MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE
NATIONAL TECHNICAL UNIVERSITY OF UKRAINE
“IGOR SIKORSKY KYIV POLYTECHNIC INSTITUTE”

Oleshchenko L.M.

FUNDAMENTALS OF WEB PROGRAMMING

Practical Tutorial

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для студентів, які навчаються
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення мультимедійних та
інформаційно-пошукових систем»)

Kyiv
Igor Sikorsky Kyiv Polytechnic Institute
2021

Рецензенти: Бідюк Петро Іванович, д-р техн. наук, проф.
Полторак Вадим Петрович, канд. техн. наук, проф.

Відповідальний редактор Легеза Віктор Петрович, д-р техн. наук, проф.

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського
(протокол № 8 від 24.06.2021 р.)
за поданням Вченої ради факультету прикладної математики
(протокол № 12 від 31.05.2021 р.)

Electronic online educational publication

Oleshchenko Liubov Mykhailivna, PhD, Associate Professor

FUNDAMENTALS OF WEB PROGRAMMING

PRACTICAL TUTORIAL

Fundamentals of Web Programming: Practical Tutorial [Електронний ресурс] : tutorial is aimed at students of the speciality 121 “Software Engineering” (educational program «Software Engineering of Multimedia and Information Retrieval Systems») / Igor Sikorsky Kyiv Polytechnic Institute; Liubov M. Oleshchenko. – Electronic text data (1 file: 4,78 Megabyte). – Kyiv: Igor Sikorsky Kyiv Polytechnic Institute, 2021. – 138 p.

This tutorial is developed for familiarizing students with basic theoretical matter and practical methods of web programming and requirements for laboratory work. The tutorial includes the introduction and 6 sections devoted to a certain laboratory task. There are a work objective, a description of the task, theoretical information, and methodological instructions for every laboratory task; questions for self-assessment and a list of recommended literature. The tutorial is aimed at students of the speciality 121 “Software Engineering”, educational program “Software Engineering of Multimedia and Information Retrieval Systems” of the Faculty of Applied Mathematics of Igor Sikorsky Kyiv Polytechnic Institute.

CONTENTS

INTRODUCTION

The discipline "Fundamentals of Web Programming" is part the of professionally-oriented disciplines cycle for bachelors of the speciality 121 “Software Engineering”.

This tutorial is developed for familiarizing students with basic theory and practical methods of of web programming and requirements for laboratory tasks.

The purpose of the tutorial is to gain skills in developing software for web applications using HTML, CSS, JavaScript, Bootstrap framework, jQuery and Leaflet library. Students also gain skills in server-side programming using Node.JS, an open source platform for high-performance networking applications written in JavaScript.

The tutorial includes the introduction and 6 sections devoted to a certain laboratory task. There are a work objective, a description of the task, theoretical information and methodological instructions for every laboratory tasks; questions for self-assessment and a list of recommended literature.

The tutorial is aimed at students of the speciality 121 “Software Engineering”, educational program “Software Engineering of Multimedia and Information Retrieval Systems” of the Faculty of Applied Mathematics of Igor Sikorsky Kyiv Polytechnic Institute.

LABORATORY WORK 1.

CREATING A WEB SITE USING HTML, CSS

Purpose: to master the skills of creating web sites in accordance with generally accepted standards, to learn how to design and create a simple web page using HTML and CSS.

Theory and methodological instructions

The process of creating a website (web-project) can be divided into 3 stages:

- planning;
- design;
- development.

Planning

This stage can be divided into several sub-steps:

- idea creation;
- project structure development;
- development of the project layout.

Idea creation

At this stage, we need to chose the subject of the project (site, service). Further, in accordance with the chosen topic, it is necessary to collect the relevant materials: text, graphic.

Project structure development

When we chose the topic of the project, we select the necessary material, the next step will be the development of the project structure. The structure of the project implies sections of the site, in accordance with which the navigation menu will be formed and the design of the project will be built. At this stage, we can classify the material into topics and sections.

Development of the project layout

After we have decided on the structure of the project, we can draw up a project layout (schematically). To draw a sketch, we can use paper and pen, Photoshop, any other graphics editor. It is important to note that this stage is not a drawing of the finished design layout, but just a sketch made to understand how the main information blocks, graphics and other design elements will be located on the site.

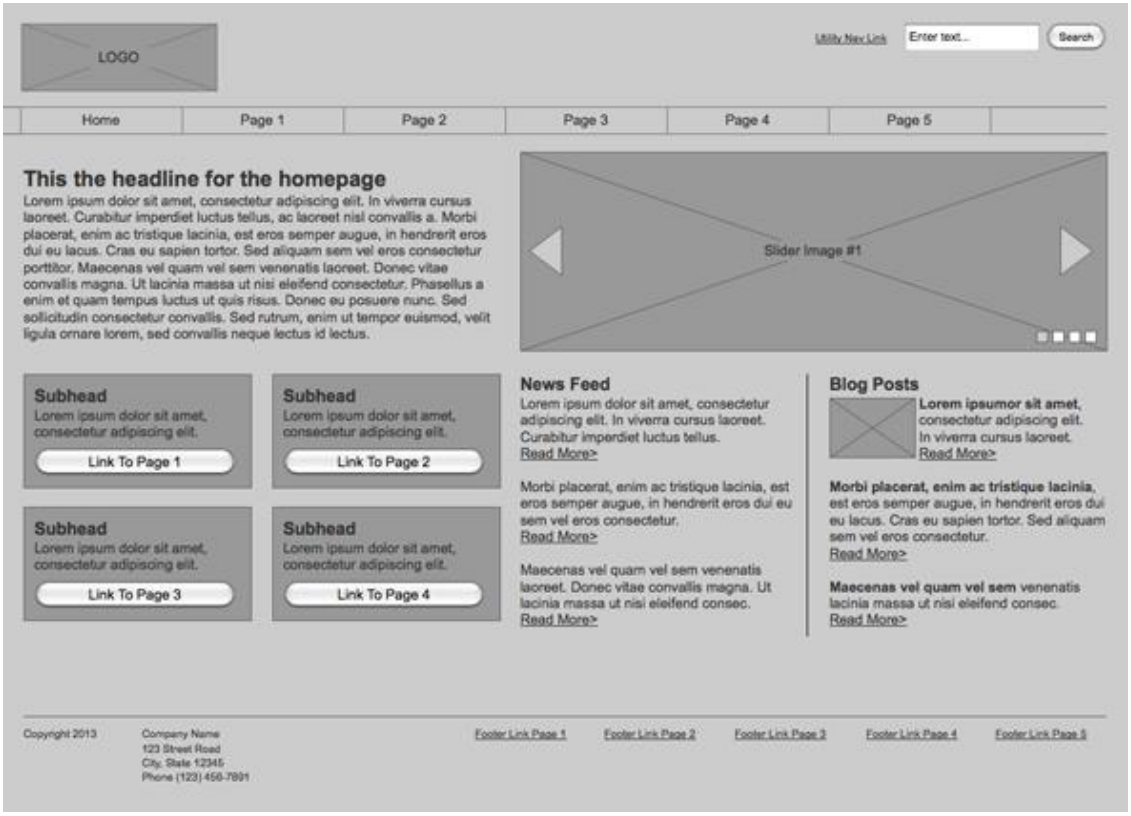


Fig.1.1. Website Prototype [1]

Key Page Elements

Often the main elements of a page are: *containing a block (wrapper, container)*, *logo*, *navigation*, *content*, *footer*, *free space* (essentially free space is not a design element, but a concept that is kept in mind when designing a page layout, our project will not look like a heap of blocks).

Containing block (container)

The role of the container on the page can be performed directly by the body element or *div*. The width of the containing block may be rubber (fluid), and may be fixed.

Logo

Text or graphic component of the project and distinguishing it from others. The logo is most often located in the upper left corner of the page or in the middle (depending on the idea, layout).

Navigation

The main navigation bar contains links to the main sections of the site. The navigation bar is often located at the top of the page (regardless of whether the navigation elements are vertically or horizontally).

Content

Content is the main component of a web page. It occupies a dominant role in the design of the page, so it takes up more space, supported by, in addition to text, graphics.

Footer

This element is located at the bottom of the page and usually contains information about the copyright holder, contact and legal information, links to the main sections of the site (often duplicates the main navigation), links to social networks, feedback form, etc.

Rubber and fixed layout

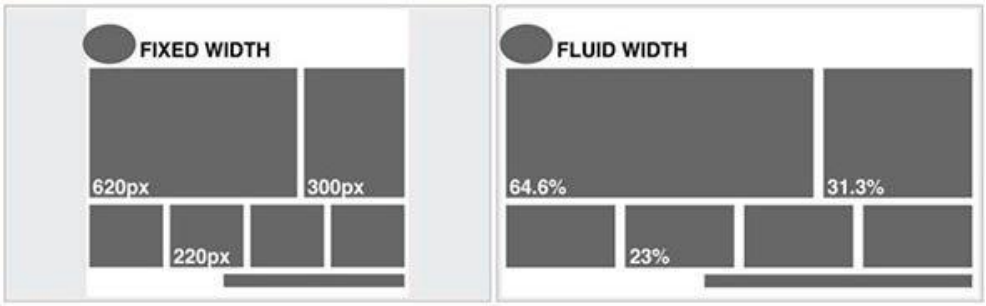


Fig.1.2. Fixed and "rubber" layouts [2]

Fixed layout

A fixed layout implies that regardless of the user's screen resolution, our site will always be the same width.

" Rubber" layout

The "rubber" layout implies that the page of the site will try to occupy all the space available to it on the user's screen, adjusting to the resolution.

In this context, concepts such as **responsive web design** and **adaptive web design** are worth understanding. The first concept fits into the concept of “rubber” and means that when we change the screen size, our site adapts to it, the second concept implies that during development we determine the basic permissions (screen sizes) that our content will adapt. In both cases, we should develop not one, but several layouts that will correspond to different screen resolutions. Often 3 layouts are created for the permissions of iPhone (Android Phone), iPad (Android Tablet) and Desktop.

When developing the layout of the mobile version of the site, developers try to bring the main content to the forefront, therefore the navigation menu is often hidden, large banners and decorative elements are hidden, the content blocks are usually placed under each other. On a pre-compiled layout, we can just decide which elements we leave on the mobile and which we hide.

Development

The page layout design process smoothly flows into the process of “revitalization” made in the previous stages. Before we start writing HTML, CSS, and JS right away, we should talk a bit about code editors and the project structure.

Code Editors

The most popular code editors are:

- Sublime Text (http://www.sublimetext.com/3);
- Atom (https://atom.io/);
- Brackets (http://brackets.io/).

Fig.1.3. Popular code editors (Sublime Text, Atom, Brackets)

In part, all these editors are similar in principle to work, when during installation we get an editor into which we can then “deliver” the necessary modules and plugins, so to speak, “editors on steroids”. The difference is only in the technologies that were used when writing the editors, if Sublime Text was written using C ++ and Python,

then the other 2 use JavaScript, HTML, CSS. Due to this difference, Sublime Text can work a little faster than its colleagues.

Project structure

Under the structure of the project refers to the storage of project files in its directory. When all the files are “piled” together, the file names are given in numbers or, for example, Ukrainian letters, etc., firstly, this is a disrespect for who will work with our project further, and secondly, the more your project will be, the more files will become and, in the end, you’ll just be confused about what is relevant and what is needed and what is not. It is best to put individual categories of files in their folders: pictures in the *images* or *img*, css in the *css* folder, JavaScript code in the *js* folder. Only *index.html* will be at the rootand pages of the site, or just *index.html*, and pages in a separate folder pages (Fig. 1.4). By following these rules we will never get confused in a project.

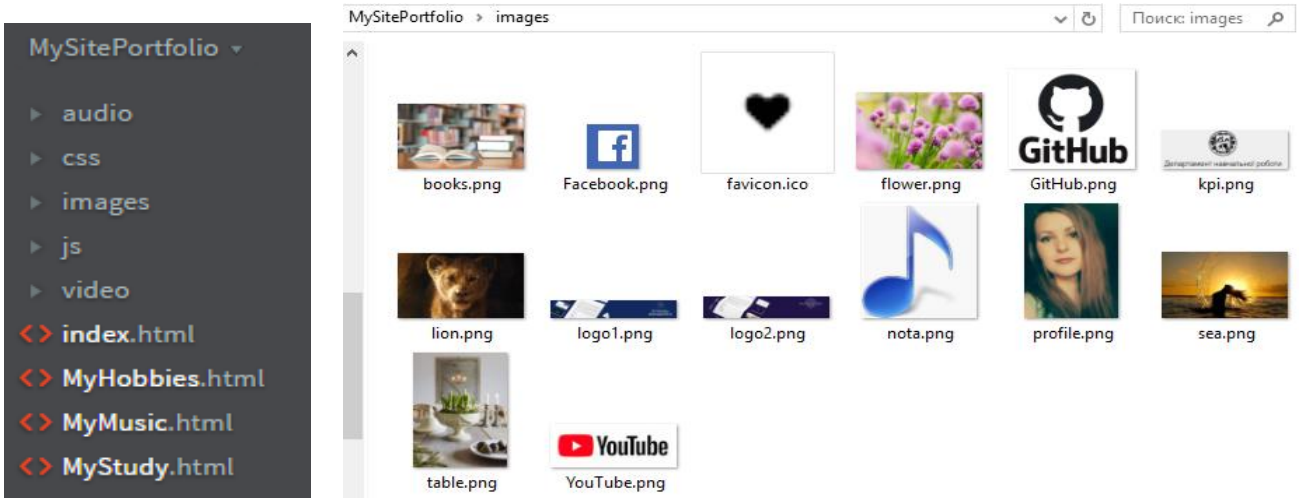


Fig.1.4. Project structure example

It is also worth mentioning the naming of project files. Most often, the following names are used: the main page is *index.html*, the styles of the project *styles.css*, scripts *scripts.js* or *app.js*, the minimized versions of the files have the prefix.min, the pictures are not long names in Russian or a set of numbers, but reflect what is depicted on them, for example, *button.png*, *download-icon.png*, *logo.png*, etc.

Project work

So, having decided on the code editor, structure, we can start development. Page layout is done in stages: first, an HTML structure (HTML code) is written, then styles are added, and then, if necessary, scripts (JS) are written, necessary plugins and libraries are added. Given the above, we can conditionally divide the work on the project into the following stages:

- writing HTML;
- writing CSS;
- writing JS.

Writing HTML

When writing HTML code, we can use the tags and markup elements that appeared with the HTML5 standard. At the time of site layout, a priori, a block approach is used, no tables, iframes, etc. Tables fulfill only their direct role – representing information in the form of a table. In layout tables are used only when working with emails.

At the stage of writing HTML, we, as it were, create the skeleton of the page, its abstract model using tags (HTML markup language). The structure may be easier to write if we have a prototype drawn up at the first stage, or if we ourselves, looking at the design layout, on a paper schematically painted all the blocks of the page. When writing markup, we can also immediately assign classes and identifiers to elements.

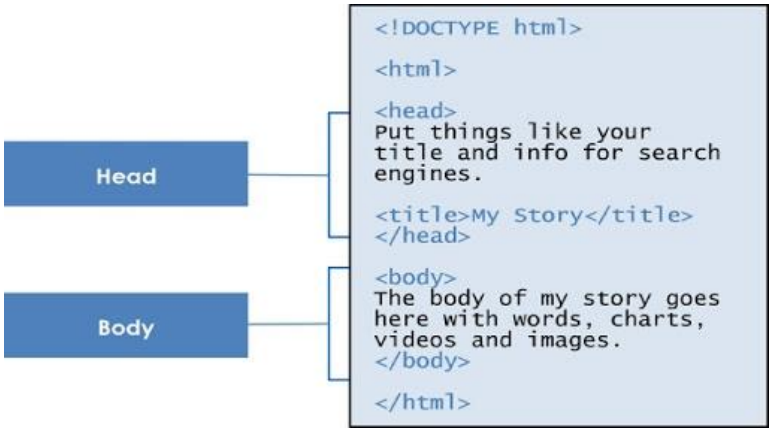


Fig. 1.5. Head and body of HTML document [3]

Fig. 1.6. The minimal structure of an HTML document [4]

Fig. 1.6 shows components of the HTML skeleton:

1. a document type declaration (also called DOCTYPE declaration) that identifies this document as an HTML5 document;

2. the entire document is contained within an **html** element wich is called the root element because it contains all the elements in the document, and it may not be contained within any other element;

3. the **head** element contains descriptive information about the document itself, such as its title, the style sheet(s) it uses, scripts, and other types of "meta" information:

Example 1 - Define keywords for search engines:

```
<meta name="keywords" content="HTML, CSS, XML, XHTML, JavaScript">
```

Example 2 - Define a description of your web page:

```
<meta name="description" content="Free Web tutorials on HTML and CSS">
```

Example 3 - Define the author of a page:

```
<meta name="author" content="John Doe">
```

Example 4 - Refresh document every 30 seconds:

```
<meta http-equiv="refresh" content="30">
```

Example 5 - Setting the viewport to make your website look good on all devices:

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

Fig. 1.7. Examples of using <meta> tag [5]

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <meta name="description" content="Free Web tutorials">
  <meta name="keywords" content="HTML,CSS,XML,JavaScript">
  <meta name="author" content="John Doe">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
</head>
<body>

<p>All meta information goes in the head section...</p>

</body>
</html>
```

Fig. 1.8. Example of using <meta> tag in HTML document [5]

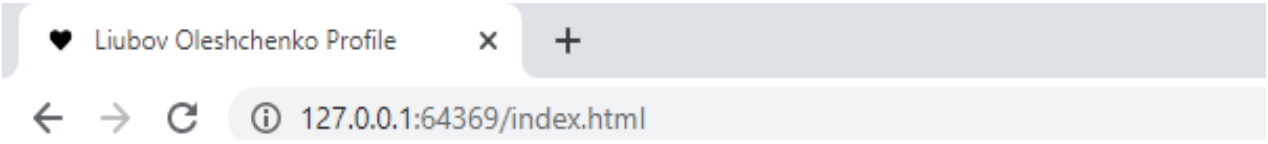
4. for specify the character encoding (the standardized collection of letters, numbers, and symbols) for the HTML document we use:

```
<head>
  <meta charset="UTF-8">
</head>
```

5. every document must contain a descriptive **title**, for example:

```
1  <!DOCTYPE html>
2  <html lang="en">
3
4  <head>
5      <title>Liubov Oleshchenko Profile</title>
6      <link rel="shortcut icon" href="images/favicon.ico">
7      <link rel="stylesheet" type="text/css" href="css/styles.css" />
8      <meta charset="utf-8">
9      <meta name="keywords" content="Liubov Oleshchenko, biography">
10 </head>
```

Visual result:



6. the **body** element contains everything that we want to show up in the browser window.

Figure 1.9 shows HTML4 and HTML5 structures.

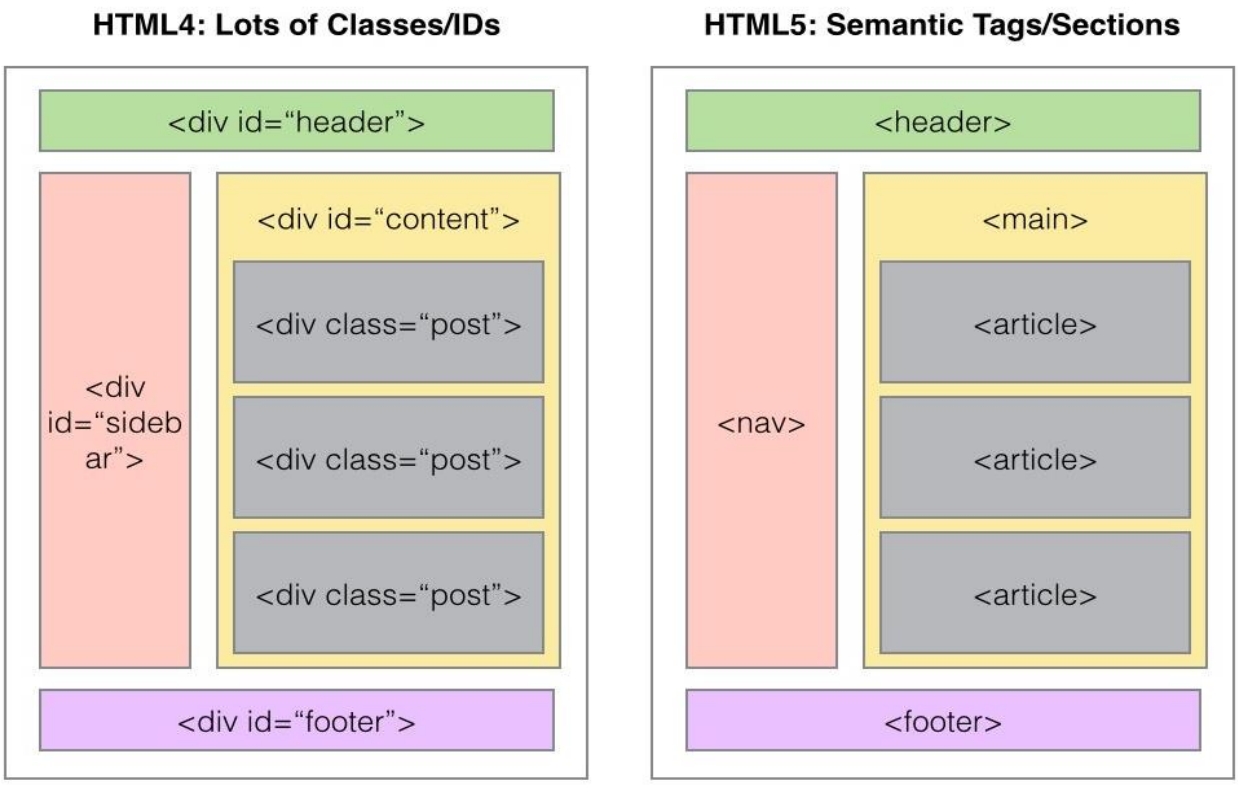


Fig. 1.9. HTML4 vs HTML5 [6]

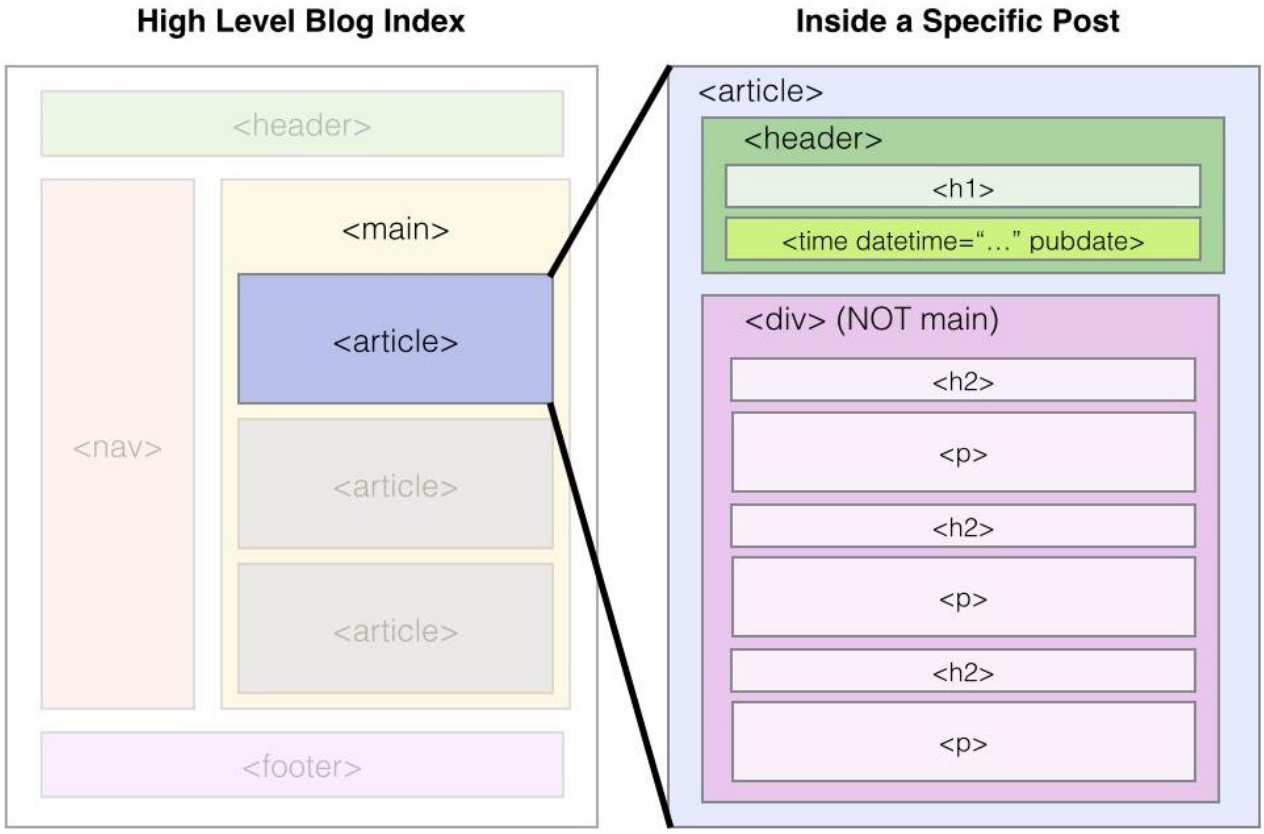


Fig. 1.10. Structure HTML5 sectioning element [6]

Table 1.1. Basic HTML tags

Tag	Description
<html>…</html>	Declares the Web page to be written in HTML
<head>…</head>	Descriptive information about the document
<title>…</ title>	Defines the title (not displayed on the pages)
<body>…</body>	Delimits the page's body
<h *n*>…</h *n*>	Delimits a level *n* headind
…	Set … in boldface
…	Set … in italics
…</ center>	Center … on the page horizontally
…	Brackets an unordered list
…	Brackets a numbered list
…	Brackets an item in an unordered or numbered list
 	Forces a line break here
<p>	Starts a paragraph
<hr>	Inserts a horizontal rule
<img src="…">	Displays an image here
<a href="…">…<a>	Defines a hyperlink

Example

For example, we have HTML code for web page **index.html:**

