =
mockResults.speedScore;
document.getElementById('seoScore').textContent =
mockResults.seoScore;
document.getElementById('mobileScore').textContent
= mockResults.mobileScore;

const recommendationsList =
document.getElementById('recommendationsList');
recommendationsList.innerHTML = '';
mockResults.recommendations.forEach(rec => {
    const li = document.createElement('li');
    li.textContent = rec;

```

```

        recommendationsList.appendChild(li);
    });
}

function shareResults() {
    const url = window.location.href + '#results';
    navigator.clipboard.writeText(url).then(() => {
        alert('Посилання на результати скопійовано!');
    });
}
</script>

```

### **Гостьові публікації та експертна співпраця**

Стратегія гостьових публікацій являє собою взаємовигідну форму співпраці, за якої фахівці пишуть статті для інших веб-ресурсів у своїй галузі, отримуючи натомість можливість представити себе як експерта, розширити охоплення своєї аудиторії та отримати якісні зворотні посилання з авторитетних джерел. Цей підхід особливо ефективний для академічних закладів та освітніх організацій, оскільки викладачі та дослідники природним чином володіють експертними знаннями, якими вони можуть ділитися з професійною спільнотою.

Успішна гостьова публікація вимагає ретельної підготовки та стратегічного підходу до вибору платформ для співпраці, створення цінного контенту, який буде корисний для аудиторії приймаючого ресурсу, а також правильного оформлення авторської інформації, яка має містити посилання на ваш основний веб-сайт та короткий опис вашої експертизи. Важливо розуміти, що гостьові публікації - це не лише спосіб отримання посилань, але й можливість побудувати довгострокові професійні відносини та зміцнити репутацію в академічному або професійному середовищі.

### **Приклад оформлення авторського блоку:**

```

<!-- Простий авторський блок для викладача -->

```

```

<div class="author-info">
  
  <div class="author-text">
    <h4>Іван Петренко</h4>
    <p>Кандидат технічних наук, викладач веб-
технологій</p>
    <p>Автор 15+ наукових статей з оптимізації
сайтів</p>
    <a href="https://university.edu/ivan-
petrenko">Більше статей автора</a>
  </div>
</div>

```

При написанні гостьових статей важливо дотримуватися високих стандартів якості контенту, оскільки ваша репутація як експерта залежить від того, наскільки корисною та професійною виявиться ваша публікація для читачів приймаючого ресурсу. Крім того, необхідно враховувати специфіку аудиторії та стиль подачі матеріалу, характерний для конкретного веб-сайту чи блогу, де ви плануєте опублікуватися.

#### **Ефективні теми для гостьових публікацій в сфері веб-технологій:**

- "5 помилок у SEO, які роблять початківці"
- "Як швидкість сайту впливає на користувацький досвід"
- "Мобільна оптимізація: тренди 2024 року"
- "Доступність веб-сайтів: чому це важливо"

#### **Стратегія пошуку платформ для публікацій:**

```

<!-- Приклад контактної форми для пропозиції співпраці
-->
<div class="collaboration-pitch">
  <h4>Пропозиція співпраці</h4>

```

```

    <p><strong>Тема статті:</strong> "Оптимізація
сайтів для навчальних закладів"</p>
    <p><strong>Короткий опис:</strong> Практичні поради
щодо покращення веб-присутності університетів та
шкіл</p>
    <p><strong>Унікальність:</strong> Базується на 5-
річному досвіді роботи з освітніми проектами</p>
    <p><strong>Обсяг:</strong> 1500-2000 слів +
практичні приклади</p>
    <p><strong>Автор:</strong> Викладач з університету,
кандидат наук</p>
</div>

```

Такий підхід демонструє професіоналізм та серйозність ваших намірів, що значно підвищує шанси на успішну співпрацю з якісними ресурсами у вашій галузі.

```

<!-- Приклад біо-блоку для гостьових публікацій -->
<div class="author-bio guest-post-bio">
    <div class="author-avatar">
        
    </div>

    <div class="author-info">
        <h4 class="author-name">Викладач технологій
інтернет</h4>
        <p class="author-title">Провідний експерт з
веб-оптимізації та SEO</p>
    </div>
</div>

```

```

    <p class="author-description">
        Понад 10 років досвіду у веб-розробці та
        оптимізації сайтів.
        Викладає курси з технологій інтернет в
        університеті та консультує
        компанії з питань цифрової присутності.
    </p>

    <div class="author-links">
        <a href="https://university.com/teacher-
        profile"
            rel="author"
            class="author-link">
            Профіль викладача
        </a>
        <a
        href="https://university.com/optimization-course"
            class="author-link">
            Курс оптимізації сайтів
        </a>
    </div>
</div>
</div>

<!-- Структуровані дані для автора -->
<script type="application/ld+json">
{
    "@context": "https://schema.org",
    "@type": "Person",
    "name": "Викладач технологій інтернет",

```

```

"jobTitle": "Провідний експерт з веб-оптимізації",
"worksFor": {
  "@type": "Organization",
  "name": "Університет технологій"
},
"url": "https://university.com/teacher-profile",
"sameAs": [
  "https://linkedin.com/in/teacher-profile",
  "https://twitter.com/teacher_tech"
],
"knowsAbout": [
  "SEO оптимізація",
  "Веб-розробка",
  "Продуктивність сайтів",
  "Технології інтернет"
]
}
</script>

```

#### 5.3.4 Локальна SEO (для місцевого бізнесу)

Локальна SEO-оптимізація є критично важливою для бізнесів, які обслуговують клієнтів у конкретних географічних регіонах. Цей тип оптимізації включає роботу з Google My Business, локальними цитуваннями, відгуками клієнтів та геотаргетингом.

Локальна оптимізація вимагає особливої уваги до географічних сигналів, структурованих даних про місцезнаходження та локального контенту.

```

<!-- Локальна SEO розмітка для університету -->
<div class="university-contact" itemscope
itemtype="https://schema.org/EducationalOrganization">
  <header>

```

```

    <h2 itemprop="name">Університет технологій</h2>
    <p itemprop="description">
        Провідний навчальний заклад України у сфері
інформаційних технологій та веб-розробки
    </p>
</header>

<div class="contact-info">
    <div class="address" itemprop="address"
itemscope itemtype="https://schema.org/PostalAddress">
        <h3>Адреса університету:</h3>
        <p>
            <span itemprop="streetAddress">вул.
Технологічна, 15</span><br>
            <span
itemprop="addressLocality">Київ</span>,
            <span
itemprop="postalCode">03056</span><br>
            <span
itemprop="addressCountry">Україна</span>
        </p>
    </div>

    <div class="contact-details">
        <p><strong>Телефон:</strong>
            <a href="tel:+380441234567"
itemprop="telephone">+38 (044) 123-45-67</a>
        </p>
        <p><strong>Email:</strong>

```

```

        <a href="mailto:info@university.com"
itemprop="email">info@university.com</a>
    </p>
    <p><strong>Веб-сайт:</strong>
        <a href="https://university.com"
itemprop="url">university.com</a>
    </p>
</div>

<div class="operating-hours">
    <h3>Години роботи:</h3>
    <div itemprop="openingHours" content="Mo-Fr
08:00-18:00">
        Понеділок - П'ятниця: 8:00 - 18:00
    </div>
    <div itemprop="openingHours" content="Sa
09:00-15:00">
        Субота: 9:00 - 15:00
    </div>
    <div>Неділя: Вихідний</div>
</div>

<!-- Географічні координати -->
<div class="geo-coordinates" style="display:
none;">
    <span itemprop="geo" itemscope
itemtype="https://schema.org/GeoCoordinates">
        <meta itemprop="latitude"
content="50.4501">

```



```

        <meta itemprop="longitude"
content="30.5234">
        </span>
    </div>
</div>

<!-- Вбудована карта -->
<div class="map-container">
    <iframe
src="https://www.google.com/maps/embed?pb=!1m18!1m12!1m
3!1d2540.8344!2d30.5234!3d50.4501!2m3!1f0!2f0!3f0!3m2!1
i1024!2i768!4f13.1!3m3!1m2!1s0x0%3A0x0!2zNTDCsDI3JzAwLj
QiTiAzMMKwMzEnMjQuMiJF!5e0!3m2!1suk!2sua!4v1234567890"
        width="100%"
        height="300"
        style="border:0;"
        allowfullscreen=""
        loading="lazy"
        referrerpolicy="no-referrer-when-
downgrade"
        title="Карта розташування Університету
технологій">
    </iframe>
</div>
</div>

<!-- JSON-LD структуровані дані для локального бізнесу
-->
<script type="application/ld+json">
{

```