```
<body>
    <header>
        <img src="images/logo2.png" alt="Логотип" class="transparent70 w-100">
    </header>

    <nav>
        <a href="MySchedule.html"><button>Schedule</button></a> |
        <a href="MyHobbies.html"><button>My hobbies</button></a> |
        <a href="MyMusic.html"><button>My music</button></a>
    </nav>
    <div class="conte">
        <div class="widget">
            <h3 class="widget-title">Links</h3>
            <ul class="widget-list">
                <li><a href="https://sinoptik.ua/%D0%BF%D0%BE%D0%B3%D0%BE%D0%B4%D0%B0-%D0%BA%D0%B8%D0%B5%D0%B2">Weather</a></li>
                <li><a href="https://ua.112.ua/">News</a></li>
            </ul>
            <p><a target="_blank" href="https://www.youtube.com/watch?v=0j7MpGebZrg"><img src="images/YouTube.png" alt="YouTube"
            wight=40></a>
            </p>
            <p><a target="_blank" href="https://github.com"><img src="images/GitHub.png" alt="GitHub" wight=40></a></p>
        </div>
```

Example using CSS for this HTML code:

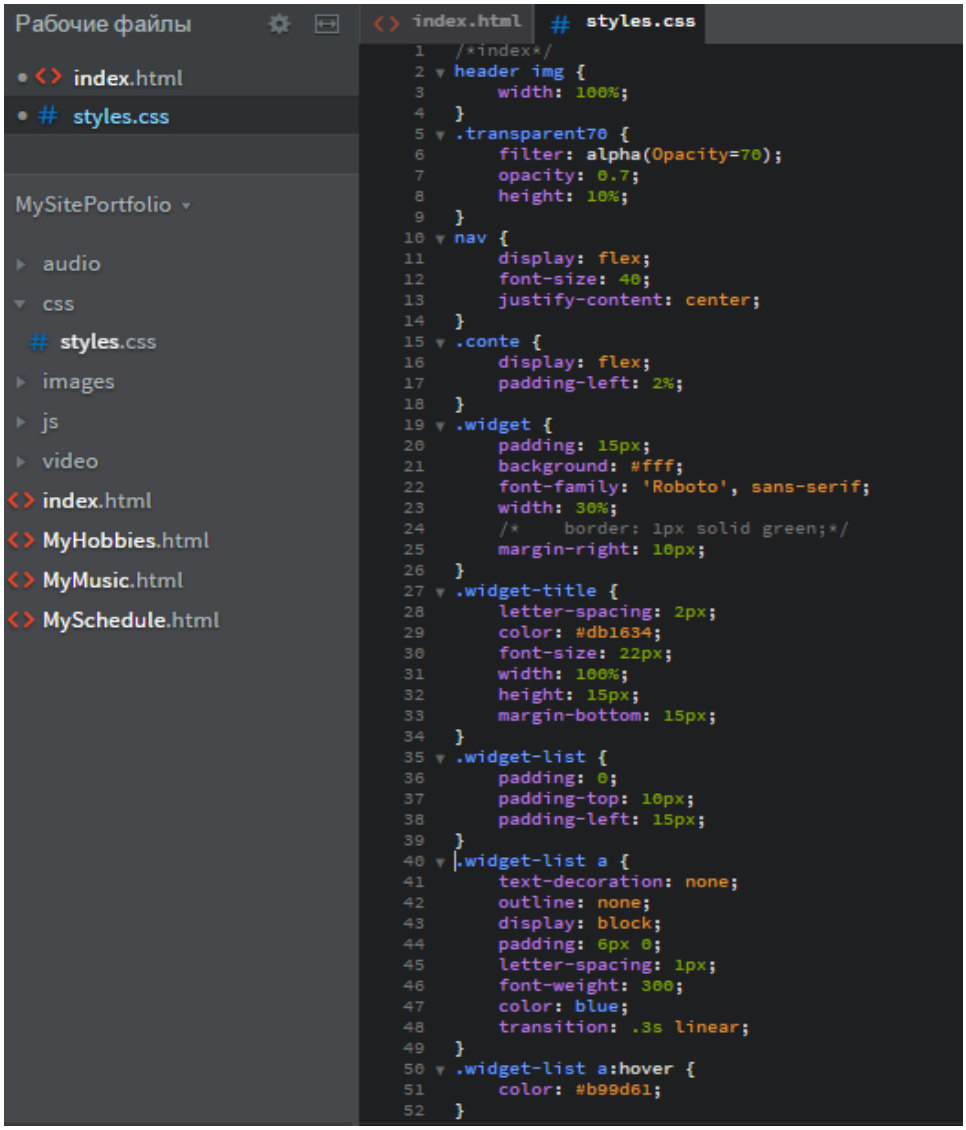


Fig. 1.11. Example CSS styling of web page (file **styles.css**)

Class Naming Rules

Everything should be in order in the project: from the project structure to class names, markup and code writing. If markup is important to monitor the type of information and its placement in the appropriate blocks (heading, list, link, line item, paragraph, etc.), then it is important to use common sense when naming classes and identifiers. Classes should give an abstract concept of the block to which they belong, so that the code is easier to read, and then write styles. In principle, there should be nothing complicated here, if we mark up the menu, then it is logical to give the containing block the class *.nav* or *.navigation*, if it is a block with text, then we can give it the class *.block-text*, etc.

All sizes and indents are taken directly from the design layout. To obtain these values, we need to use the “ruler” tool and guides (we are talking about Adobe Photoshop tools), and then transfer the obtained values to the code. If we work with a fixed layout, then the values are transferred in pixels as is, if we have “rubber”, then the values need to be converted to percentages. The basic formula is the width of the element divided by the width of the context (width of the containing block). For example, if a block containing text and a picture has a width of *400px* on the layout, and a block with text in it should have a width of *340px*, then in percentage terms it will be *(340/400) * 100%*, i.e. *85%* will occupy a block with text.

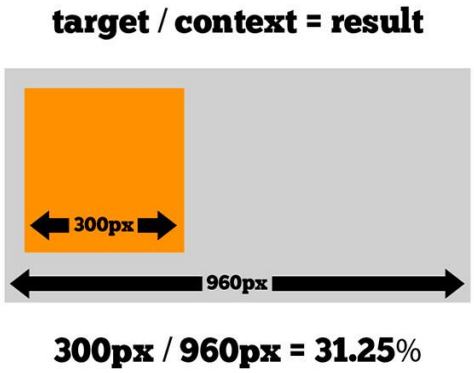


Fig.1.12. Converting size from pixels to percentages

Do not overload our styles with excessive specificity, for example, if we want the link inside the list to be red, then it is not necessary to write down the entire line of classes and tags: *.main –nav ul li a*, just specify *.main-nav a*. Too “specific" rules make the rules context-sensitive and force you to write redundant code, and also affect the speed of rendering the page, because when parsing CSS rules, the parser reads them from right to left and first, if you take the example described above, you need to take all the links (*a*), then drop all links that are not in *li*, etc. until he gets to the containing class. The specificity of the code also means excessive binding to the base tags.

For example, if we want the element inside *.block* to be blue and use the span element when marking (we wrote *.block span {background-color: blue}* in the rules), then when replacing it with a *div*, for example, we will need to write a new the rule is already for *div* inside *.block*. Thus, our code will be surrounded by unnecessary rules, which essentially duplicate each other. It is much simpler to define just one class

(element class), which, when assigned to internal elements, will lead us to the desired result. Another example of contextual code dependency is the use of tag names with class names, for example, *div.block {display: block}*. If we want to apply the same class to span, then we have to write *span.block {display: block}* or *span.block, div.block {display: block}* again, which in any case will lead to an increase in the code by an extra line. Why is it important? Firstly, we do not do unnecessary work, secondly, we do not need to search for the necessary rule among the 10,000 lines, and finally, the more lines of code, the larger the file size, and large files are slower to transfer over the network, which in turn, it can cause a long load, and excessive specificity can cause a long rendering of our page.

Code check

After writing html, css and js for our page, we need to check whether everything is done correctly. To do this, we can use online tools:

- To check the html: https://validator.w3.org /
- For CSS validation: http://jigsaw.w3.org/css-validator/
- To check JS: http://www.jslint.com/

Thanks to these services, we can check whether we forgot to close the tag somewhere, whether we use the parameters and attributes correctly, whether everything is in order with our styles and rules in them, and also check our code for the correct writing of functions, methods, etc.

Web Page Layouts

Among the variety of layout of the web page, there are four most common:

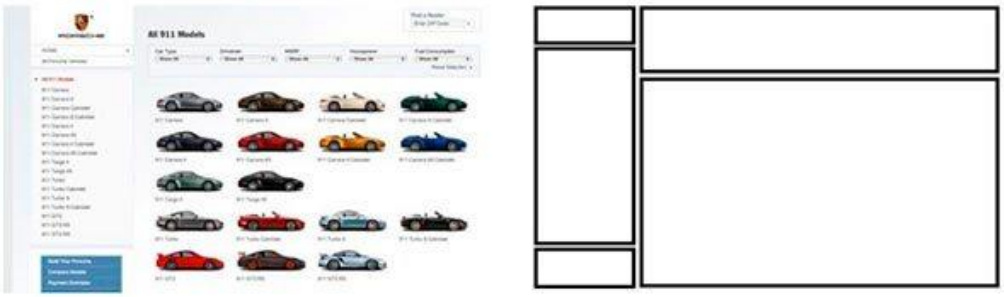


Fig.1.13. Left column navigation

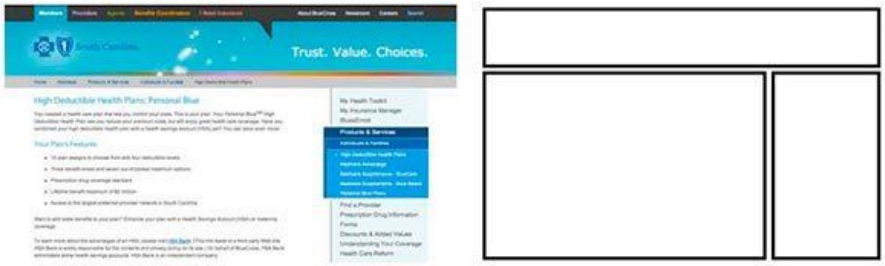


Fig. 1.14. Right column navigation

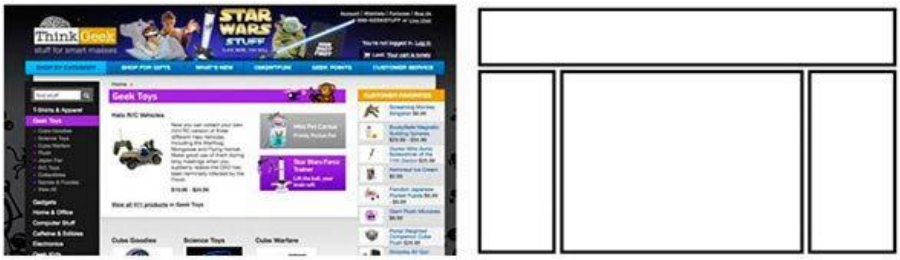


Fig. 1.15. Three-column navigation

At this stage, sites with this type of navigation make up the majority. The convenience of this approach is easily explained by the fact that in this case we have more space left for the content that makes up our site.

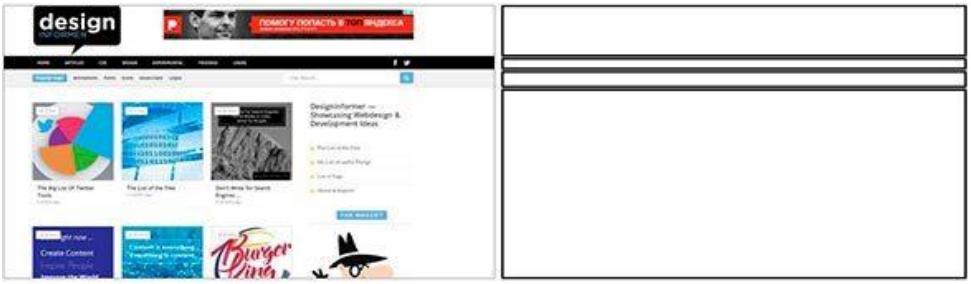


Fig. 1.16. Horizontal navigation

Tasks for laboratory work 1

1. Install the code editor and create the MySite folder in it. In this folder create a file for layout of the **index.html** page, a file for styling the elements of the **styles.css** page and a folder for storing images.

2. In a graphical editor (Paint, Photoshop, etc.) draw a web page layout about yourself with the layout structure according to the option taking into account the color scheme of the page (this file must also be demonstrated to the teacher).

3. Using free online services for creating logo (for example, https://www.logaster.com.ua/ or others) create a picture of the logo of your name

taking into account the color scheme of the page and save it in the project folder images with the name logo. When page layout, place the logo in the upper corner of the page.

Fig. 1.17. Example of long logo

4. The page should have the following information in the page blocks:

- The Header section should contain the student's logo;

- the Menu section (horizontal) should contain three hypertext links to three pages inside the "My Study" project (Page 1), where a numbered list of subjects and a link with a picture to the schedule will be located, on this page you must manually create a table with the schedule and a screenshot of the original;

"My hobbies" (Page 2), where you can vertically place an unordered list of your hobbies, place relevant photos (pictures) next to them;

"My Music" (Page 3), where the list of musical compositions from three audio and video tracks will be located (the tracks should be placed in the project's audio and video folder), the names of the performers should be made as hyperlinks to web pages with information about these performers, the pages will open in a new window.

- the Menu section (vertical) should contain a news feed, four hypertext external links (to four pages according to the option, the last two should be with a picture of the site for each position).

- the Content section contains a photo and a biography of the student, his main achievements and goals, at least three paragraphs (the text should contain words in bold and words in italics, underlined words, paragraphs are separated by a horizontal line);

- The Footer section contains the student's university address, e-mail, phone, a link to a page on the social network and the date the site was created.

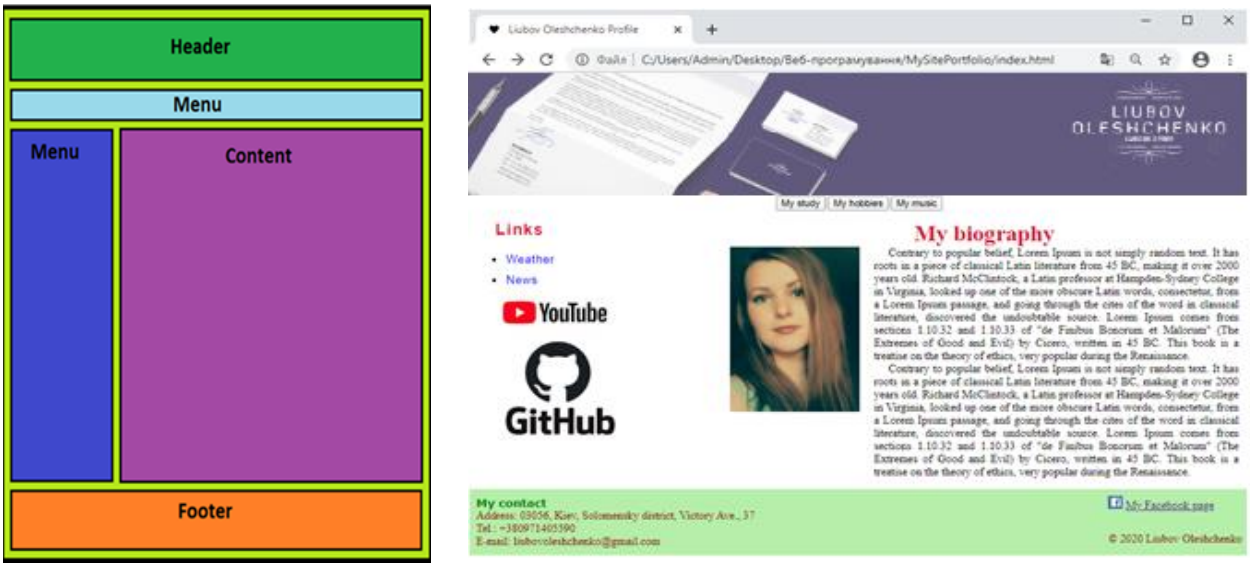


Fig. 1.18. Example with a vertical menu on the left side of the page
(dominant colors of the page are violet, green, white)

It is forbidden to use tables for marking, only basic tags , . Create site menus using lists (<ul>, <li>) with appropriate CSS formatting.

Check compliance with code standards at http://validator.w3.org.

Text and graphic fragments of pages are required.

The title of the page should be in the form of an image or text name. From each page, a transition to the main page should be available.

The maximum number of colors and fonts on the site is 4.

Create CSS styles in a separate file that connects to all pages of the site.

The student should be able to change the stylization (background, color), structure, placement of elements and replace elements of the web page with others.

It is forbidden to use any frameworks like Bootrstrap in this lab!

Table 1.1. Individual tasks (by number in the group list)

№	Location of the vertical menu	External hypertext links for the vertical menu	Dominant colors of the Web site (or their shades)
1	left	Your country's weather website Your country news site https://www.netacad.com/ https://www.python.org/	green, blue, white
2	right	Your country's weather website Your country news site https://github.com/ https://www.w3schools.com/	blue, white
3	left	Your country's weather website Your country news site https://www.netacad.com/ https://www.youtube.com/	purple, white
4	right	Your country's weather website Your country news site https://www.java.com/ https://www.w3schools.com/	red, white
5	left	Your country's weather website Your country news site https://github.com/ https://www.python.org/	green, white
6	right	Your country's weather website Your country news site https://www.netacad.com/ https://www.java.com/	pink, white
7	left	Your country's weather website Your country news site https://www.python.org/ https://www.youtube.com/	purple, white
8	right	Your country's weather website Your country news site https://github.com/ https://getbootstrap.com/	green, blue, white

9	left	Your country's weather website Your country news site https://www.netacad.com/ https://nodejs.org/	gray, green, white
10	right	Your country's weather website Your country news site https://www.java.com/ https://www.youtube.com/	red, white
11	left	Your country's weather website Your country news site https://github.com/ https://www.apple.com/	blue, white
12	right	Your country's weather website Your country news site https://www.netacad.com/ https://www.w3schools.com/	purple, white
13	left	Your country's weather website Your country news site https://www.java.com/ https://getbootstrap.com/	green, blue, white
14	right	Your country's weather website Your country news site https://github.com/ https://www.java.com/	pink, white
15	left	Your country's weather website Your country news site https://www.netacad.com/ https://getbootstrap.com/	blue, white
16	right	Your country's weather website Your country news site https://www.python.org/ https://www.youtube.com/	green, blue, white
17	left	Your country's weather website Your country news site https://github.com/ https://www.apple.com/	purple, white

18	right	Your country's weather website Your country news site https://www.netacad.com/ https://nodejs.org/	blue, white
19	left	Your country's weather website Your country news site https://www.python.org/ https://getbootstrap.com/	gray, green, white
20	right	Your country's weather website Your country news site https://github.com/ https://nodejs.org/	blue, white
21	left	Your country's weather website Your country news site https://www.netacad.com/ https://www.apple.com/	red, white
22	right	Your country's weather website Your country news site https://www.python.org/ https://www.w3schools.com/	purple, white
23	left	Your country's weather website Your country news site https://github.com/ https://www.youtube.com/	blue, white
24	right	Your country's weather website Your country news site https://www.netacad.com/ https://github.com/	pink, white
25	left	Your country's weather website Your country news site https://www.java.com/ https://www.w3schools.com/	green, white
26	right	Your country's weather website Your country news site https://getbootstrap.com/ https://www.apple.com/	pink, white

27	left	Your country's weather website Your country news site https://www.netacad.com/ https://nodejs.org/	green, blue, white
28	right	Your country's weather website Your country news site https://www.python.org/ https://www.w3schools.com/	red, white
29	left	Your country's weather website Your country news site https://www.java.com/ https://getbootstrap.com/	blue, white
30	right	Your country's weather website Your country news site https://www.netacad.com/ https://www.w3schools.com/	red, white

Report requirements for laboratory work

The report should include:

1. Title page.

2. Tasks for laboratory work.

3. Description of development steps. This section consists of a sequential description of the steps performed according to the instructions for laboratory work.

4. Conclusions.

Questions for self-assessment

1. What stages of website creation do you know?

2. What are HTML, CSS, JS used for?

3. What should be the structure of the web project?

4. What is the structure of the HTML code?

5. How to connect CSS to HTML?

6. How to connect external / internal links to the page?

7. What are the main elements of a web page?

8. What is the difference between a fixed and a "rubber" layout of a web page?
9. What class naming rules do you know?
10. Why we use <head> and <meta> tags in HTML document?
11. Which basic HTML tags do you know?
12. Which tag is used to add a picture to a web page?
13. Which tag is used to add a hyperlink to a web page?
14. Why we use record <meta charset=”UTF-8”> in HTML document?
15. What is the difference in HTML4 and HTML5 document structure?
16. Which common web page layouts do you know?

References

1. The Importance of Building a Website Prototype https://www.rattleback.com/insights/articles/why_building_a_prototype_is_so_important/
2. Resizing: Fixed, Fluid, or Responsive Layouts https://www.sitepoint.com/resizing-fixed-fluid-or-responsive-layouts/
3. Basic structure of an HTML document http://www.scriptingmaster.com/html/basic-structure-HTML-document.asp
4. Learning Web Design, 4th Edition by Jennifer Robbins https://www.oreilly.com/library/view/learning-web-design/9781449337513/ch04.html
5. HTML5 Semantic Tags https://www.vikingcodeschool.com/html5-and-css3/html5-semantic-tags
6. Web technologies for developers https://developer.mozilla.org/en/docs/Web
7. HTML Tutorial https://www.w3schools.com/html/default.asp
8. CSS Tutorial https://www.w3schools.com/css/default.asp
9. Markup Validation Service https://validator.w3.org/
10. CSS Validation Service https://jigsaw.w3.org/css-validator/

LABORATORY WORK 2.
JAVASCRIPT EVENTS HANDLING

Purpose: to consolidate knowledge of client script programming and to get practical JavaScript event handling skills.

Theory and methodological instructions

JavaScript or **JS** is a programming language that conforms to the ECMAScript specification. JavaScript is high-level, often just-in-time compiled, and multi-paradigm. It has dynamic typing, prototype-based object-orientation, and first-class functions. Alongside HTML and CSS, JavaScript is one of the core technologies of the World Wide Web. JavaScript enables interactive web pages and is an essential part of web applications. The vast majority of websites use it for client-side page behavior, and all major web browsers have a dedicated JavaScript engine to execute it [1-2].

As a multi-paradigm language, JavaScript supports event-driven, functional, and imperative programming styles. It has application programming interfaces (APIs) for working with text, dates, regular expressions, standard data structures, and the Document Object Model (DOM). JavaScript engines were originally used only in web browsers, but they are now embedded in some servers, usually via Node.js.

HTML events are "things" that happen to HTML elements (the change in the state of an object). When JavaScript is used in HTML pages, JavaScript can "react" on these events. This process of reacting over the events is called **Event Handling**. Thus, JS handles the HTML events via Event Handlers. An HTML event can be something the browser does, or something a user does.

Here are some examples of HTML events:

- an HTML web page has finished loading;
- an HTML input field was changed;
- an HTML button was clicked.

Often, when events happen, we may want to do something. JavaScript lets execute code when events are detected. HTML allows event handler attributes, with JavaScript code, to be added to HTML elements.

For example, when a user clicks over the browser, add JS code, which will execute the task to be performed on the event.

Common HTML Events

Table 2.1 shows a list of some common HTML events.

Table 2.1. Some common HTML events [3]

Event	Description
onchange	An HTML element has been changed
onclick	The user clicks an HTML element
onmouseover	The user moves the mouse over an HTML element
onmouseout	The user moves the mouse away from an HTML element
onkeydown	The user pushes a keyboard key
onload	The browser has finished loading the page

JavaScript's interaction with HTML is handled through events that occur when the user or the browser manipulates a page. When the page loads, it is called an event. When the user clicks a button, that click too is an event. Other examples include events like pressing any key, closing a window, resizing a window, etc.

Developers can use these events to execute JavaScript coded responses, which cause buttons to close windows, messages to be displayed to users, data to be validated, and virtually any other type of response imaginable.

Events are a part of the DOM Level 3 and every HTML element contains a set of events which can trigger JavaScript code.

Event handlers can be used to handle, and verify, user input, user actions and browser actions:

- things that should be done every time a page loads;
- things that should be done when the page is closed;
- action that should be performed when a user clicks a button;
- content that should be verified when a user inputs data.

Many different methods can be used to let JavaScript work with events:

- HTML event attributes can execute JavaScript code directly;
- HTML event attributes can call JavaScript functions;

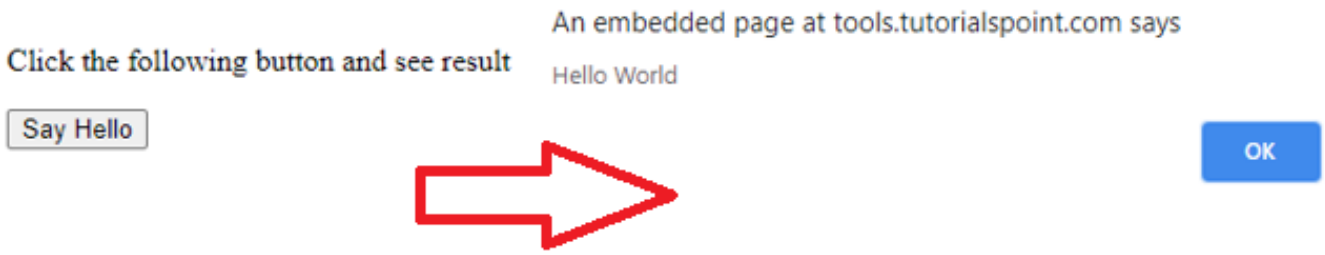
- we can assign own event handler functions to HTML elements;
- we can prevent events from being sent or being handled.

Example

onclick Event is the most frequently used event type which occurs when a user clicks the left button of his mouse.