```
"@context": "https://schema.org",
"@type": "EducationalOrganization",
"name": "Університет технологій",
"description": "Провідний навчальний заклад України у
сфері інформаційних технологій та веб-розробки",
"url": "https://university.com",
"telephone": "+380441234567",
"email": "info@university.com",
"address": {
  "@type": "PostalAddress",
  "streetAddress": "вул. Технологічна, 15",
  "addressLocality": "Київ",
  "postalCode": "03056",
  "addressCountry": "UA"
},
"geo": {
  "@type": "GeoCoordinates",
  "latitude": "50.4501",
  "longitude": "30.5234"
},
"openingHoursSpecification": [
  {
    "@type": "OpeningHoursSpecification",
    "dayOfWeek": [
      "Monday",
      "Tuesday",
      "Wednesday",
      "Thursday",
      "Friday"
    ]
  }
],
```

```

        "opens": "08:00",
        "closes": "18:00"
    },
    {
        "@type": "OpeningHoursSpecification",
        "dayOfWeek": "Saturday",
        "opens": "09:00",
        "closes": "15:00"
    }
],
"sameAs": [
    "https://facebook.com/university.tech",
    "https://linkedin.com/school/university-tech"
]
}
</script>

```

## Висновки

Оптимізація веб-сайтів є ключовим чинником успішності онлайн-проектів у сучасному цифровому середовищі. Вона охоплює широкий спектр заходів – від технічної оптимізації коду, серверних налаштувань і швидкості завантаження до SEO-стратегії, покращення користувацького досвіду (UX/UI), внутрішньої перелінковки та побудови якісного зовнішнього посилального профілю.

Застосування сучасних інструментів і методів, таких як структурована розмітка Schema.org, адаптивний дизайн, lazy loading мультимедіа, регулярний технічний аудит та контентна оптимізація, дозволяє зробити сайт доступним, зручним і конкурентоспроможним.

Правильно реалізована оптимізація не лише підвищує позиції у пошукових системах і збільшує органічний трафік, але й формує позитивний

користувачський досвід, сприяє зростанню довіри аудиторії та досягненню бізнес-цілей.

Таким чином, оптимізація сайту є безперервним і багатогранним процесом, що поєднує технічні, маркетингові та дизайнерські аспекти й визначає ефективність цифрової присутності компанії чи організації в Інтернеті.

### **Питання для самоперевірки**

- 1) Що таке технічна оптимізація веб-сайтів і які її основні складові?
- 2) Чому важливе використання семантичного HTML? Наведіть приклади.
- 3) Як ключові слова впливають на SEO-оптимізацію?
- 4) Яке призначення мета-тегів title, description та canonical?
- 5) Що таке структурована розмітка Schema.org і які її переваги?
- 6) Які сучасні формати зображень використовуються для оптимізації продуктивності?
- 7) Як внутрішня перелінковка впливає на SEO та користувачський досвід?
- 8) Які основні принципи адаптивного дизайну й чому вони важливі для SEO?
- 9) Що рекомендують стандарти WCAG щодо контрастності тексту?
- 10) Як навігаційна структура та "хлібні крихти" допомагають користувачам?
- 11) Яку роль відіграють соціальні сигнали у просуванні сайту?
- 12) Чим відрізняється трафік із Twitter, LinkedIn та інших соцмереж?
- 13) Що таке зворотні посилання і як їх якість впливає на рейтинг сайту?
- 14) Які ризики пов'язані з надмірною концентрацією комерційних анкорів?
- 15) Як контент-маркетинг поєднується з SEO-стратегією?

## СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1. Duckett J. HTML & CSS: Design and Build Websites / Jon Duckett. – Indianapolis : Wiley, 2025. – 500 p.
2. Freeman E., Robson E. Head First HTML and CSS / Elisabeth Robson, Eric Freeman. – Sebastopol : O'Reilly Media, 2024. – 768 p.
3. W3Schools. HTML5 Нові елементи: електронний ресурс. – W3Schools українською, 2025. – Режим доступу: [https://w3schoolsua.github.io/html/html5\\_new\\_elements.html](https://w3schoolsua.github.io/html/html5_new_elements.html)
4. GoIT. HTML і CSS: що це, кому та для чого потрібно [Електронний ресурс] // GoIT Articles. – 2024. – Режим доступу: <https://goit.global/ua/articles/html-i-css-shcho-tse-komu-ta-dlia-choho-potribno/>