```
<html>
   <head>

      <script type = "text/javascript">
         <!--
            function sayHello() {
               alert("Hello World")
            }
         //-->
      </script>

   </head>

   <body>
      <p>Click the following button and see result</p>

      <form>
         <input type = "button" onclick = "sayHello()" value = "Say Hello" />
      </form>

   </body>
</html>
```

Output:



Tasks for laboratory work 2

In the **MySite** folder, add the **taskjs.html** file. Make a button called "Task JS" on the page **My Study** to go to the page **taskjs.html**. Set the background image to the theme of the task, set the transparency of the background image to 45%.

Table 2.1. Individual tasks (by number in the group list)

№	Individual tasks (by number in the group list)
1	Create a page that contains a map of the city and attractions (zoo, theater, museum). When you hover over the landmark mark, an animated picture of the institution is displayed and a message about the schedule of the institution is displayed in a separate window.
2	Create a page containing test tasks on the subject "Fundamentals of Programming" (5 questions with 3 possible answers). One correct answer must be chosen. After each question is answered in a separate window, the user is given a message correctly or incorrectly, user answered and an animated picture with a hint.
3	Create a page containing a field for calculating the area of the rectangle according to the values of its sides entered in the dialog boxes. Enter an error message when entering a negative number. The output is displayed in a separate window.
4	Horoscope. Pressing one of the 12 buttons with the names of the zodiac signs shows a picture of the corresponding zodiac sign and a brief description of this sign in a separate window. Make a dialogue to enter the date of birth and find out the user's zodiac sign.
5	When you select a name from the group drop-down list, a student’s image is displayed in a separate window. When you hover over the image, the student’s date of birth is displayed.
6	Create an application that displays the current time by pressing the buttons "Current time", "Current day", "Current month" and "Current year". The result is displayed in a separate window. Create a dialog to enter the day of the week date for that date.
7	Create a calculator that finds the remainder of dividing two positive numbers. The output is displayed in a separate window. Enter an error message when entering a negative number. When dividing numbers, a picture with an emoticon is displayed.
8	When you select the name of a programming language (5 names) from a drop-down list, a text message about this language is displayed in a separate window. When hovering over the text, display the logo of the corresponding programming language.

9	Recipe of dish. When you press one of the 12 buttons on which the dish is marked, a recipe for this dish and animation (at least three pictures) of the cooking stages are displayed in a separate window.
10	When you select the name of the animal (10 names) from the drop-down list, animated images of this animal and its description are displayed in a separate window.
11	Create a page containing a map of the zoo and 6 animals marked on it. When you hover over the animals, an animated photo of the animal and its description are displayed.
12	Calculator for rounding a positive number. The result of the execution is displayed in a separate window. If you enter a negative number, display an error message.
13	Test tasks for the subject "Organization of computer networks" (3 questions with 5 answer options). You need to select some correct answers. After answering the question, a window is displayed where it is indicated correctly or incorrectly the answer is given.
14	When you select the name of the phone model (10 names) from the drop-down list, animated images of this phone and its technical characteristics are displayed in a separate window.
15	When you mouse over one of the 8 photos of the student, his name and phone number are displayed.
16	Application, asks the user for the current day of the week. After the answer, indicate its correctness in the form of an animated picture. If the answer is incorrect, report it and indicate the correct answer.
17	When you select the name of an outstanding scientist (10 names) from the drop-down list, an animated photo and a text message about his contribution to science are displayed in a separate window.
18	Test tasks on the subject of “Databases” (3 questions with 4 possible answers). You must select one correct answer. After answering each question in a separate window, the answer is indicated correctly or incorrectly.
19	Create a page on which, when you hover over one of the 6 images of birds, its name and audio accompaniment - a description of the bird - are displayed.

20	When you select the name from the drop-down list of the group, a text message is displayed in a separate window, where the student’s date of birth and his phone number are indicated.
21	Create a page containing a map of the territory of the university and mark buildings on it. When you move the cursor to the mark of the building, an animated photo of the building and the name of the faculty are displayed.
22	Car showroom. When you hover over each of the 6 images of cars, a message is displayed about its cost and technical characteristics.
23	Musical instruments. When you hover over one of 6 musical instrument, musical accompaniment in the performance of this instrument is displayed.
24	When you click on one of the 6 photos of students in a separate window, his name and group number are displayed.
25	Create a page containing a map and historical places marked on it. When you move the cursor to the label, audio accompaniment-message about this place is launched.
26	When choosing a plant name (10 names), an animated image of this plant and its description are displayed in a separate window.
27	When you select a name from the drop-down list of singers, a picture of the singer is displayed in a separate window. When you hover over the image, the date of birth of the singer and a short biography are displayed.
28	If you select the name of the medicine (10 names) from the drop-down list, the animated image of this medicine and its description are displayed in a separate window.
29	Cocktail recipe. Pressing one of the 12 buttons on which the cocktail is marked displays the recipe for this cocktail and the animation (at least three images) of the cooking steps in a separate window.
30	Test tasks on the subject "Object-Oriented Programming" (3 questions for 4 answer options). One correct answer must be chosen. After each question is answered in a separate field, the answer is right or wrong.

Report requirements for laboratory work

The report should include:

1. Title page.

2. Tasks for laboratory work.

3. Description of development steps. This section consists of a sequential description of the steps performed according to the instructions for laboratory work.

4. Conclusions.

Questions for self-assessment

1. Describe ways to connect a JS code to a web page.

2. Describe JavaScript language syntax.

3. Describe data types in JavaScript.

4. Data input / output. What are the three methods for displaying dialog boxes you know? How are alert, confirm, prompt methods different?

5. What is the use of the parseInt function?

6. Describe the document object and its document.write method.

7. Describe JavaScript Events. Types of events.

8. Assigning event handlers to items/

9. Load event and its onLoad handler.

10. Describe Click event.

11. Describe Math object methods.

12. How to call a method in JavaScript? Give an example.

13. Describe Date object.

14. Describe Date object methods.

15. Describe Array object.

16. How is the naming of document objects in JavaScript?

17. How is a document object referenced in JavaScript?

18. How do I access object properties in JavaScript?

19. The Window object and its methods.

20. How are window parameters set in JavaScript?
21. Methods open () and close () of document object.
22. Appeal to form elements - checkboxes, radio buttons, lists.
23. The checked object's input property.
24. Properties of the option object.
25. Image object and its properties.
26. Events and their handlers when interacting with the mouse.
27. Style object and its properties.
28. Properties of the display object.
29. Describe setTimeout (expression, msec) method.

References

1. Javascript https://runestone.academy/runestone/books/published/webfundamentals/Javascript/toctree.html
2. JavaScript https://www.w3schools.com/js/default.asp
3. JavaScript Events https://www.w3schools.com/js/js_events.asp
4. JavaScript Events https://www.javatpoint.com/javascript-events
5. Introduction to browser events https://javascript.info/introduction-browser-events

LABORATORY WORK 3.
JQUERY LIBRARY. CREATE A WEBSITE USING BOOTSTRAP FRAMEWORK

Purpose: to get practical skills working with the jQuery library, using the Bootstrap framework to create a website.

Theory and methodological instructions

jQuery

jQuery is a lightweight, "write less, do more", JavaScript library. The purpose of jQuery is to make it much easier to use JavaScript on a website. jQuery takes a lot of common tasks that require many lines of JavaScript code to accomplish, and wraps them into methods that we can call with a single line of code. jQuery also simplifies a lot of the complicated things from JavaScript, like AJAX calls and DOM manipulation.

The jQuery library contains the following features:

- HTML/DOM manipulation;
- CSS manipulation;
- HTML event methods;
- Effects and animations;
- AJAX;
- Utilities.

There are two versions of jQuery available for downloading:

- **production version** – for live website because it has been minified and compressed;
- **development version** – for testing and development (uncompressed and readable code);

Both versions can be downloaded from jQuery.com. The jQuery library is a single JavaScript file, and you reference it with the HTML <script> tag (the <script> tag should be inside the <head> section):

<head>

<script src="jquery-3.4.1.min.js"></script>

</head>

Instead downloading and host jQuery we can include it from a CDN (Content Delivery Network). Both Google and Microsoft host jQuery. To use jQuery from Google or Microsoft, use one of the following:

Google CDN:

<head>

<script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.4.1/jquery.min.js"></script>

</head>

Microsoft CDN:

<head>

<script src="https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.4.1.min.js"> </script>

</head>

The jQuery syntax is tailor-made for selecting HTML elements and performing some action on the element(s).

Basic syntax is: **$(selector).action()**.

A $ sign to define/access jQuery. A (selector) to "query (or find)" HTML elements. A jQuery action() to be performed on the element(s). Examples:

$(this).hide() – hides the current element.

$("p").hide() – hides all <p> elements.

$(".test").hide() – hides all elements with class="test".

$("#test").hide() – hides the element with id="test".

All jQuery methods in our examples, are inside a document ready event:

$(document).ready(function(){

// jQuery methods go here... });

This is to prevent any jQuery code from running before the document is finished loading (is ready). It is good practice to wait for the document to be fully loaded and ready before working with it. This also allows to have JavaScript code before the body of document, in the head section. Here are some examples of actions that can fail if methods are run before the document is fully loaded:

- trying to hide an element that is not created yet;
- trying to get the size of an image that is not loaded yet [1].

jQuery Event Methods

The jQuery team has also created an even shorter method for the document ready event:

```
$(function(){
  // jQuery methods go here...
});
```

jQuery selectors are used to "find" (or select) HTML elements based on their name, id, classes, types, attributes, values of attributes and much more. It's based on the existing CSS Selectors, and in addition, it has some own custom selectors.

An event represents the precise moment when something happens:

- moving a mouse over an element;
- selecting a radio button;
- clicking on an element.

The term "fires/fired" is often used with events. Example: "The keypress event is fired, the moment you press a key".

Table. 3.1. Some common DOM events [2]

Mouse Events	Keyboard Events	Form Events	Document/Window Events
click	keypress	submit	load
dblclick	keydown	change	resize
mouseenter	keyup	focus	scroll
mouseleave		blur	unload

In jQuery, most DOM events have an equivalent jQuery method. To assign a click event to all paragraphs on a page, you can do this:

```
$("p").click();
```

The next step is to define what should happen when the event fires. We must pass a function to the event:

```
$("p").click(function(){
  // action goes here!!
});
```

The $(document).ready() method allows to execute a function when the document is fully loaded.

The click() method attaches an event handler function to an HTML element.

The function is executed when the user clicks on the HTML element.

For example, when a click event fires on a <p> element; hide the current <p> element:

```
$("p").click(function(){
  $(this).hide();  });
```

The dblclick() method attaches an event handler function to an HTML element.

The function is executed when the user double-clicks on the HTML element:

```
$("p").dblclick(function(){
  $(this).hide(); });
```

The mouseenter() method attaches an event handler function to an HTML element. The function is executed when the mouse pointer enters the HTML element:

```
$("#p1").mouseenter(function(){
  alert("You entered p1!"); });
```

The mouseleave() method attaches an event handler function to an HTML element.

The function is executed when the mouse pointer leaves the HTML element:

```
$("#p1").mouseleave(function(){
  alert("Bye! You now leave p1!");
});
```

The mousedown() method attaches an event handler function to an HTML element. The function is executed, when the left, middle or right mouse button is pressed down, while the mouse is over the HTML element:

```
$("#p1").mousedown(function(){
  alert("Mouse down over p1!");
});
```

The mouseup() method attaches an event handler function to an HTML element.

The function is executed, when the left, middle or right mouse button is released, while the mouse is over the HTML element:

```
$("#p1").mouseup(function(){
  alert("Mouse up over p1!");
});
```

The focus() method attaches an event handler function to an HTML form field.

The function is executed when the form field gets focus:

```
$("input").focus(function(){
  $(this).css("background-color", "#cccccc");
});
```

The blur() method attaches an event handler function to an HTML form field.

The function is executed when the form field loses focus:

```
$("input").blur(function(){
  $(this).css("background-color", "#ffffff");
});
```

The on() method attaches one or more event handlers for the selected elements [2].

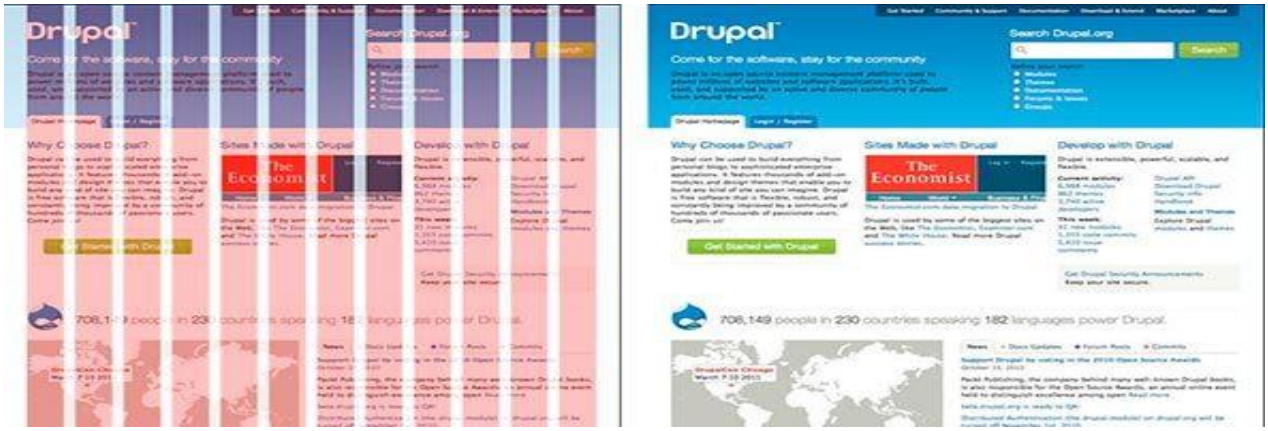
Bootstrap is a free front-end framework for faster and easier web development. Bootstrap includes HTML and CSS based design templates for typography, forms, buttons, tables, navigation, modals, image carousels and many other, as well as optional JavaScript plugins.

Bootstrap, Foundation, Material Design Lite frameworks offer ready-made design elements (buttons, input forms, etc.), their modular grid, CSS snippets (part of the code, markup, which can be used repeatedly) to insert elements into the page (the

same buttons, form elements, etc.) and markup classes, as well as JS scripts for the corresponding interactive elements. Using these libraries can significantly save time when developing a project (design, layout), though at the same time it can make site look like the others if we use the framework design elements as is. On the basis of each framework, we can find a huge number of paid and free themes and pages, as well as develop your own. One can not fail to mention some emerging trends of recent times as a layout and page design. It is worth mentioning the so-called landing pages, which mean a long page, divided into appropriate sections and familiarizing the user with the main content of the site.

Modular grid

Before scheduling a project, it is also necessary to understand the concept of a modular grid. The modular grid means dividing the page into separate columns vertically and arranging the content, while developing the layout design, precisely along this grid. The most popular system is the modular grid 960 Grid System, which divides the page as much as possible into 12, 16 and 24 columns. The maximum width of the grid is 960 pixels. This solution is based on the fact that most modern monitors, at the time of the grid creation, had a resolution of at least 1024 by 768 pixels. Creating a layout based on this grid, in the future, will help speed up the development process.



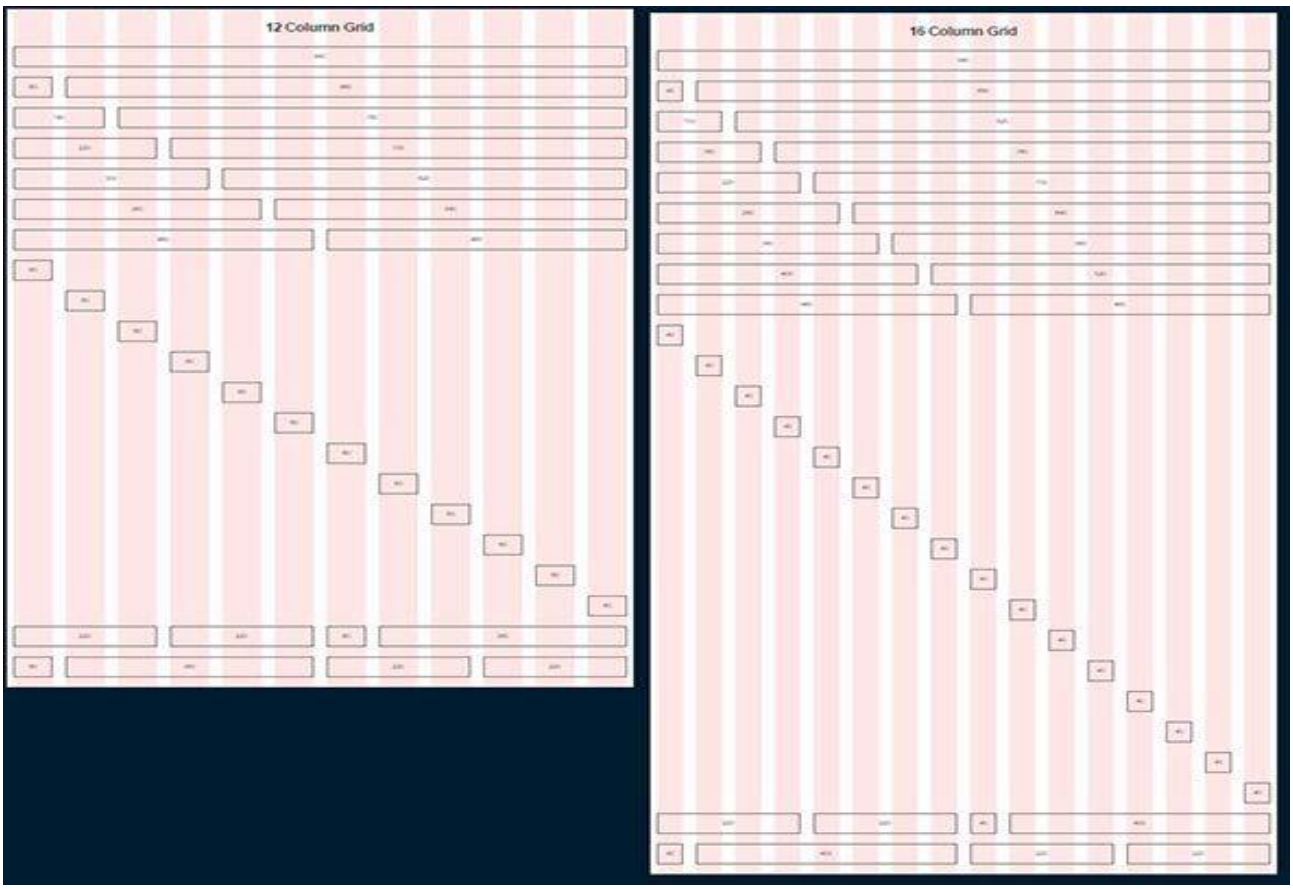


Fig.3.1. Layout grid [3]

When developing a “rubber” page layout, there is a concept of maximum width. This statement is based on the convenience of perception of information. If we assume that our site does not have a maximum width, then on large monitors the information will be very stretched and inconvenient to read. Most often limited to a width of 1280 pixels.

The 960GS modular grid meets the concept of a “fixed” design, for a “rubber” design you can pay attention to adapting the same grid at http://www.designinfluences.com/fluid960gs/ or use the grid that the Bootstrap framework offers (http: // getbootstrap.com/css/#grid).

Thanks to the modular grid, content blocks and elements will be located at a certain distance from each other, will have a digestible width, which in the future will be visually pleasing to the user and will not cause any inconvenience in the perception of the site. The modular grid, in fact, is a kind of visual abstraction, the visual division of the page into equally wide columns with equal indents between them. This model can

be visualized using guides or a separate layer on which these columns will be displayed.

Mobile first

Given the trends of recent years, this approach is tightly occupying its niche in the development and design of sites. The trend is that more than 60% of Internet users use mobile devices to access the network, so the development of not only the desktop version of the site, but also the mobile version becomes the rule of good taste. Using this approach, the development of the site's layout, design and layout begins with the mobile version, and then the layouts for other permissions are already worked out: blocks, banners, additional design elements, etc. are added.

After creating a project layout, you can go directly to creating a design layout. At this stage, it's worth starting with determining the color scheme of the project. One way to determine the primary color in a project is to create a mood board. To do this, we need to write all the synonyms associated with the topic of the project, and then type each synonym in the search for Google pictures. Based on the images found, write the colors that are most often found on them (which colors are more). The colors found will make up the visual perception of our project and evoke the corresponding feelings in the user.

Call to Action Elements

The concept of a call to action refers to the interactive elements of a site: buttons, banners, etc. These elements are designed in such a way that the user should definitely want to click on them. For example, it can be a button with a call to action (Click, Buy, Save), a bright banner with a tempting offer, a bright picture, etc.

Page layout

Quite often there is a Z-scheme for viewing the page. In accordance with this, page elements are usually positioned as follows: logo at the top left, menu at the top right, information blocks, pictures at the bottom left, a call to action button at the bottom right.

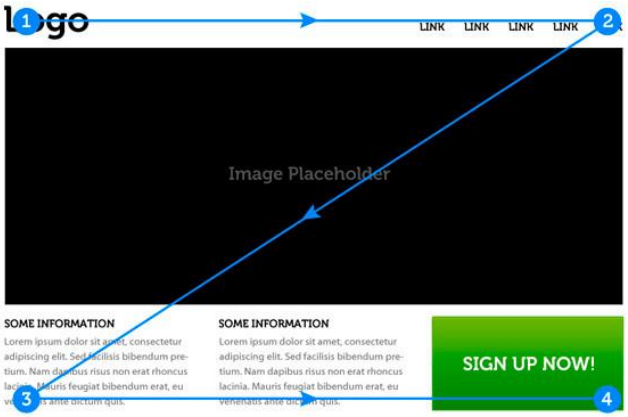


Fig. 3.2. Z-Pattern [4]

Landing

Landing is often the only page on which it is immediately possible to show all the necessary information without forcing the user to navigate through the pages. Landing is usually accompanied by a good design, accurate and thoughtful presentation of information, call-to-action elements, interactivity (counters, animation, etc.).

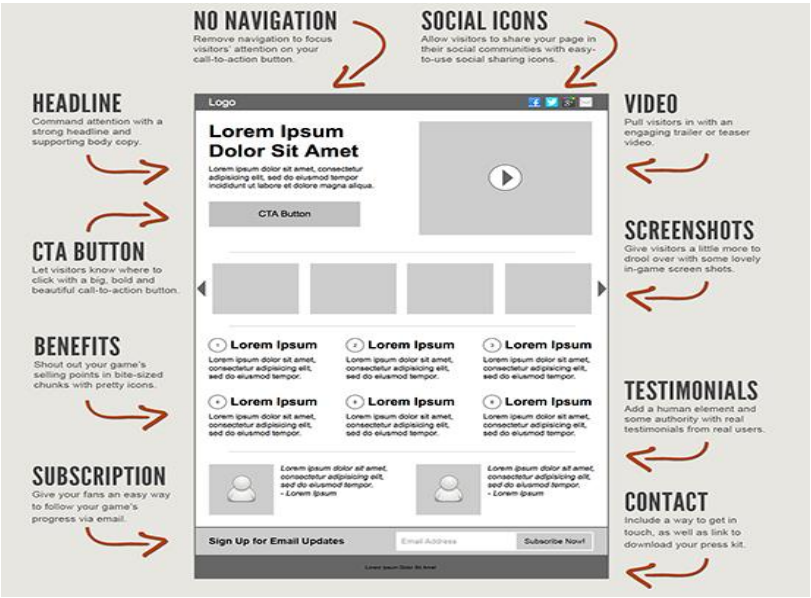


Fig. 3.3. An example of a landing page [5]

There are two ways to start using Bootstrap 4 on your own web site:

- include Bootstrap 4 from a CDN;
- download Bootstrap 4 from getbootstrap.com.

If you don't want to download and host Bootstrap 4 yourself, you can include it from a CDN (Content Delivery Network). MaxCDN provides CDN support for Bootstrap's CSS and JavaScript. We also include jQuery [6]:

```
  <!-- Latest compiled and minified CSS -->
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.4.1/css
/ bootstrap.min.css">
<!-- jQuery library -->
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.4.1/jquery.min.js">
</script>
<!-- Popper JS -->
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.16.0/umd/popper.
min.js"></script>
<!-- Latest compiled JavaScript -->
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.4.1/js/bootstrap.min.js
"></script>
```

Many users already have downloaded Bootstrap 4 from MaxCDN when visiting another site. As a result, it will be loaded from cache when they visit your site, which leads to faster loading time. Also, most CDN's will make sure that once a user requests a file from it, it will be served from the server closest to them, which also leads to faster loading time. Bootstrap 4 use jQuery and Popper.js for JavaScript components (like modals, tooltips, popovers etc). For downloading and hosting Bootstrap 4, go to https://getbootstrap.com/, and follow the instructions there.

Bootstrap 4 is designed to be responsive to mobile devices. Mobile-first styles are part of the core framework. To ensure proper rendering and touch zooming, add the following <meta> tag inside the <head> element:

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

The **width=device-width** part sets the width of the page to follow the screen-width of the device (which will vary depending on the device). The **initial-scale=1** part sets the initial zoom level when the page is first loaded by the browser.

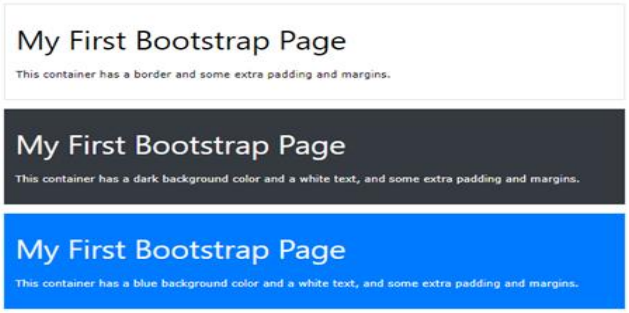
Bootstrap 4 also requires a containing element to wrap site contents. There are two container classes to choose from:

- the **.container** class provides a responsive fixed width container;
- the **.container-fluid** class provides a full width container, spanning the entire width of the viewport [6].

By default, containers have 15px left and right padding, with no top or bottom padding. Therefore, we often use spacing utilities, such as extra padding and margins to make them look even better. For example, **.pt-3** means "add a top padding of 16px":

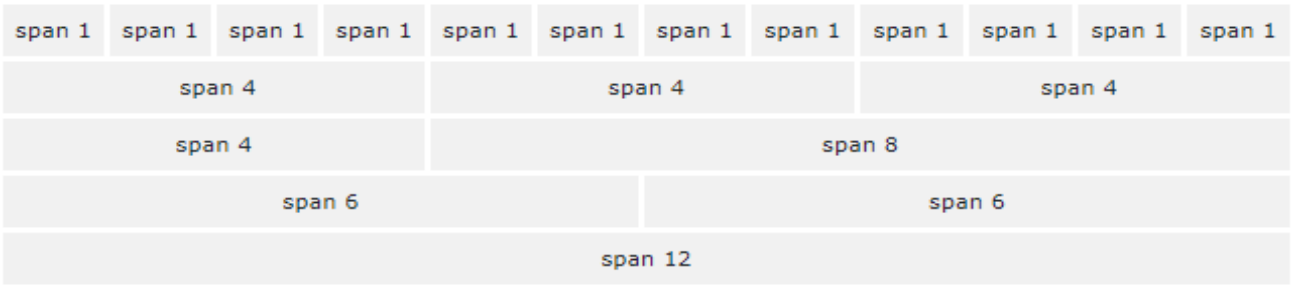
```
<div class="container pt-3"></div>
```

Other utilities, such as borders and colors, are also often used together with containers [6]:




```
<div class="container p-3 my-3 border"></div>
<div class="container p-3 my-3 bg-dark text-white"></div>
<div class="container p-3 my-3 bg-primary text-white"></div>
```

Bootstrap's grid system is built with flexbox and allows up to 12 columns across the page. If we do not want to use all 12 columns individually, we can group the columns together to create wider columns:



The grid system is responsive, and the columns will re-arrange automatically depending on the screen size. Make sure that the sum adds up to 12 or fewer (it is not required that you use all 12 available columns).

The Bootstrap 4 grid system has five classes:

.col- (extra small devices - screen width less than 576px)

.col-sm- (small devices - screen width equal to or greater than 576px)

.col-md- (medium devices - screen width equal to or greater than 768px)

.col-lg- (large devices - screen width equal to or greater than 992px)

.col-xl- (xlarge devices - screen width equal to or greater than 1200px).

The classes above can be combined to create more dynamic and flexible layouts. Each class scales up, so if you wish to set the same widths for **sm** and **md**, you only need to specify **sm**.

The following is a basic structure of a Bootstrap 4 grid:

```
  <!-- Control the column width, and how they should appear on different devices -->
<div class="row">
  <div class="col-*-*"></div>
  <div class="col-*-*"></div>
</div>
<div class="row">
  <div class="col-*-*"></div>
  <div class="col-*-*"></div>
  <div class="col-*-*"></div>
</div>
<!-- Or let Bootstrap automatically handle the layout -->
<div class="row">
  <div class="col"></div>
  <div class="col"></div>
  <div class="col"></div>
</div>
```

Example 1. Create a row ( ). Then, add the desired number of columns (tags with appropriate **.col-*-*** classes). The first star (*) represents the responsiveness: sm, md, lg or xl, while the second star represents a number, which should add up to 12 for each row.

Example 2. Instead of adding a number to each col, let bootstrap handle the layout, to create equal width columns: two "col" elements = 50% width to each col. three cols = 33.33% width to each col. four cols = 25% width, etc. We can also use **.col-sm|md|lg|xl** to make the columns responsive [6].

The following example shows how to create three equal-width columns, on all devices and screen widths:

.col	.col	.col

```
  <div class="row">
  <div class="col">.col</div>
  <div class="col">.col</div>
  <div class="col">.col</div>
</div>
```

The following example shows how to create four equal-width columns starting at tablets and scaling to extra large desktops.

On mobile phones or screens that are less than 576px wide, the columns will automatically stack on top of each other:

.col-sm-3	.col-sm-3	.col-sm-3	.col-sm-3

```
  <div class="row">
  <div class="col-sm-3">.col-sm-3</div>
  <div class="col-sm-3">.col-sm-3</div>
  <div class="col-sm-3">.col-sm-3</div>
  <div class="col-sm-3">.col-sm-3</div>
</div>
```

Bootstrap 4 Default Settings

Bootstrap 4 uses a default font-size of 16px, and its line-height is 1.5. The default font-family is "Helvetica Neue", Helvetica, Arial, sans-serif. In addition, all <p> elements have margin-top: 0 and margin-bottom: 1rem (16px by default) [6].

Text Colors

Bootstrap 4 has some contextual classes that can be used to provide "meaning through colors":

```
<p class="text-muted">This text is muted.</p>
<p class="text-primary">This text is important.</p>
<p class="text-success">This text indicates success.</p>
<p class="text-info">This text represents some information.</p>
<p class="text-warning">This text represents a warning.</p>
<p class="text-danger">This text represents danger.</p>
<p class="text-secondary">Secondary text.</p>
<p class="text-dark">This text is dark grey.</p>
<p class="text-body">Default body color (often black).</p>
<p class="text-light">This text is light grey (on white background).</p>
<p class="text-white">This text is white (on white background).</p>
```

This text is muted.

This text is important.

This text indicates success.

This text represents some information.

This text represents a warning.

This text represents danger.

Secondary text.

This text is dark grey.

Default body color (often black).

The classes for background colors are: **.bg-primary, .bg-success, .bg-info, .bg-warning, .bg-danger, .bg-secondary, .bg-dark and .bg-light**. Background colors do not set the text color, so in some cases you'll want to use them together with a **.text-*** class [6].

```
<p class="bg-primary text-white">This text is important.</p>
<p class="bg-success text-white">This text indicates success.</p>
<p class="bg-info text-white">This text represents some information.</p>
<p class="bg-warning text-white">This text represents a warning.</p>
<p class="bg-danger text-white">This text represents danger.</p>
<p class="bg-secondary text-white">Secondary background color.</p>
<p class="bg-dark text-white">Dark grey background color.</p>
<p class="bg-light text-dark">Light grey background color.</p>
```

This text is important.

This text indicates success.

This text represents some information.

This text represents a warning.

This text represents danger.

Secondary background color.

Dark grey background color.

Light grey background color.

Bootstrap 4 Basic Table

A basic Bootstrap 4 table has a light padding and horizontal dividers. The **.table** class adds basic styling to a table:

```
<table class="table">
  <thead>
    <tr>
      <th>Firstname</th>
      <th>Lastname</th>
      <th>Email</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>John</td>
      <td>Doe</td>
      <td>john@example.com</td>
    </tr>
    <tr>
      <td>Mary</td>
      <td>Moe</td>
      <td>mary@example.com</td>
    </tr>
    <tr>
      <td>July</td>
      <td>Dooley</td>
      <td>july@example.com</td>
    </tr>
  </tbody>
</table>
```

Firstname	Lastname	Email
John	Doe	john@example.com
Mary	Moe	mary@example.com
July	Dooley	july@example.com

For example, the **.table-striped** class adds zebra-stripes to a table. The **.table-bordered** class adds borders on all sides of the table and cells. The **.table-hover** class adds a hover effect (grey background color) on table rows. The **.table-dark** class adds a black background to the table [7].

Bootstrap 4 Images

The **.rounded** class adds rounded corners to an image [8]:

```
<img src="cinqueterre.jpg" class="rounded" alt="Cinque Terre">
```

The **.rounded-circle** class shapes the image to a circle:

```
<img src="cinqueterre.jpg" class="rounded-circle" alt="Cinque Terre">
```

The **.img-thumbnail** class shapes the image to a thumbnail (bordered):

```
<img src="cinqueterre.jpg" class="img-thumbnail" alt="Cinque Terre">
```

Float an image to the right with the **.float-right** class or to the left with **.float-left**:

```
<img src="paris.jpg" class="float-left">
<img src="paris.jpg" class="float-right">
```

Center an image by adding the utility classes **.mx-auto** (margin:auto) and **.d-block** (display:block) to the image [8]:

```
<img src="paris.jpg" class="mx-auto d-block">
```

A **jumbotron** indicates a big grey box for calling extra attention to some special content or information:

```
<div class="jumbotron">
  <h1>Bootstrap Tutorial</h1>
  <p>Bootstrap is the most popular HTML, CSS...</p>
</div>
```

Button Styles

Bootstrap 4 provides different styles of buttons [9]:


```
<button type="button" class="btn">Basic</button>
<button type="button" class="btn btn-primary">Primary</button>
<button type="button" class="btn btn-secondary">Secondary</button>
<button type="button" class="btn btn-success">Success</button>
<button type="button" class="btn btn-info">Info</button>
<button type="button" class="btn btn-warning">Warning</button>
<button type="button" class="btn btn-danger">Danger</button>
<button type="button" class="btn btn-dark">Dark</button>
<button type="button" class="btn btn-light">Light</button>
<button type="button" class="btn btn-link">Link</button>
```

Bootstrap 4 Cards

A card in Bootstrap 4 is a bordered box with some padding around its content. It includes options for headers, footers, content, colors, etc. Use **.card-title** to add card titles to any heading element. The **.card-text** class is used to remove bottom margins for a <p> element if it is the last child (or the only one) inside **.card-body**. The **.card-link** class adds a blue color to any link, and a hover effect. Add **.card-img-top** or **.card-img-bottom** to an <img> to place the image at the top or at the bottom inside the card.

We have added the image outside of the **.card-body** to span the entire width [10].