5. Riley J. Learning Web Design: A Beginner's Guide to HTML, CSS, JavaScript, and Web Graphics / Jennifer Niederst Robbins. – Sebastopol : O'Reilly Media, 2024. – 808 p.
6. Kurose J. Computer Networking: A Top-Down Approach / J. Kurose, K. Ross. – 8th ed. – Boston : Pearson, 2021. – 864 p.
7. Beaird J. The Principles of Beautiful Web Design / J. Beaird, J. George. – 4th ed. – Melbourne : SitePoint, 2020. – 216 p.
8. Cutts M. Google SEO Starter Guide [Електронний ресурс]. – Google, 2022. – Режим доступу: <https://developers.google.com/search/docs/beginner/seo-starter-guide> (дата звернення: 25.09.2025).
9. Cederholm D. Responsive Web Design with HTML5 and CSS / D. Cederholm. – 3rd ed. – Birmingham : Packt Publishing, 2020. – 520 p.
10. Ignatovsky R. The History of the Computer: People, Inventions, and Technology that Changed Our World / Rachel Ignatovsky. – New York : Ten Speed Press, 2022. – 128 p.
11. Metz C. Genius Makers: The Mavericks Who Brought AI To Google, Facebook, and the World / Cade Metz. – London : Penguin Books, 2021. – 384 p.

12. Miller C. Chip War. Боротьба за найважливішу технологію / Кріс Міллер ; перекл. з англ. – Київ : Наш Формат, 2023. – 416 с.
13. Свобода в мережі. Як насправді працює Інтернет : пер. з англ. / заг. ред. Є. Мостова. – Київ : Видавництво Старого Лева, 2021. – 302 с.
14. Війна за реальність. Як перемагати у світі фейків, правд і спільнот / Айзексон В., Мітнік К. – Київ : Yakaboo Publishing, 2021. – 320 с.
15. Lean UX: Створення класних продуктів із командами Agile / Джефф Готельф, Джош Сейден ; пер. з англ. С. Прокопенко. – Київ : Літопис, 2022. – 256 с.
16. Петренко І.В., Кравець М.Б. Інформаційні ресурси мережі Інтернет як джерело знань : монографія / І.В. Петренко, М.Б. Кравець. – Харків : ХНУРЕ, 2017. – 232 с.
17. World Wide Web Consortium (W3C). Web Standards [Електронний ресурс]. – Режим доступу: <https://www.w3.org> (дата звернення: 25.09.2025).
18. Internet Engineering Task Force (IETF). RFC Series [Електронний ресурс]. – Режим доступу: <https://www.rfc-editor.org> (дата звернення: 25.09.2025).
19. ISO/IEC 27001:2022 Information security, cybersecurity and privacy protection — Information security management systems — Requirements. – Geneva : International Organization for Standardization, 2022.
20. World Wide Web Consortium. HTML Living Standard [Електронний ресурс]. — Режим доступу: <https://html.spec.whatwg.org/>
21. World Wide Web Consortium. CSS Snapshot [Електронний ресурс]. — Режим доступу: <https://www.w3.org/Style/CSS/current-work>
22. W3C. Web Content Accessibility Guidelines (WCAG) 2.2 [Електронний ресурс]. — 2023. — Режим доступу: <https://www.w3.org/TR/WCAG22/>
23. W3C. DOM Standard [Електронний ресурс]. — Режим доступу: <https://dom.spec.whatwg.org/>

24. Internet Engineering Task Force. RFC 7540: Hypertext Transfer Protocol Version 2 (HTTP/2) [Электронный ресурс]. — May 2015. — Режим доступа: <https://www.rfc-editor.org/rfc/rfc7540>
25. Internet Engineering Task Force. RFC 9114: HTTP/3 [Электронный ресурс]. — June 2022. — Режим доступа: <https://www.rfc-editor.org/rfc/rfc9114>
26. Internet Engineering Task Force. RFC 9000: QUIC: A UDP-Based Multiplexed and Secure Transport [Электронный ресурс]. — May 2021. — Режим доступа: <https://www.rfc-editor.org/rfc/rfc9000>
27. Internet Engineering Task Force. RFC 9001: Using TLS to Secure QUIC [Электронный ресурс]. — May 2021. — Режим доступа: <https://www.rfc-editor.org/rfc/rfc9001>
28. Internet Engineering Task Force. RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3 [Электронный ресурс]. — August 2018. — Режим доступа: <https://www.rfc-editor.org/rfc/rfc8446>
29. Internet Engineering Task Force. RFC 1034: Domain Names — Concepts and Facilities [Электронный ресурс]. — November 1987. — Режим доступа: <https://www.rfc-editor.org/rfc/rfc1034>