```
<html lang="en">
<head>
  <title>Bootstrap Card Example</title>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css">
  <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.4.1/jquery.min.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.16.0/umd/popper.min.js"></script>
  <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.4.1/js/bootstrap.min.js"></script>
</head>
<body>
<div class="container">
    <div class="card" style="width:400px">
    <img class="card-img-top" src="images/1profile.png" alt="Card image" style="width:100%">
    <div class="card-body">
      <h4 class="card-title">Liubov Oleshchenko</h4>
      <p class="card-text">Teacher</p>
      <a href="#" class="btn btn-primary">See Profile</a>
    </div>
  </div>
   </div>
</body>
</html>
```

For creating a simple horizontal menu, add the **.nav class** to a <ul> element, followed by **.nav-item** for each <li> and add the **.nav-link** class to their links [11]:

Link Link Link Disabled

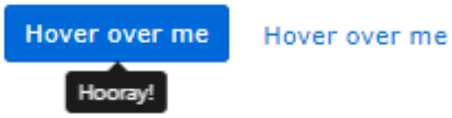
```
<ul class="nav">
<li class="nav-item">
  <a class="nav-link" href="#">Link</a> </li>
<li class="nav-item">
  <a class="nav-link" href="#">Link</a> </li>
<li class="nav-item">
  <a class="nav-link" href="#">Link</a> </li>
<li class="nav-item">    <a class="nav-link disabled" href="#">Disabled</a>
</li>    </ul>
```

Bootstrap 4 Navigation Bar

A navigation bar is a navigation header that is placed at the top of the page:

With Bootstrap, a navigation bar can extend or collapse, depending on the screen size [12].

The **Tooltip** component is small pop-up box that appears when the user moves the mouse pointer over an element:



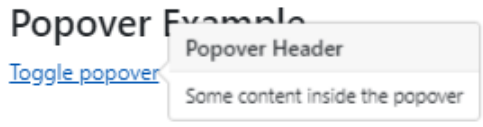
To create a tooltip, add the **data-toggle="tooltip**" attribute to an element. Use the title attribute to specify the text that should be displayed inside the tooltip:

```
<a href="#" data-toggle="tooltip" title="Hooray!">Hover over me</a>
```

Tooltips must be initialized with jQuery: select the specified element and call the **tooltip()** method. The following code will enable all tooltips in the document:

```
<script>
$(document).ready(function(){
  $('[data-toggle="tooltip"]').tooltip();
});
</script>
```

The **Popover** component is similar to tooltips; it is a pop-up box that appears when the user clicks on an element. The difference is that the popover can contain much more content.



To create a popover, add the **data-toggle="popover"** attribute to an element. Use the title attribute to specify the header text of the popover, and use the data-content attribute to specify the text that should be displayed inside the popover's body:

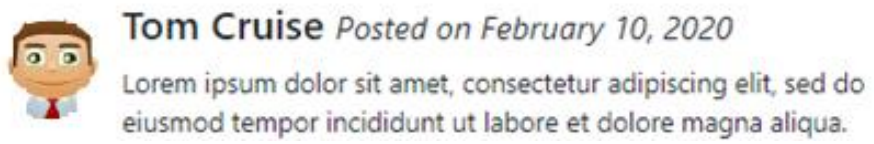
```
<a href="#" data-toggle="popover" title="Popover Header" data-
content="Some content inside the popover">Toggle popover</a>
```

Popovers must be initialized with jQuery: select the specified element and call the **popover()** method. The following code will enable all popovers in the document:

```
  <script>
$(document).ready(function(){
  $('[data-toggle="popover"]').popover();
});
</script>
```

Bootstrap 4 Media Objects

Bootstrap provides an easy way to align media objects (like images or videos) together with content. Media objects are often used to display blog comments, tweets and so on [13]:




```
<div class="media border p-3">
    <img src="images/10.jpg" alt="images/1profile" class="mr-3 mt-3
    rounded-circle" style="width:60px;">
    <div class="media-body">
        <h4>Tom Cruise <small><i>Posted on February 10, 2020</i></small>
        </h4>
        <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed
        do eiusmod tempor incididunt ut labore et dolore magna aliqua.
        </p>
    </div>
</div>
```

Tasks for laboratory work 3

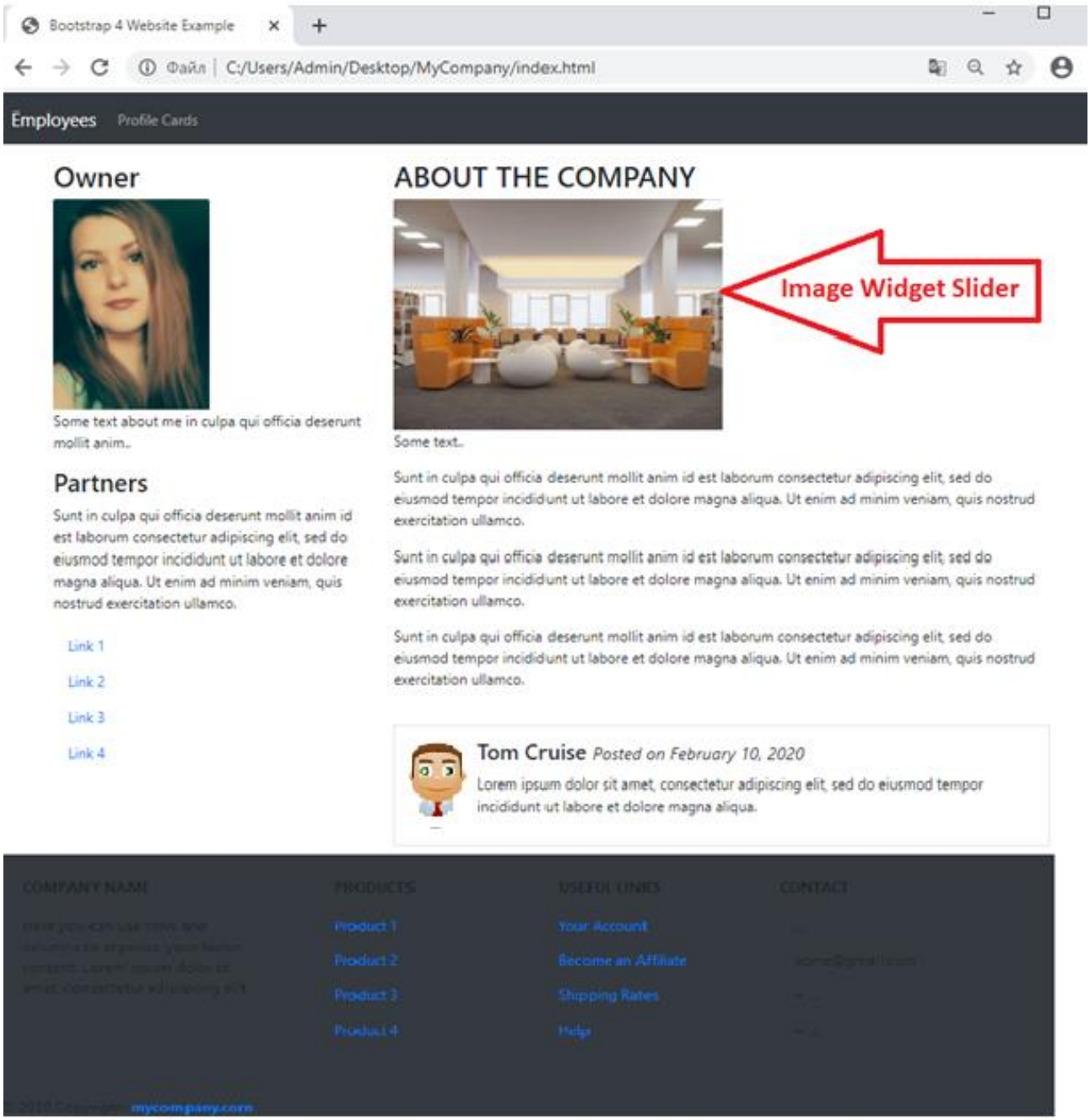
Imagine you are a business owner and you need to create a site for your business. The site should be responsive to different types of devices. Create **MyCompanyBootstrap** folder with Web page **index.html** using Bootstrap framework. Add photos and short information about the founder / owner of the business.

The page should have a vertical and horizontal navigation menu.

In the vertical navigation menu add 5 external links to company sites that conduct activities similar to your business. Analyze the code of the Web pages of these sites.

Add your business information and display a small blog comment from a well-known company service user (celebrity). The page should also have a footer.

Using jQuery library build an Image Slider Widget using 5 pictures (according to the subject of activity of the company) on the page using SetInterval function. When to click of mouse over Image Slider Widget, the photo animation must stop.



In the horizontal menu, create a link to the employee information page (using Bootstrap 4 Card), which, using the "See profile" button, opens detailed information about each of the best16 employees in a separate window.

Create one such window with details for one employee and create a slideshow for this employee with picture certificates (awards) using Bootstrap 4 Carousel.

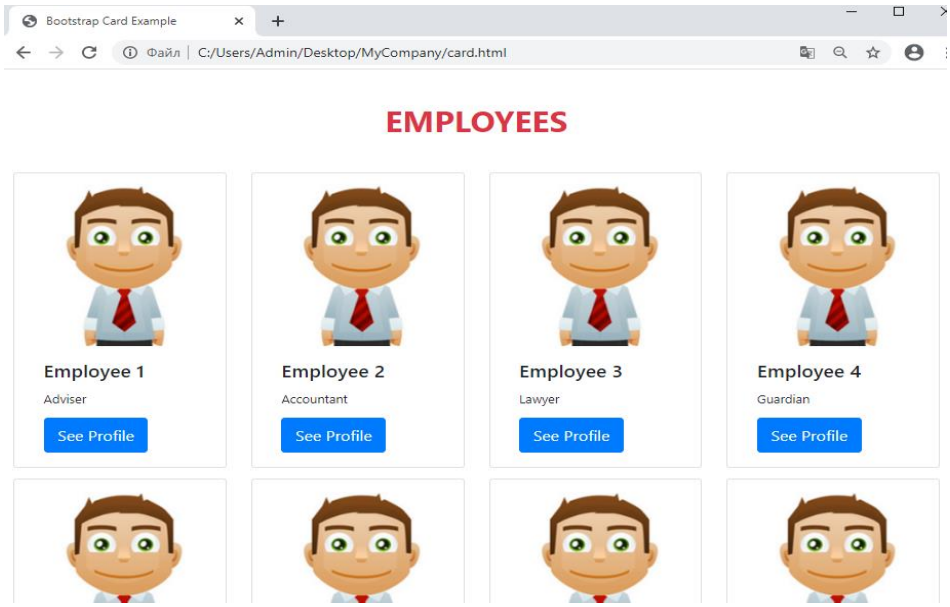


Table 3.1. Individual task (by number in the group list)

№	Location of the vertical menu	The topic of marketing activities of the company
1	right	appliances
2	left	tourism in your city
3	right	network equipment
4	left	perfumes
5	right	tea
6	left	medicine
7	right	clothes
8	left	sweets

9	right	travel agency
10	left	cars
11	right	phones
12	left	bicycles
13	right	coffee makers
14	left	stereo systems
15	right	ovens
16	left	kitchen combiners
17	right	electric kettles
18	left	refrigerators
19	right	computer equipment
20	left	sports equipment
21	right	motorcycles
22	left	server hardware
23	right	books
24	left	laptops
25	right	kitchens
26	left	Men's clothes
27	right	bags
28	left	shoes
29	right	air conditioners
30	left	vacuum cleaners

Report requirements for laboratory work

The report should include:

1. Title page.

2. Tasks for laboratory work.

3. Description of development steps. This section consists of a sequential description of the steps performed according to the instructions for laboratory work.

4. Conclusions.

Questions for self-assessment

1. Why do we use jQuery library?
2. Which jQuery library features do you know?
3. How to add jQuery library for a project?
4. What is basic jQuery syntax?
5. Which jQuery Event Methods do you know?
6. What is the modular grid used for?
7. What is the feature of landing pages?
8. Which ways to start using Bootstrap 4 on a web site do you know?
9. What mean classes:

.col-

.col-sm-

.col-md-

.col-lg-

.col-xl-?

10. What are Bootstrap 4 Default Settings?
11. Which colors mean classes .bg-primary, .bg-success, .bg-info, .bg-warning, .bg-danger, .bg-secondary, .bg-dark and .bg-light?
12. What is Bootstrap 4 jumbotron?
13. How to create a card in Bootstrap 4?
14. How to create a popover or tooltip components in Bootstrap 4?
15. How to create a media object in Bootstrap 4?

References

1. jQuery Tutorial https://www.w3schools.com/jquery/default.asp
2. jQuery Event Methods https://www.w3schools.com/jquery/jquery_events.asp
3. Effective Web Design Principles – Visual Hierarchy https://2stallions.com/blog/effective-web-design-principles-visual-hierarchy/#:~:text=Visual%20hierarchy%2C%20one%20of%20the,eye%20perceives%20what%20it%20sees.
4. Responsive layout grid https://material.io/design/layout/responsive-layout-grid.html#columns-gutters-margins
5. What Are Landing Pages? https://www.thebuzzstand.com/what-are-landing-pages/
6. Bootstrap 4 Tutorial https://www.w3schools.com/bootstrap4/default.asp
7. Bootstrap 4 Tables https://www.w3schools.com/bootstrap4 /bootstrap_tables.asp
8. Bootstrap 4 Images https://www.w3schools.com/bootstrap4/ bootstrap_images.asp
9. Bootstrap 4 Buttons https://www.w3schools.com/bootstrap4/ bootstrap_buttons.asp
10. Bootstrap 4 Cards https://www.w3schools.com/bootstrap4/ bootstrap_cards.asp
11. Bootstrap 4 Navs https://www.w3schools.com/bootstrap4/bootstrap_navs.asp
12. Bootstrap 4 Navigation Bar https://www.w3schools.com/bootstrap4/ bootstrap_navbar.asp
13. Bootstrap 4 Media Objects https://www.w3schools.com/bootstrap4/ bootstrap_media_objects.asp
14. 30 Best Bootstrap 4 Footer Templates in 2020 https://www.mockplus.com/blog/post/bootstrap-4-footer-template
15. Bootstrap 4 Carousel https://www.w3schools.com/bootstrap4/ bootstrap_carousel.asp

LABORATORY WORK 4.
NODE.JS. INSTALLATION OF MODULES. NODE.JS AS A FILE SERVER

Purpose: to get practical skills of creating a file web server, installation and use basic Node.js modules.

Theory and methodological instructions

Node.js is an open-source, cross-platform, back-end JavaScript runtime environment that runs on the V8 engine and executes JavaScript code outside a web browser. Node.js runs on various platforms (Windows, Linux, Unix, Mac OS X, etc.).

A common task for a web server can be to open a file on the server and return the content to the client [1].

PHP or **ASP** handling a file request:

1. Sends the task to the computer's file system.
2. Waits while the file system opens and reads the file.
3. Returns the content to the client.
4. Ready to handle the next request.

Node.js handling a file request:

1. Sends the task to the computer's file system.
2. Ready to handle the next request.
3. When the file system has opened and read the file, the server returns the content to the client.

Node.js eliminates the waiting, and simply continues with the next request.

Node.js runs single-threaded, non-blocking, asynchronously programming, which is very memory efficient [1].

Node.js Features

- generating dynamic page content;
- Node.js can create, open, read, write, delete, and close files on the server;
- collecting form data;
- Node.js can add, delete, modify data in database.

Node.js File

Node.js files contain tasks that will be executed on certain events. A typical event is someone trying to access a port on the server. Node.js files must be initiated on the server before having any effect. Node.js files have extension ".js".

Download Node.js

The official Node.js website has installation instructions for Node.js [2]:

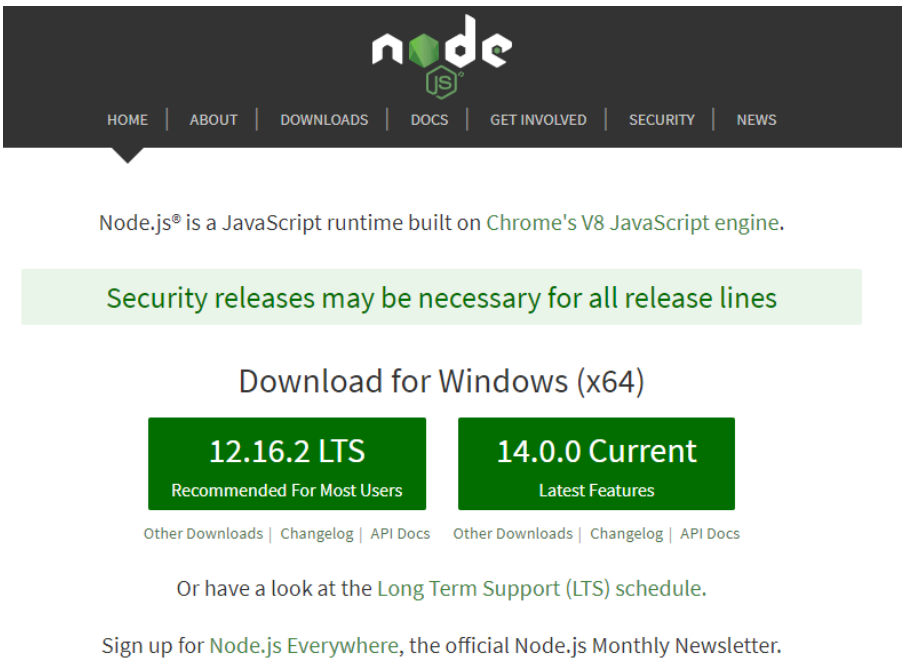
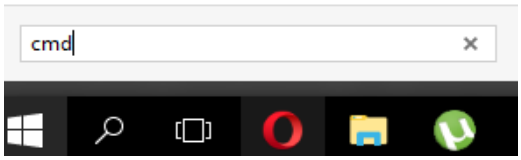
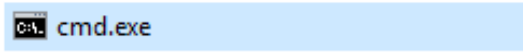
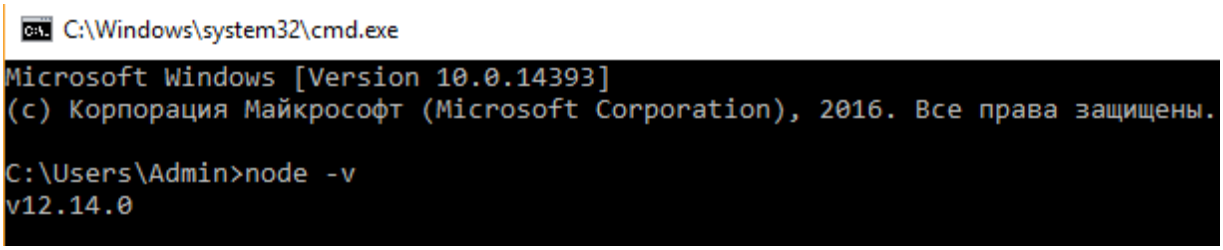


Fig. 4.1. The official Node.js website

After a successful installation, we can enter the **node -v** command on the command line / terminal and the current version of Node.js will be displayed:



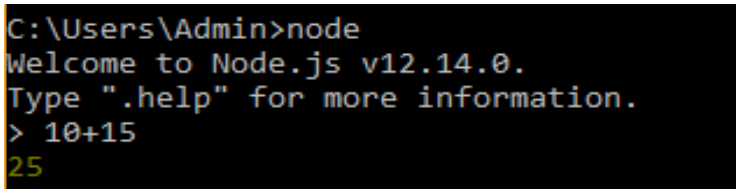




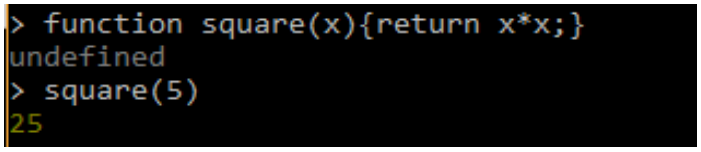
Versions of node.js for other operating systems along with the source code can be found at [3].

REPL

After installing Node.js, a tool like **REPL** (Read Eval Print Loop) becomes available to us. REPL provides the ability to run JavaScript expressions on the command line or terminal. Run the command line (on Windows) or the terminal (on OS X or Linux) and enter the **node** command. After entering this command, we can execute various JavaScript expressions:



We can define your functions and then call them, for example, squaring a number:



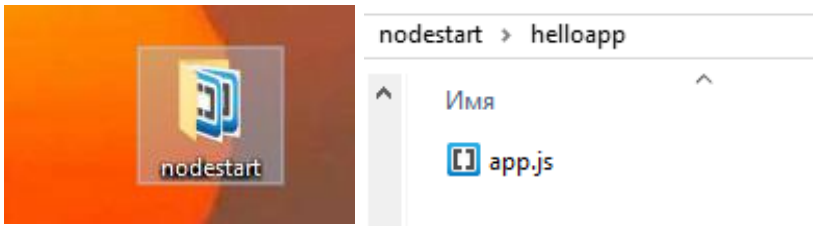
File execution

Instead of entering all the code directly into the console, it is more convenient to put it into an external file. For example, create a new directory on hard drive **C:\nodestart\helloapp**, into which we will place a new **app.js** file with the following code:

```
{ }app.js
1   console.log("Hello world");
```

At the command prompt, use the **cd** command (change directory) to navigate to the **helloapp** directory, and then run the command: `>node app`

This command will execute the code from the **app.js** file:




```
C:\Users\Admin>cd Desktop

C:\Users\Admin\Desktop>cd nodestart

C:\Users\Admin\Desktop\nodestart>cd helloapp

C:\Users\Admin\Desktop\nodestart\helloapp>node app
Hello world
```

We will write the first simplest application. Almost all standard JavaScript language constructs can be used to create applications. An exception is working with the DOM, since the application will run on the server, and not in the browser, so the DOM and objects such as window or document will not be available to us in this case.

To do this, first create a directory for the application on the hard drive. For example, we created the directory **C:\node\helloapp**. In this directory, we create the file **myfirst.js** to display "Hello World" in a Web browser with following code:

myfirst.js

```
1  const http = require("http");
2  http.createServer(function(request,response){
3
4      response.end("Hello World");
5
6  }).listen(3000, "127.0.0.1",function(){
7      console.log("Server started listening requests on port 3000");
8  });
```

On the first line we get the **http** module, which is necessary to create a Web server. This is a built-in module, and we need to use the **require ()** function to load it:

```
1  const http = require("http");
```

Next, using the **createServer ()** method, a new server is created to listen for incoming connections and process requests. This method takes a **function** that has two parameters. The first request parameter stores all the information about the **request**, and the second response parameter is used to send a **response**. In this case, the answer is a simple string "Hello World!" and is sent using the **response.end ()** method. But the **http.createServer ()** method only creates the server. In order for the server to start listening for incoming connections, it must call the listen method:

```
}).listen(3000, "127.0.0.1",function(){
    console.log("Server started listening requests on port 3000");
});
```

This method takes three parameters. The first parameter indicates the local port on which the server starts. The second parameter indicates the local address. That is, in this case, the server will start at the address **127.0.0.1** or localhost on port **3000**.

The third parameter represents a function that starts when listening to connections starts. Here, this function simply displays a diagnostic message to the console.

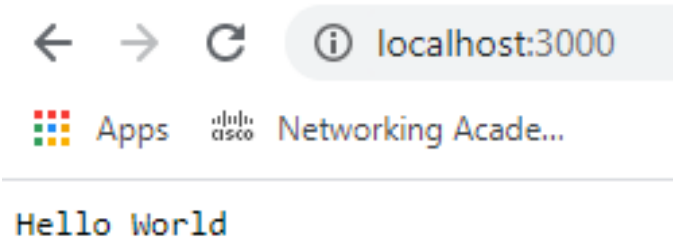
“Localhost” refers to the computer that a certain program is running on. For example, if we’re running a program on own computer (like a web browser or local web development environment), then this computer is the “localhost”.

In the most simple terms, we can essentially think of localhost as meaning “this computer”. Just remember that “this computer” applies to the program, not necessarily to the computer that you’re physically using. On a more technical level, localhost typically resolves to the IP address 127.0.0.1, which is known as the loopback address. Because of its importance, the term “localhost” is a reserved domain name. That means that, in order to avoid confusion, it’s impossible to register a domain name that contains “localhost” in the top-level or second-level domain name.

Now run the server. To do this, open a terminal (in OS X or Linux) or a command line (in Windows). Using the **cd** command, we will go to the application directory and using **node myfirst** command we run server:

```
C:\Users\Admin\Desktop\nodestart\helloapp>node myfirst
Server started listening requests on port 3000
```

Next, open the browser and enter the address http://localhost:3000/ and we will see the message that was sent in the **response.end ()** method.



Modules

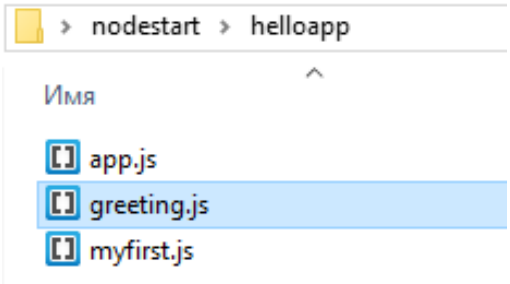
Node.js uses a modular system. This means that the built-in functionality is divided into separate packages or modules. A module is a block of code that can be reused in other modules. If necessary, we can connect the modules we need. We can find out what built-in modules in node.js are and what functionality they provide from the documentation [4]. To load modules, the **require ()** function is used, into which the module name is passed. For example, to receive and process a request, the **http** module was needed:

```
1  const http = require("http");
```

After receiving the module, we will be able to use all the functionality defined in it, which again can be found in the documentation [5]. Similarly, we can load and use other built-in modules. For example, we use the os module, which provides information about the environment and the operating system (OS):

```
const os = require("os");
// we will get the name of the current user
let userName = os.userInfo().username;

console.log(userName);
```

We are not limited to built-in modules and, if necessary, we can create our own. We created a project which consisted of the **myfirst.js** file, in which a server was processed that processed requests. Add a new **greeting.js** file to the same directory



and define the following code in it:

```
1   console.log("greeting module");
```

In the **myfirst.js** file, connect our module:

```
myfirst.js  { }greeting.js

1   const greeting = require("./greeting");
```

In contrast to the built-in modules, we connect our modules, we must pass the relative path with the file name to the require function. Launch the application. The line that is defined in the **greeting.js** file is displayed on the console:

```
C:\Users\Admin\Desktop\nodestart\helloapp>node myfirst
greeting module
Server started listening requests on port 3000
```

Now modify the **greeting.js** file:

```
let currentDate = new Date();
module.exports.date = currentDate;

module.exports.getMessage = function(name){
    let hour = currentDate.getHours();
    if(hour > 16)
        return "Good evening, " + name;
    else if(hour > 10)
        return "Good afternoon, " + name;
    else
        return "Good morning, " + name;
}
```

The **currentDat**e variable is defined here. However, from outside it is not available. It is available only within this module. For any variables or module functions to be available, we must define them in the **module.exports** object. The **module.exports** object is what the **require ()** function returns when the module is received. In general, the module object represents a link to the current module, and its exports property defines all the properties and methods of the module that can be exported and used in other modules. For more details on determining module loading and all its functions,

see https://github.com/nodejs/node/blob/master/lib/module.js. In particular, the date property and the **getMessage** method, which takes some parameter, are defined here.

Next, modify the **myfirst.js** file:

```
const os = require("os");
const greeting = require("./greeting");

// we will get the name of the current user
let userName = os.userInfo().username;


console.log(`Request Date: ${greeting.date}`);
console.log(greeting.getMessage(userName));
```

All exported methods and module properties are available by name: **greeting.date** and **greeting.getMessage ().** Restart the application:

```
C:\Users\Admin\Desktop\nodestart\helloapp>node myfirst
Request Date: Fri Apr 24 2020 17:18:54 GMT+0300 (GMT+03:00)
Good evening, Admin
```

Defining constructors and objects in a module

In addition to defining simple functions or properties in a module, complex objects or constructor functions can be defined, which are then used to create objects. So, add a new **user.js** file to the project folder:

greeting.js | myfirst.js | app.js | user.js

```
function User(name, work){

    this.name = name;
    this.work = work;
    this.displayInfo = function(){

        }
}
User.prototype.sayHi = function() {
    console.log(`Hello, my name is ${this.name}. I work at ${this.work}`);
};

module.exports = User;

```

The standard **constructor function User** is defined here, which takes two parameters (name, work). Moreover, the entire module now points to this constructor function:

```
13  module.exports = User;
```

Connect and use this module in the **myfirst.js** file:

```
const User = require("./user.js");
 let teacher = new User("Liubov", "KPI");
teacher.sayHi();
```

So, we have three files:

greeting.js | { }myfirst.js | { }user.js

```
let currentDate = new Date();
module.exports.date = currentDate;

module.exports.getMessage = function(name){
    let hour = currentDate.getHours();
    if(hour > 16)
        return "Good evening, " + name;
    else if(hour > 10)
        return "Good afternoon, " + name;
    else
        return "Good morning, " + name;
}

```

greeting.js | { }myfirst.js | { }user.js

```
const os = require("os");
const greeting = require("./greeting");

// we will get the name of the current user
let userName = os.userInfo().username;

 console.log(`Request Date: ${greeting.date}`);
console.log(greeting.getMessage(userName));

const User = require("./user.js");
 let teacher = new User("Liubov", "KPI");
teacher.sayHi();

```

greeting.js { }myfirst.js { }user.js

```
function User(name, work){

    this.name = name;
    this.work = work;
    this.displayInfo = function(){

        }
}
User.prototype.sayHi = function() {
    console.log(`Hello, my name is ${this.name}. I work at ${this.work}`);
};

module.exports = User;

```

and the result of starting the **myfirst.js** application from the command line:

```
C:\Users\Admin\Desktop\nodestart\helloapp>node myfirst
Request Date: Fri Apr 24 2020 17:53:08 GMT+0300 (GMT+03:00)
Good evening, Admin
Hello, my name is Liubov. I work at KPI
```

Global object and global variables

Node.js provides a special global object that provides access to global variables, which are accessible from each application module, and functions. An example analog of this object in browser JavaScript is the **window** object. All available global objects can be found in [6].

For example, create the following **greeting.js** module:

```
let currentDate = new Date();

global.date = currentDate;

module.exports.getMessage = function(){
    let hour = currentDate.getHours();
    if(hour >18)
        return "Good evening, " + global.name;
    else if(hour >12)
        return "Good afternoon, " + name;
    else
        return "Good morning, " + name;
}
```

Firstly, the global variable date is set: **global.date = currentDate;**

Secondly, in the module we get the global variable name, which will be set from outside. At the same time, we can access the global variable name through the global: **global.name** object, or simply through the **name** name, since the variable is global. Define the following **myfirst.js** application file:

```
const greeting = require("./greeting");
 global.name = "Liubov";

 console.log(date);
console.log(greeting.getMessage());
```

We set the global variable name, which we get in the **greeting.js** module and also print the global variable date to the console. Moreover, all global functions and objects, for example, console, are also available inside global, so we can write **global.console.log ()** and just **console.log ().**

Run the **myfirst.js**. file:

```
C:\Users\Admin\Desktop\nodestart\helloapp>node myfirst
2020-04-24T16:36:46.612Z
Good evening, Liubov
```

Passing parameters to the application

When starting the application from the terminal / command line, we can pass parameters to it. To obtain parameters in the application code, the **process.argv** array is used. This is similar to how in C / C ++ / C # / Java, a set of arguments is passed to the main function as a string array.

The first element of this array always points to the path to the **node.exe** file that the application calls. The second element of the array always points to the path to the application file that is running. For example, define the following **appparam.js** file:

appparam.js

```
let nodePath = process.argv[0];
let appPath = process.argv[1];
let name = process.argv[2];
let age = process.argv[3];

console.log("nodePath: " + nodePath);
console.log("appPath: " + appPath);
console.log();
console.log("name: " + name);
console.log("age: " + age);
```

In this case, we expect two parameters to be passed to the application: name and age. Now run the application using the following command:

```
node appparam Tom 23
```

In this case, “Tom” and “23” are those values that are placed respectively in process.argv [2] and process.argv [3]:

```
C:\Users\Admin\Desktop\nodestart\helloapp>node appparam Tom 23
nodePath: C:\Program Files\nodejs\node.exe
appPath: C:\Users\Admin\Desktop\nodestart\helloapp\appparam
```

NPM

In addition to the built-in and custom Node.js modules, there is a huge layer of various libraries and frameworks, various utilities that are created by third-party manufacturers and which can also be used in the project, for example, **express, grunt, gulp**. And they are also available to us as part of Node.js. To make it more convenient to work with all third-party solutions, they are distributed in the form of packages. A package essentially represents a set of functionalities.

To automate the installation and updating of packages, as a rule, a package management system or managers are used. Directly in Node.js, the **NPM** (Node Package Manager) is used for this purpose. NPM is installed by default with Node.js,

so nothing needs to be installed. But we can update the installed version to the latest. To do this, run the following command in the command line / terminal:

```
npm install npm@latest -g
```

To find out the current version of npm, enter the following command at the command line / terminal:

```
npm -v
```

The **npm** manager is important in that it is easy to manage packages with it. For example, create a new **modulesapp** folder on hard drive (the folder will be located on the path **C:\nodestart\modulesapp**).

Package.json file. Installation of modules

For more convenient configuration and application package management, npm uses the **package.json** configuration file. So, add the new **package.json** file to the **modulesapp** project folder:

```
{
  "name": "modulesapp",
  "version": "1.0.0"
}
```

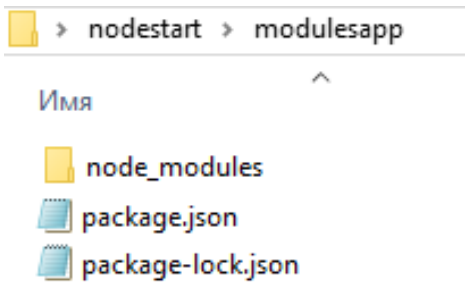
Only two sections are defined here: the project name is **modulesapp** and its version is 1.0.0. This is the minimum required definition for the **package.json** file. This file may include many more sections. See the documentation for more details. Next, install **Express** in the project.

Express is a lightweight web framework to simplify working with Node.js. To install Express functionality in a project, we first go to the project folder using the **cd** command. Then enter the command:

```
npm install express
```

```
C:\Windows\System32>cd C:\Users\Admin\Desktop\nodestart\modulesapp

C:\Users\Admin\Desktop\nodestart\modulesapp>npm install express
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN modulesapp@1.0.0 No description
npm WARN modulesapp@1.0.0 No repository field.
npm WARN modulesapp@1.0.0 No license field.

+ express@4.17.1
added 50 packages from 37 contributors and audited 126 packages in 8.875s
found 0 vulnerabilities
```

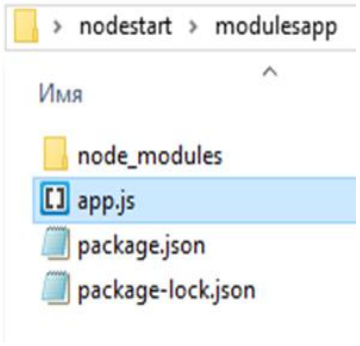
After installing express, a subfolder **node_modules** will appear in the **modulesapp** project folder, in which all installed external modules will be stored.



In particular, in the subdirectory **node_modules/express** the files of the **Express** framework will be located. After executing the command, if we open the **package.json** file, we will see information about the package:

```
{
  "name": "modulesapp",
  "version": "1.0.0",
  "dependencies": {
    "express": "^4.17.1"
  }
}
```

Information about all the added packages that are used when the application is running is added to the **dependencies** section. We use the added express package and for this we define the file of the simplest server. To do this, add the new **app.js** file to the root folder of the **modulesapp** project:



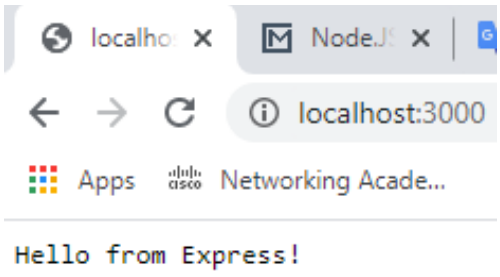

```
{ }app.js

1   // we get the Express module
2   const express = require("express");
3   // create an application
4   const app = express();
5
6   // set the handler for the route "/"
7   app.get("/", function(request, response){
8
9       response.end("Hello from Express!");
10  });
11  // start listening for connections on port 3000
12  app.listen(3000);
```

The first line receives the installed express module, and the second creates the application object. In Express we can associate request processing with specific routes. For example, "/" represents the main page or root route.

To process the request, the **app.get ()** function is called. The first parameter of the function is the route, and the second is the function that will process the request along this route. And for the server to start listening for connections, we need to call the **app.listen ()** method, to which the port number is transmitted.

Start the server with the **node app.js** command:

```
C:\Users\Admin\Desktop\nodestart\modulesapp>node app
```



If in the future we no longer need express, then we can delete it with the following command:

```
npm uninstall express
```

Adding Multiple Packages

The **package.json** file plays a big role and can make it easier to work with packages in various situations. For example, we plan to use many packages. But entering the appropriate command to install each package in the console is not very convenient. In this case, we can determine all the packages in the **package.json** file and then install them with one command.

For example, we modify the **package.json** file as follows:

```
{
  "name": "modulesapp",
  "version": "1.0.0",
  "dependencies": {
    "express": "^4.17.1",
    "react": "^16.9.0",
    "react-dom": "^16.9.0"
  }
}
```

Here are the definitions of two packages that represent the **React library**. Then, to download all the packages, run the command

```
npm install
```

This command will take the definition of all packages from the dependencies sections and load them into the project. If the package with the desired version already has a project, as in this case express, then it will not be loaded on the new one.

devDependencies

In addition to the packages that are used in the application when it is running and is in working condition, for example, express, that is, in the "production" state, there are also packages that are used to develop the application and test it. Such packages are usually added to another **devDependencies** section. For example, load the **jasmine-node package** into the project, which is used to test the application:

```
npm install jasmine-node --save-dev
```

The **--save-dev** flag indicates that information about the package should be stored in the devDependencies section of the **package.json** file:

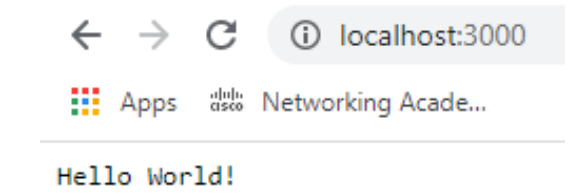
```
{
  "name": "modulesapp",
  "version": "1.0.0",
  "dependencies": {
    "express": "^4.17.1",
    "react": "^16.9.0",
    "react-dom": "^16.9.0"
  },
  "devDependencies": {
    "jasmine-node": "^3.0.0"
  }
}
```

Nodemon

During the development process, you may need to make changes to an already running project. Suppose we have the following code defined in the **app.js** file:

```
const http = require("http");

let message = "Hello World";
http.createServer(function(request,response){

    console.log(message);
    response.end(message);

}).listen(3000, "127.0.0.1",()=>{
    console.log("Server started listening for requests");
});
```

We start the server using the **node app.js** command, and when the user contacts **http://localhost:3000/**, the user's browser displays the string "Hello World!". At the same time, the line is displayed on the console. If we refresh the page three times in the browser, the line will be displayed three times in the console.




```
C:\Users\Admin\Desktop\nodestart\modulesapp>node app
Server started listening for requests
Hello World!
Hello World!
Hello World!
```

At the same time, the server continues to be running. And if we change the message variable in the **app.js** file, then this will **not affect** the server in any way, and it will continue to return the string "Hello World!" to the client.

In this case, we must restart the server. However, this is not very convenient, especially when it is often necessary to make various changes, to test the execution. And in this case, a special nodemon tool can help us.

Install **nodemon** into the project using the following command:

```
npm install nodemon -g
```

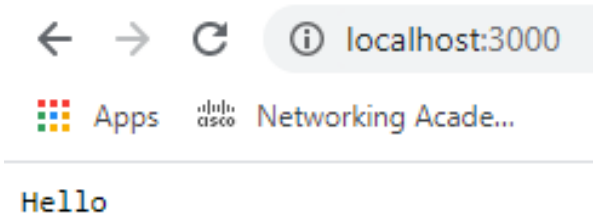
The **-g** flag represents an abbreviation for global and allows to set the nodemon dependency globally for all projects on a given local machine.

After installation, run the **app.js** file using the following command:

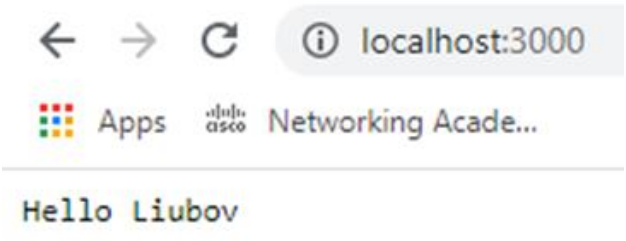
```
nodemon app.js
```

```
C:\Users\Admin\Desktop\nodestart\helloapp>nodemon app
[nodemon] 2.0.3
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node app.js`
Server started listening for requests
```

And if suddenly after starting the server we change its code, for example, change the message variable with "Hello World!" to "Hello!" then “Hello Liubov!” (ctrl+S) in our code, then refresh the page, the server will automatically be restarted:




```
const http = require("http");

let message = "Hello Liubov";
http.createServer(function(request,response){

    console.log(message);
    response.end(message);

}).listen(3000, "127.0.0.1",()=>{
    console.log("Server started listening for requests");
});
```




```
Server started listening for requests
Hello
[nodemon] restarting due to changes...
[nodemon] restarting due to changes...
[nodemon] starting `node app.js`
Server started listening for requests
Hello Liubov
```

Node.js File System Module

Node.js as a File Server

The Node.js file system module allows to work with the file system on your computer. To include the File System module, use the **require()** method:

var fs = require('fs');

Common use for the File System module [8]:

- Read files
- Create files
- Update files
- Delete files
- Rename files
- Read Files.

The **fs.readFile()** method is used to read files on your computer.

Assume we have the following HTML file (located in the same folder as Node.js):

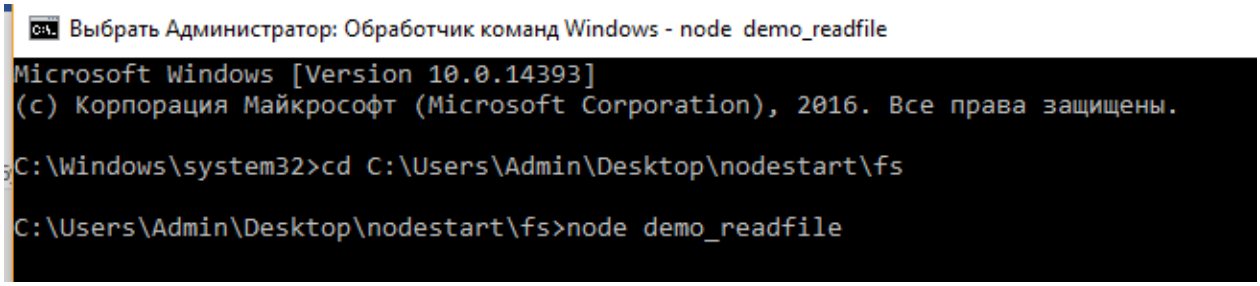
demofile1.html

```
<html>
<body>
<h1>My Header</h1>
<p>My paragraph.</p>
</body>
</html>
```

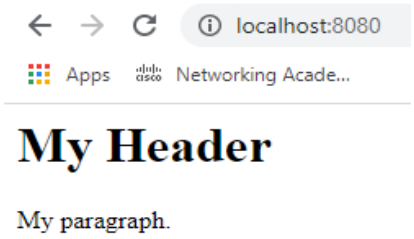
Create a **demo_readfile.js** file that reads the HTML file, and return the content:

```
var http = require('http');
var fs = require('fs');
http.createServer(function (req, res) {
  fs.readFile('demofile1.html', function(err, data) {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write(data);
    return res.end();
  });
}).listen(8080);
```

Initiate **demo_readfile.js**:



If you have followed the same steps on your computer, you will see the same result as the example: **http://localhost:8080**



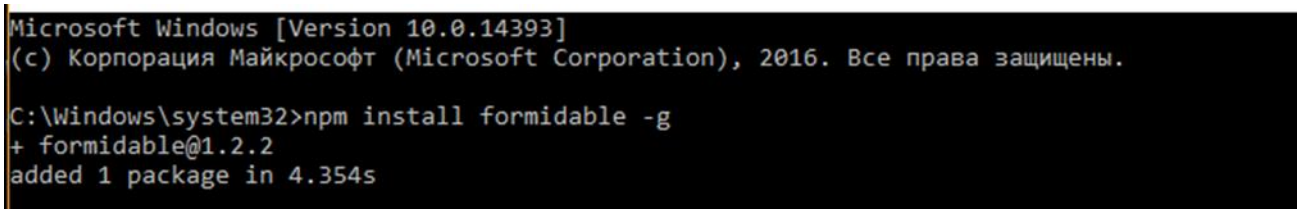
Node.js Upload Files

The Formidable Module

There is a very good module for working with file uploads, called "Formidable".

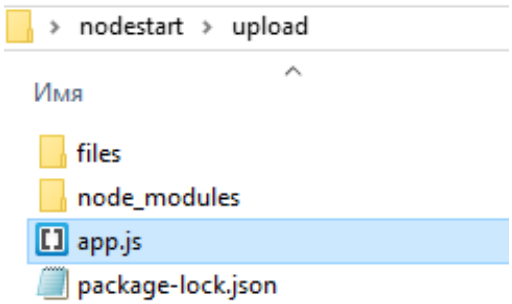
The Formidable module can be downloaded and installed using NPM:

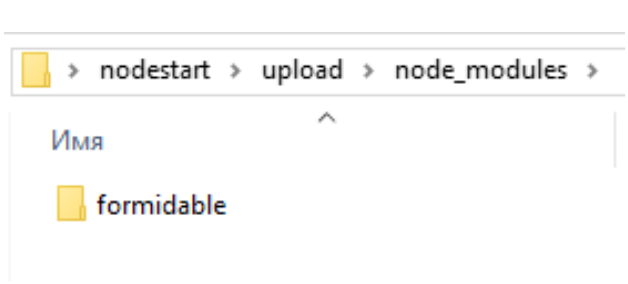
>npm install formidable

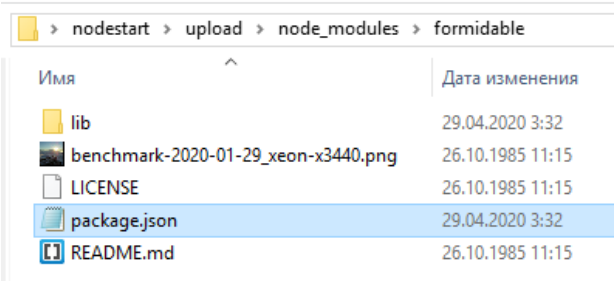


After you have downloaded the Formidable module, you can include the module in any application:

var formidable = require('formidable');







Upload Files

Now you are ready to make a web page in Node.js that lets the user upload files to your computer [9]:

Step 1: Create an Upload Form

Create a Node.js file that writes an HTML form, with an upload field:

This code will produce an HTML form:

```
  var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write('<form action="fileupload" method="post"
enctype="multipart/form-data">');
  res.write('<input type="file" name="filetoupload"><br>');
  res.write('<input type="submit">');
  res.write('</form>');
  return res.end();
}).listen(8080);
```

Step 2: Parse the Uploaded File

Include the Formidable module to be able to parse the uploaded file once it reaches the server. When the file is uploaded and parsed, it gets placed on a temporary folder on your computer. The file will be uploaded, and placed on a temporary folder:

```
  var http = require('http');
var formidable = require('formidable');
http.createServer(function (req, res) {
  if (req.url == '/fileupload') {
    var form = new formidable.IncomingForm();
    form.parse(req, function (err, fields, files) {
      res.write('File uploaded');
      res.end();
    });
  } else {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write('<form action="fileupload" method="post" enctype="multipart/form-
data">');
    res.write('<input type="file" name="filetoupload"><br>');
    res.write('<input type="submit">');
    res.write('</form>');
    return res.end();
  }
}).listen(8080);
```

Step 3: Save the File

When a file is successfully uploaded to the server, it is placed on a temporary folder.

The path to this directory can be found in the "files" object, passed as the third argument in the **parse()** method's callback function.

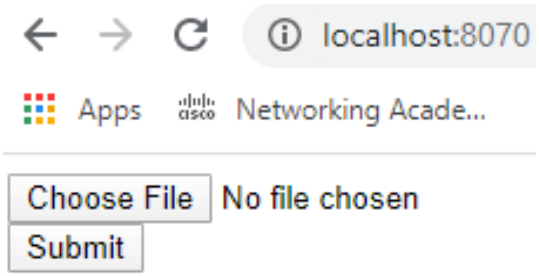
To move the file to the folder of your choice, use the File System module, and rename the file. Include the **fs** module, and move the file to the current folder:

```
  var http = require('http');
var formidable = require('formidable');
var fs = require('fs');
http.createServer(function (req, res) {
  if (req.url == '/fileupload') {
    var form = new formidable.IncomingForm();
    form.parse(req, function (err, fields, files) {
      var oldpath = files.filetoupload.path;
      var newpath = 'C:/Users/Your Name/' + files.filetoupload.name;
      fs.rename(oldpath, newpath, function (err) {
```

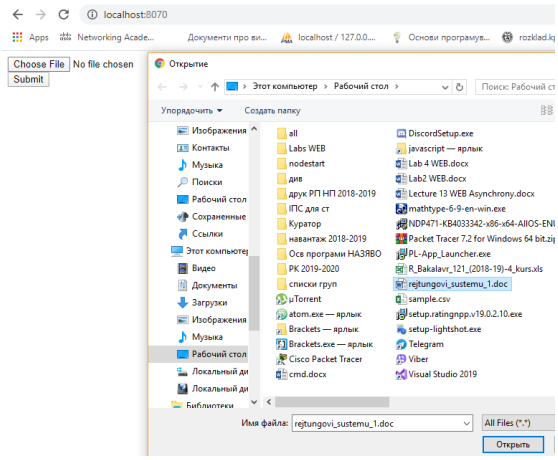
```
        if (err) throw err;
        res.write('File uploaded and moved!');
        res.end();
      });
 });
  } else {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write('<form action="fileupload" method="post" enctype="multipart/form-
data">');
    res.write('<input type="file" name="filetoupload"><br>');
    res.write('<input type="submit">');
    res.write('</form>');
    return res.end();
  }
}).listen(8080);
```

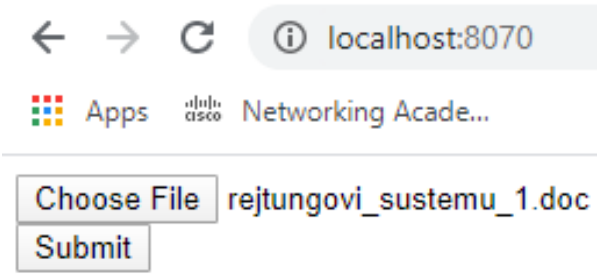
```
var http = require('http');
var formidable = require('formidable');
var fs = require('fs');

http.createServer(function (req, res) {
  if (req.url == '/fileupload') {
    var form = new formidable.IncomingForm();
    form.parse(req, function (err, fields, files) {
      var oldpath = files.filetoupload.path;
      var newpath = 'C:/Users/Admin/Desktop/nodestart/upload/files/' + files.filetoupload.name;
      fs.rename(oldpath, newpath, function (err) {
        if (err) throw err;
        res.write('File uploaded and moved!');
        res.end();
      });
  });
  } else {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write('<form action="fileupload" method="post" enctype="multipart/form-data">');
    res.write('<input type="file" name="filetoupload"><br>');
    res.write('<input type="submit">');
    res.write('</form>');
    return res.end();
  }
}).listen(8070);
```

In this example, we use, for example, a 8070 port for a localhost. So we will use the address http://localhost:8070/

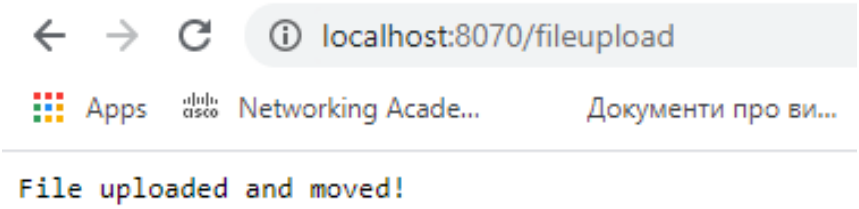


Press the **Choose File** button and select File:





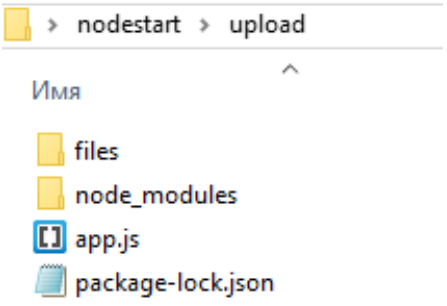
Press the **Submit** button and we have the address http://localhost:8070/fileupload



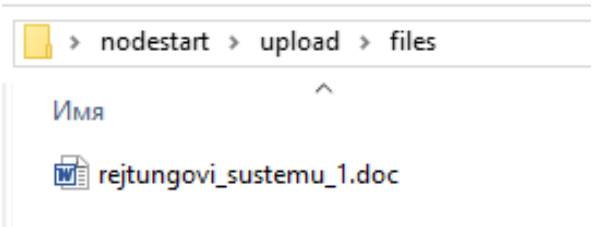
because we set the path with this name for URL:

```
if (req.url == '/fileupload') {
```

Open the folder of our project:



Now the file we selected is located in the **files** folder:



Tasks for laboratory work 4

1. Install Node.js and all the necessary components that are described in the tutorial.

2. Create a simple server and a Node.js file named "**myfirst.js**" to display "Hello [*your name*]!" in a web browser.

3. Create a Node.js server and a **readfile.js** file that reads the HTML file (“My hobbies” Web page from Lab 1), and return the content to user.

4. Make a web page in Node.js that lets the user upload files to your computer. Show the process of downloading files to a teacher.

Report requirements for laboratory work

The report should include:

1. Title page.

2. Tasks for laboratory work.

3. Description of development steps. This section consists of a sequential description of the steps performed according to the instructions for laboratory work.

4. Conclusions.

Questions for self-assessment

1. What is Node.js?

2. What is the difference between Node.js request processing and PHP or ASP?

3. What is REPL?

4. What is the module for Node.js?

5. What is NPM?

6. What is Express?

7. What is **package.json** file used for?

8. What is Nodemon used for?

9. What is a localhost and what is it used for?

10. What Node.js modules for the file system do you know?

11. How to read a file from the server and display the contents to the user?

12. What module is used to upload files to the server by the user?

References

11. Node.js Introduction https://www.w3schools.com/nodejs/nodejs_intro.asp
12. Node.js https://nodejs.org
13. Downloads https://nodejs.org/en/download/
14. Node.js Documentation https://nodejs.org/api/
15. Node.js Documentation. HTTP https://nodejs.org/api/http.html
16. Global Objects https://nodejs.org/api/globals.html
17. Creating a package.json file https://docs.npmjs.com/creating-a-package-json-file
18. Node.js File System Module https://www.w3schools.com/nodejs/nodejs_filesystem.asp
19. Node.js Upload Files https://www.w3schools.com/nodejs/nodejs_uploadfiles.asp
20. Nodemon https://metanit.com/web/nodejs/2.6.php

LABORATORY WORK 5.
CREATING API WITH NODE.JS AND EXPRESS.
GEOLOCATION API. USING LEAFLET LIBRARY

Purpose: to get practical skills in working with the Node.js Express web application framework, creating own API.

Theory and methodological instructions

API (application programming interface, API) is a description of the methods (a set of classes, procedures, functions, structures or constants) that one computer program can interact with another program. It is usually included in the description of the Internet protocol, software framework (framework), or the standard for calling operating system functions. It is often implemented by a separate software library or operating system service. Used by programmers to write all kinds of applications.

Using Express and Node.js, we can implement a full REST-style API for user interaction. The REST architecture involves the use of the following methods or types of HTTP requests to interact with the server:

• **GET** - getting a resource (for example, a web page);

• **POST** - creating a resource (for example, filling out a web form and sending data to the server);

• **PUT** - resource update;

• **DELETE** - delete a resource.

Often, the REST style is especially convenient when creating all sorts of Single Page Application (SPA), which often use special JavaScript frameworks such as Angular or React.

Consider how to create your own API. For a new project, create a new folder, which will be called **webapp**.

Immediately define the **package.json** file in the project (or we can use command **npm install express**):

```
{
  "name": "webapp",
  "version": "1.0.0",
  "dependencies": {
    "body-parser": "^1.16.0",
    "express": "^4.14.0"
  }
}
```

We also added **nodemon** globally to our project.

```
  "name": "webapp",
  "version": "1.0.0",
  "dependencies": {
    "body-parser": "^1.16.0",
    "express": "^4.17.1",
    "nodemon": "^1.19.2"
  }
```

In this case, we will create an experimental project that will store data in a json file and which is designed to simply show the creation of the API in Node.js in the REST style. In the meantime, add a new users.json file to the project folder with the following contents:

```
[{
    "id":1,
    "name":"Tom",
    "age":24
},
{
    "id":2,
    "name":"Bob",
    "age":27
},
{
    "id":3,
    "name":"Alice",
    "age":"23"
}]
```

For reading and writing to this file, we will use the built-in **fs** module. To process requests, we define the following **app.js** file in the project:

```
var express = require("express");
var bodyParser = require("body-parser");
var fs = require("fs");
 var app = express();
var jsonParser = bodyParser.json();
 app.use(express.static(__dirname + "/public"));
// getting a list of data
app.get("/api/users", function(req, res){
        var content = fs.readFileSync("users.json", "utf8");
    var users = JSON.parse(content);
    res.send(users);
});
// getting one user by id
app.get("/api/users/:id", function(req, res){
        var id = req.params.id; // get id
    var content = fs.readFileSync("users.json", "utf8");
    var users = JSON.parse(content);
    var user = null;
    // we find in the user array by id
    for(var i=0; i<users.length; i++){
        if(users[i].id==id){
            user = users[i];
            break;
        }
    }
    // send user
```

```
    if(user){
        res.send(user);
    }
    else{
        res.status(404).send();
    }
});
// receiving sent data
app.post("/api/users", jsonParser, function (req, res) {
        if(!req.body) return res.sendStatus(400);
        var userName = req.body.name;
    var userAge = req.body.age;
    var user = {name: userName, age: userAge};
        var data = fs.readFileSync("users.json", "utf8");
    var users = JSON.parse(data);
        // we find the maximum id
    var id = Math.max.apply(Math,users.map(function(o){return o.id;}))
    // increase it by one
    user.id = id+1;
    // add the user to the array
    users.push(user);
    var data = JSON.stringify(users);
    // overwrite the file with the new data
    fs.writeFileSync("users.json", data);
    res.send(user);
});
 // delete user by id
app.delete("/api/users/:id", function(req, res){
```

```
        var id = req.params.id;
    var data = fs.readFileSync("users.json", "utf8");
    var users = JSON.parse(data);
    var index = -1;
    // find the user index in the array
    for(var i=0; i<users.length; i++){
        if(users[i].id==id){
            index=i;
            break;
        }
    }
    if(index > -1){
        // delete the user from the array by index
        var user = users.splice(index, 1)[0];
        var data = JSON.stringify(users);
        fs.writeFileSync("users.json", data);
        // send remote user
        res.send(user);
    }
    else{
        res.status(404).send();
    }
});
// user change
app.put("/api/users", jsonParser, function(req, res){
         if(!req.body) return res.sendStatus(400);
        var userId = req.body.id;
    var userName = req.body.name;
```

```
    var userAge = req.body.age;
        var data = fs.readFileSync("users.json", "utf8");
    var users = JSON.parse(data);
    var user;
    for(var i=0; i<users.length; i++){
       if(users[i].id==userId){
          user = users[i];
          break;
       }
    }
    // change user data
    if(user){
       user.age = userAge;
       user.name = userName;
       var data = JSON.stringify(users);
       fs.writeFileSync("users.json", data);
       res.send(user);
    }
    else{
       res.status(404).send(user);      }
});
  app.listen(3000, function(){
    console.log("The server is waiting for a connection ...");
});
```

Five methods are defined for processing requests for each type of request: **app.get() / app.post() / app.delete() / app.put()**.

When the application receives a request of type GET at the address "api / users", the following method works:

```
app.get("/api/users", function(req, res){
        var content = fs.readFileSync("users.json", "utf8");
    var users = JSON.parse(content);
    res.send(users);
});
```

As a result of processing, we must send an array of users that we read from the file. To simplify the application code within the framework of this project, the synchronous methods **fs.readFileSync()** / **fs.writeFileSync()** are used to read / write the file. But in reality, as a rule, work with data will go through the database, and then we will consider all this with the example of MongoDB (laboratory work 6).

To get data from a file using the **fs.readFileSync()** method, we read the data into a string, which we parse into an array of objects using the **JSON.parse()** function. And at the end, we send the received data to the client using the **res.send()** method.

Another **app.get()** method works similarly, which fires when the user id is specified in the address:

```
app.get("/api/users/:id", function(req, res){
        var id = req.params.id; // we get id
    var content = fs.readFileSync("users.json", "utf8");
    var users = JSON.parse(content);
    var user = null;
    // we find in the user array by id
    for(var i=0; i<users.length; i++){
        if(users[i].id==id){
            user = users[i];
            break;
        }
    }
    // send user
    if(user){
```

```
        res.send(user);
    }
    else{
        res.status(404).send();
    }  });
```

In this case, we need to find the desired user by **id** in the array, and if user was not found, return the status code 404: **res.status(404).send().**

When receiving a request using the POST method, we need to use the **jsonParser** parser to extract data from the request:

```
// receive sent data

app.post("/api/users", jsonParser, function (req, res) {
        if(!req.body) return res.sendStatus(400);
        var userName = req.body.name;
    var userAge = req.body.age;
    var user = {name: userName, age: userAge};
        var data = fs.readFileSync("users.json", "utf8");
    var users = JSON.parse(data);
        // we find the maximum id
    var id = Math.max.apply(Math,users.map(function(o){return o.id;}))
    // increase it by one
    user.id = id+1;
    // add the user to the array
    users.push(user);
    var data = JSON.stringify(users);
    // overwrite the file with the new data
    fs.writeFileSync("users.json", data);
    res.send(user);
});
```

After receiving the data, we need to create a new object and add it to the array of objects. To do this, read the data from the file, add a new object to the array and overwrite the file with the updated data.

When deleting, we perform similar actions, only now we extract the object to be deleted from the array and again overwrite the file:

```
app.delete("/api/users/:id", function(req, res){
      var id = req.params.id;
    var data = fs.readFileSync("users.json", "utf8");
    var users = JSON.parse(data);
    var index = -1;
    // find the user index in the array
    for(var i=0; i<users.length; i++){
        if(users[i].id==id){
            index=i;
            break;
        }
    }
    if(index > -1){
        // delete the user from the array by index
        var user = users.splice(index, 1)[0];
        var data = JSON.stringify(users);
        fs.writeFileSync("users.json", data);
        // send remote user
        res.send(user);
    }
    else{
        res.status(404).send();
    }   });
```

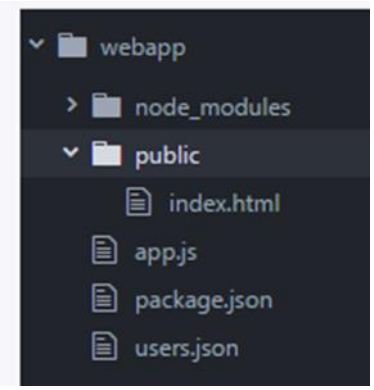
If the object is not found, return the status code 404.

If the application receives a PUT request, then it is processed by the **app.put()** method, in which using **jsonParser** we get the changed data:

```
app.put("/api/users", jsonParser, function(req, res){
        if(!req.body) return res.sendStatus(400);
       var userId = req.body.id;
    var userName = req.body.name;
    var userAge = req.body.age;
        var data = fs.readFileSync("users.json", "utf8");
    var users = JSON.parse(data);
    var user;
    for(var i=0; i<users.length; i++){
        if(users[i].id==userId){
            user = users[i];
            break;
        }
    }
    if(user){
        user.age = userAge;
        user.name = userName;
        var data = JSON.stringify(users);
        fs.writeFileSync("users.json", data);
        res.send(user);
    }
    else{
        res.status(404).send(user);
    }
});
```

Here, also to search for a mutable object, we read the data from the file, find the mutable user by id, change his properties and save the updated data to a file.

Thus, we have defined the simplest API. Now add the client code. So, as it is set in the code, Express uses the **public** folder to store static files, so we will create a similar folder in the project. In this folder, define a new **index.html** file that will act as a client. As a result, the entire project will look like this:



Next, define the following code in the **index.html** file:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title> A list of users </title>
    <link
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
rel="stylesheet" />
    <script src="https://code.jquery.com/jquery-2.2.4.min.js"></script>
</head>
<body>
    <h2> A list of users </h2>
    <form name="userForm">
        <input type="hidden" name="id" value="0" />
```

```
    <div class="form-group">
        <label for="name">Name:</label>
        <input class="form-control" name="name" />
    </div>
    <div class="form-group">
        <label for="age"> Age:</label>
        <input class="form-control" name="age" />
    </div>
    <div class="panel-body">
        <button type="submit" class="btn btn-sm btn-primary"> Save </button>
        <a id="reset" class="btn btn-sm btn-primary"> Reset </a>
    </div>
</form>
<table class="table table-condensed table-striped table-bordered">
   <thead><tr><th>Id</th><th>Name</th><th>Age</th><th></th></tr></thead>
    <tbody>
    </tbody>
</table>
  <script>
    // Getting all users
    function GetUsers() {
        $.ajax({
            url: "/api/users",
            type: "GET",
            contentType: "application/json",
            success: function (users) {
                var rows = "";
                $.each(users, function (index, user) {
```

```
            // add the received elements to the table
            rows += row(user);
        })
        $("table tbody").append(rows);
    }
  });
}
// Getting one user
function GetUser(id) {
  $.ajax({
    url: "/api/users/"+id,
    type: "GET",
    contentType: "application/json",
    success: function (user) {
      var form = document.forms["userForm"];
      form.elements["id"].value = user.id;
      form.elements["name"].value = user.name;
      form.elements["age"].value = user.age;
    }
  });
}
// Add User
function CreateUser(userName, userAge) {
  $.ajax({
    url: "api/users",
    contentType: "application/json",
    method: "POST",
    data: JSON.stringify({
```

```
            name: userName,
            age: userAge
        }),
        success: function (user) {
            reset();
            $("table tbody").append(row(user));
        }
    })
}
// User change
function EditUser(userId, userName, userAge) {
    $.ajax({
        url: "api/users",
        contentType: "application/json",
        method: "PUT",
        data: JSON.stringify({
            id: userId,
            name: userName,
            age: userAge
        }),
        success: function (user) {
            reset();
            $("tr[data-rowid='" + user.id + "']").replaceWith(row(user));
        }
    })
}
  // form reset
function reset() {
```

```
        var form = document.forms["userForm"];
        form.reset();
        form.elements["id"].value = 0;
    }
     // Delete user
    function DeleteUser(id) {
        $.ajax({
            url: "api/users/"+id,
            contentType: "application/json",
            method: "DELETE",
            success: function (user) {
                console.log(user);
                $("tr[data-rowid='" + user.id + "']").remove();
            }
        })
    }
    // creating a row for a table
    var row = function (user) {
        return "<tr data-rowid='" + user.id + "'><td>" + user.id + "</td>" +
               "<td>" + user.name + "</td> <td>" + user.age + "</td>" +
               "<td><a class='editLink' data-id='" + user.id + "'> Edit </a> | " +
                "<a class='removeLink' data-id='" + user.id + "'> Delete </a>
</td>
</tr>";
    }
    // reset form values
    $("#reset").click(function (e) {
         e.preventDefault();
```

```
            reset();
        })
         // form submission
        $("form").submit(function (e) {
            e.preventDefault();
            var id = this.elements["id"].value;
            var name = this.elements["name"].value;
            var age = this.elements["age"].value;
            if (id == 0)
                CreateUser(name, age);
            else
                EditUser(id, name, age);
        });
         // click on the link Edit
        $("body").on("click", ".editLink", function () {
            var id = $(this).data("id");
            GetUser(id);
        })
        // click on the Delete link
        $("body").on("click", ".removeLink", function () {
            var id = $(this).data("id");
            DeleteUser(id);
        })
         // user upload
        GetUsers();
    </script>
</body>
</html>
```

The main logic here is JavaScript. To simplify interaction with the server, the jqQery library is used here. When loading the page in the browser, we get all the objects from the database using the GetUsers function:

```
function GetUsers() {
    $.ajax({
        url: "/api/users",
        type: "GET",
        contentType: "application/json",
        success: function (users) {
            var rows = "";
            $.each(users, function (index, user) {
                // add the received elements to the table
                rows += row(user);
            })
            $("table tbody").append(rows);
        }
    });
}
```

To add rows to a table, use the **row()** function, which returns a row. This line will define the links for changing and deleting the user.

The link for changing the user using the **GetUser()** function receives from the dedicated user server:

```
function GetUser(id) {
    $.ajax({
        url: "/api/users/"+id,
        type: "GET",
        contentType: "application/json",
        success: function (user) {
```

```
            var form = document.forms["userForm"];
            form.elements["id"].value = user.id;
            form.elements["name"].value = user.name;
            form.elements["age"].value = user.age;
        }
    });
}
```

And the highlighted user is added to the form above the table. The same form is used to add an object. Using the hidden field that stores the user id, we can find out what action is performed - adding or editing.

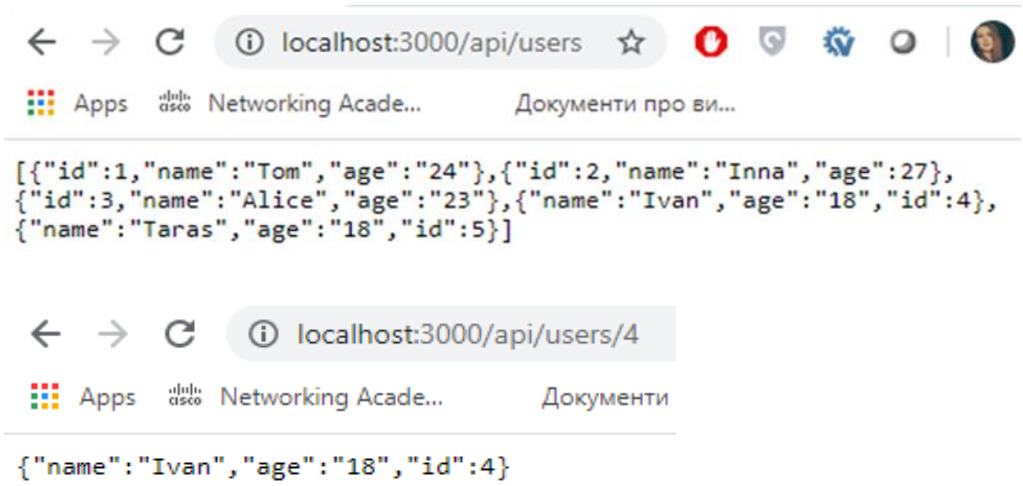
If id is 0, then the CreateUser function is executed, which sends data in the POST request:

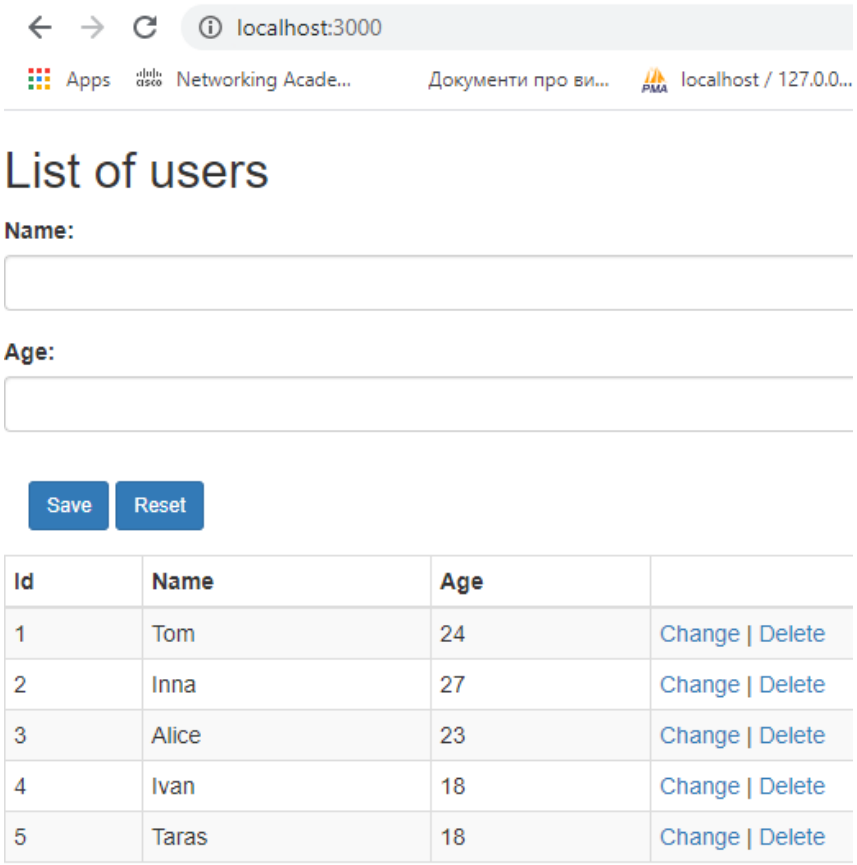
```
function CreateUser(userName, userAge) {
    $.ajax({
        url: "api/users",
        contentType: "application/json",
        method: "POST",
        data: JSON.stringify({
            name: userName,
            age: userAge
        }),
        success: function (user) {
            reset();
            $("table tbody").append(row(user));
        }
    })
}
```

If previously the user was uploaded to the form and his id was stored in a hidden field, the EditUser function is executed, which sends a PUT request:

```
function EditUser(userId, userName, userAge) {
    $.ajax({
        url: "api/users",
        contentType: "application/json",
        method: "PUT",
        data: JSON.stringify({
            id: userId,
            name: userName,
            age: userAge
        }),
        success: function (user) {
            reset();
            $("tr[data-rowid='" + user.id + "']").replaceWith(row(user));
        }
    })
}
```

Run the application, go to the browser at http://localhost:3000 and we can manage the users stored in the json file:





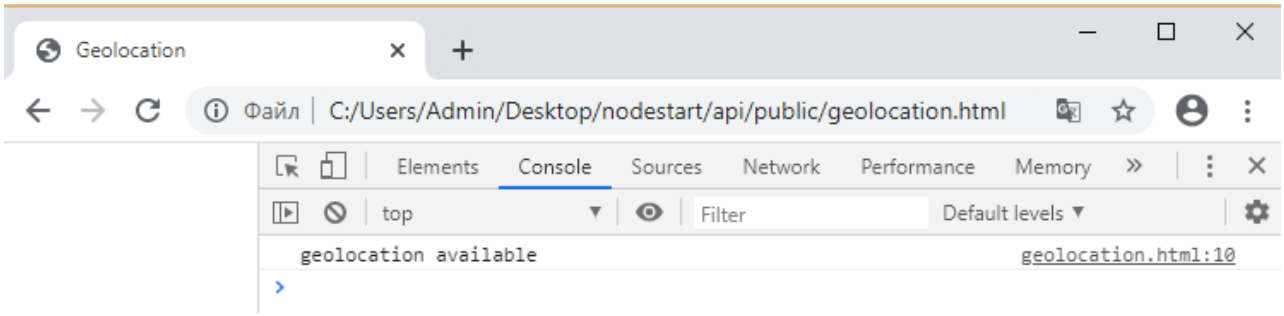
Geolocation API

This feature is available only in secure contexts (HTTPS), in some or all supporting browsers

The Geolocation API allows the user to provide their location to web applications if they so desire. For privacy reasons, the user is asked for permission to report location information. WebExtensions that wish to use the Geolocation object must add the "geolocation" permission to their manifest. The user's operating system will prompt the user to allow location access the first time it is requested [4,5].

In order to check whether geolocation is available on our computer, create a **geolocation.html** file and run it in the browser console [6]:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Geolocation</title>
</head>
<body>
   <script>
       if ('geolocation' in navigator) {
           console.log('geolocation available');
       }
       else {
           console.log('geolocation not available');
       }
    </script>
</body>
</html>
```



To obtain the user's current location, you can call the **getCurrentPosition()** method. This initiates an asynchronous request to detect the user's position, and queries the positioning hardware to get up-to-date information. When the position is determined, the defined callback function is executed. We can optionally provide a second callback function to be executed if an error occurs. A third, optional, parameter is an options object where we can set the maximum age of the position returned, the time to wait for a request, and if we want high accuracy for the position.

By default, **getCurrentPosition()** tries to answer as fast as possible with a low accuracy result. It is useful if you need a quick answer regardless of the accuracy. Devices with a GPS, for example, can take a minute or more to get a GPS fix, so less accurate data (IP location or wifi) may be returned to **getCurrentPosition().**

```
navigator.geolocation.getCurrentPosition((position) => {
  doSomething(position.coords.latitude, position.coords.longitude);
});
```

The above example will cause the **doSomething()** function to execute when the location is obtained.

Leaflet

Leaflet is the leading open-source JavaScript library for mobile-friendly interactive maps. Weighing just about 38 KB of JS, it has all the mapping features most developers ever need. Leaflet is designed with simplicity, performance and usability in mind. It works efficiently across all major desktop and mobile platforms, can be extended with lots of plugins, has a beautiful, easy to use and well-documented API and a simple, readable source code that is a joy to contribute to [7].

Before writing any code for the map, we need to do the following preparation steps on our page [8]:

- Include Leaflet CSS file in the head section of your document:

```
<link rel="stylesheet" href="https://unpkg.com/leaflet@1.6.0/dist/leaflet.css"
   integrity="sha512-xwE/Az9zrjBIphAcBb3F6JVqxf46+CDLwfLMHloNu6KEQCAWi6HcDUbeOfBIptF7tcCzusKFjFw2yuvEpDL9wQ=="
   crossorigin=""/>
```

- Include Leaflet JavaScript file after Leaflet's CSS:

```
<!-- Make sure you put this AFTER Leaflet's CSS -->
<script src="https://unpkg.com/leaflet@1.6.0/dist/leaflet.js"
   integrity="sha512-gZwIG9x3wUXg2hdXF6+rVkLF/0Vi9U8D2Ntg4Ga5I5BZpVkVxlJWbSQtXPSiUTtC0TjtGOmxa1AJPuV0CPthew=="
   crossorigin=""></script>
```

- Put a div element with a certain id where you want your map to be:

```
<div id="mapid"></div>
```

- Make sure the map container has a defined height, for example by setting it in CSS:

```
#mapid {height: 180px; }
```

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <link
      rel="stylesheet"
      href="https://unpkg.com/leaflet@1.4.0/dist/leaflet.css"
      integrity="sha512-puBpdR0798OZvTTbP4A8Ix/l+A4dHDD0DGqYW6RQ+9jxkRFclaxxQb/SJAWZfWAkuyeQUytO7+7N4QKrDh+drA=="
      crossorigin=""
    />
    <script
      src="https://unpkg.com/leaflet@1.4.0/dist/leaflet.js"
      integrity="sha512-QVftwZFqvtRNi0ZyCtsznlKSWOStnDORoefr1enyq5mVL4tmKB3S/EnC3rRJcxCPavG10IcrVGSmPh6Qw5lwrg=="
      crossorigin=""
    ></script>
    <style>
      #mymap {
        height: 180px;
      }
    </style>
    <title>Geolocation</title>
  </head>
```

First we'll initialize the map and set its view to our chosen geographical coordinates and a zoom level [8]:

```
var mymap = L.map('mapid').setView([51.505, -0.09], 13);
```

By default (as we didn't pass any options when creating the map instance), all mouse and touch interactions on the map are enabled, and it has zoom and attribution controls.

SetView call also returns the map object, most Leaflet methods act like this when they don't return an explicit value, which allows convenient jQuery-like method chaining.

Next we'll add a tile layer to add to our map, in this case it's a Mapbox Streets tile layer. Creating a tile layer usually involves setting the URL template for the tile images, the attribution text and the maximum zoom level of the layer. In this example we'll use the mapbox/streets-v11 tiles from Mapbox's Static Tiles API (in order to use tiles from Mapbox, we must also request an access token). Because this API returns

512x512 tiles by default (instead of 256x256), we will also have to explicitly specify this and offset our zoom by a value of -1.

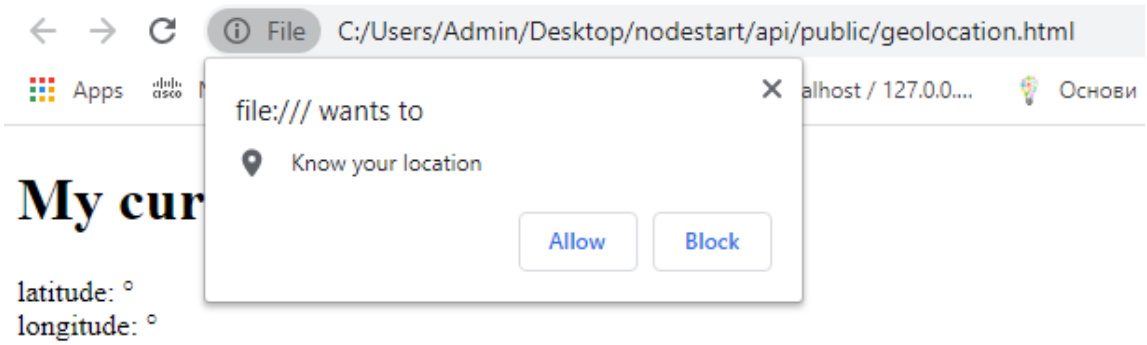
```
  L.tileLayer('https://api.mapbox.com/styles/v1/{id}/tiles/{z}/{x}/{y}?access_token={accessToken}', {
      attribution: 'Map data &copy; <a href="https://www.openstreetmap.org/">OpenStreetMap</a> contributors, <a href="https://creativecommons.org/licenses/by-sa/2.0/">CC-BY-SA</a>, Imagery © <a href="https://www.mapbox.com/">Mapbox</a>',
      maxZoom: 18,
      id: 'mapbox/streets-v11',
      tileSize: 512,
      zoomOffset: -1,
      accessToken: 'your.mapbox.access.token'
  }).addTo(mymap);
```

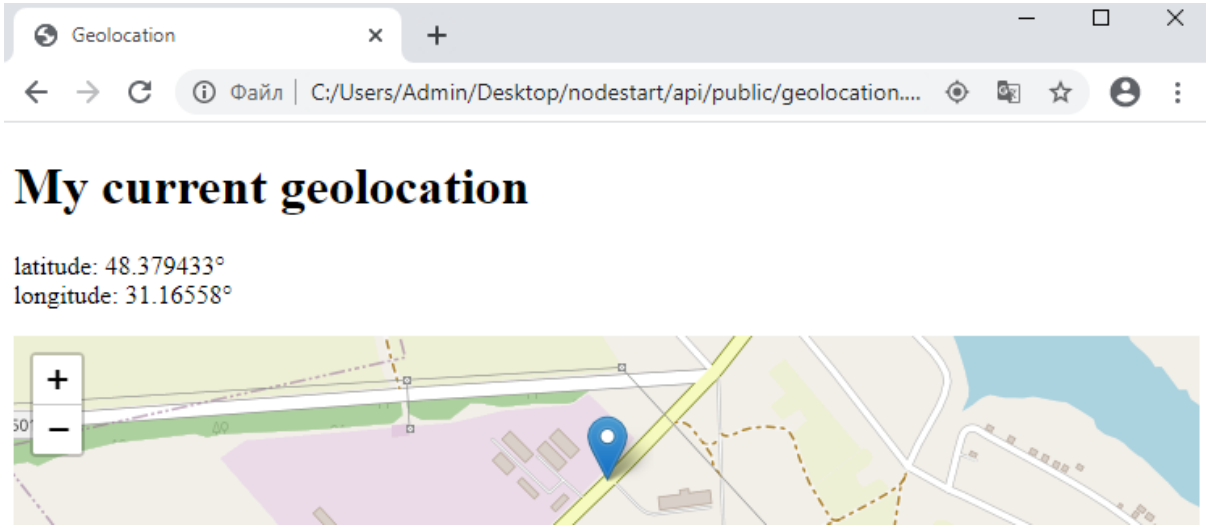
Adding a marker for a map [8]:

```
var marker = L.marker([51.5, -0.09]).addTo(mymap);
```

Let's change and run our code for this task, displaying the user's current location:

```
<body>
  <h1>My current geolocation</h1>
  <p>
    latitude: <span id="latitude"></span>&deg;<br />
    longitude: <span id="longitude"></span>&deg;
  </p>
  <div id="mymap"></div>
  <script>
    if ('geolocation' in navigator) {
      console.log('geolocation available');
      navigator.geolocation.getCurrentPosition(position => {
        lat = position.coords.latitude;
        lon = position.coords.longitude;
        console.log(lat, lon);
        document.getElementById('latitude').textContent = lat;
        document.getElementById('longitude').textContent = lon;

        const mymap = L.map('mymap').setView([lat, lon], 15);
        const attribution =
          '&copy; <a href="https://www.openstreetmap.org/copyright">OpenStreetMap</a> contributors';
        const tileUrl = 'https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png';
        const tiles = L.tileLayer(tileUrl, { attribution });
        tiles.addTo(mymap);
        const marker = L.marker([lat, lon]).addTo(mymap);
      });
    } else {
      console.log('geolocation not available');
    }
  </script>
</body>
</html>
```





Tasks for laboratory work 5

Task 1

Using Express and body-parser, create REST-style Node.js API that will store data in a json file. Create an API for the server, add a json file with 5 entries to the project folder according to the option in the table below.

To search for a mutable object, read data from a file, the user is searched by id, it can change properties and save updated data to a file. To process requests, use the GET, POST, PUT, DELETE methods for each type of request. To store static files, use the **public** folder. In this folder, create an **index.html** file that will act as a client and perform changes or deletions of data, as shown in the example above. If the desired user was not found by **id** in the array, return the status code 404.

Table 5.1. Individual task (by number in the group list)

№	Example JSON Record (create 5 different entries)	The objects
1	"id":1, "name":"", "price":	appliances
2	"id":1, "name":"", "price":	tourism in your city (types of services)
3	"id":1, "name":"", "price":	network equipment
4	"id":1, "name":"", "price":	perfumes
5	"id":1, "name":"", "price":	tea
6	"id":1, "name":"", "price":	medicine
7	"id":1, "name":"", "price":	clothes
8	"id":1, "name":"", "price":	sweets
9	"id":1, "name":"", "price":	travel agency (types of services)
10	"id":1, "name":"", "price":	cars
11	"id":1, "name":"", "price":	phones
12	"id":1, "name":"", "price":	bicycles
13	"id":1, "name":"", "price":	coffee makers

14	"id":1, "name":"", "price":	stereo systems
15	"id":1, "name":"", "price":	ovens
16	"id":1, "name":"", "price":	kitchen combiners
17	"id":1, "name":"", "price":	electric kettles
18	"id":1, "name":"", "price":	refrigerators
19	"id":1, "name":"", "price":	computer equipment
20	"id":1, "name":"", "price":	sports equipment
21	"id":1, "name":"", "price":	motorcycles
22	"id":1, "name":"", "price":	server hardware
23	"id":1, "name":"", "price":	books
24	"id":1, "name":"", "price":	laptops
25	"id":1, "name":"", "price":	kitchens
26	"id":1, "name":"", "price":	Men's clothes
27	"id":1, "name":"", "price":	bags
28	"id":1, "name":"", "price":	shoes
29	"id":1, "name":"", "price":	air conditioners

30	"id":1, "name":"", "price":	vacuum cleaners

Task 2

Create a **geolocation.html** file in the **public** folder, which will check the ability to display the user's geolocation and, with the consent of the user, will display a map with the coordinates of the user's current location.

Report requirements for laboratory work

The report should include:

1. Title page.

2. Tasks for laboratory work.

3. Description of development steps. This section consists of a sequential description of the steps performed according to the instructions for laboratory work.

4. Conclusions.

Questions for self-assessment

1. What is API? What is the REST architecture?

2. Why are the methods app.get () / app.post () / app.delete () / app.put () used?

4. How to get data from a file using the fs.readFileSync () method?

5. What is the JSON.parse () method used for?

6. What opportunities does Geolocation API have?

7. What is Leaflet for JavaScript? How to use Leaflet to add the user's current location on the map?

References

1. Express https://metanit.com/web/nodejs/4.1.php
2. Body-parser https://www.npmjs.com/package/body-parser
3. Creating an API https://metanit.com/web/nodejs/4.11.php
4. Geolocation API https://developer.mozilla.org/en-US/docs/Web/API/Geolocation_API
5. Navigator https://developer.mozilla.org/en-US/docs/Web/API/Navigator
6. Using the Geolocation API https://developer.mozilla.org/en-US/docs/Web/API/Geolocation_API/Using_the_Geolocation_API
7. Leaflet https://leafletjs.com/
8. Leaflet Quick Start Guide https://leafletjs.com/examples/quick-start/

LABORATORY WORK 6.
NODE.JS AND MONGODB

Purpose: to get practical skills using MongoDB with Node.js for connection the database to the server, get practical skills getting database objects on the server, getting the collection object in the database and interaction with the collection.

Theory methodological instructions

MongoDB is an open source document management database system that does not require a description of the table schema. Classified as NoSQL, uses JSON-like documents and database schema. It is written in C ++. It is used in web development, in particular, as part of the JavaScript-oriented MEAN stack.

MongoDB implements a new approach to building databases where there are no tables, schemas, SQL queries, foreign keys, and many other things that are inherent in object-relational databases.

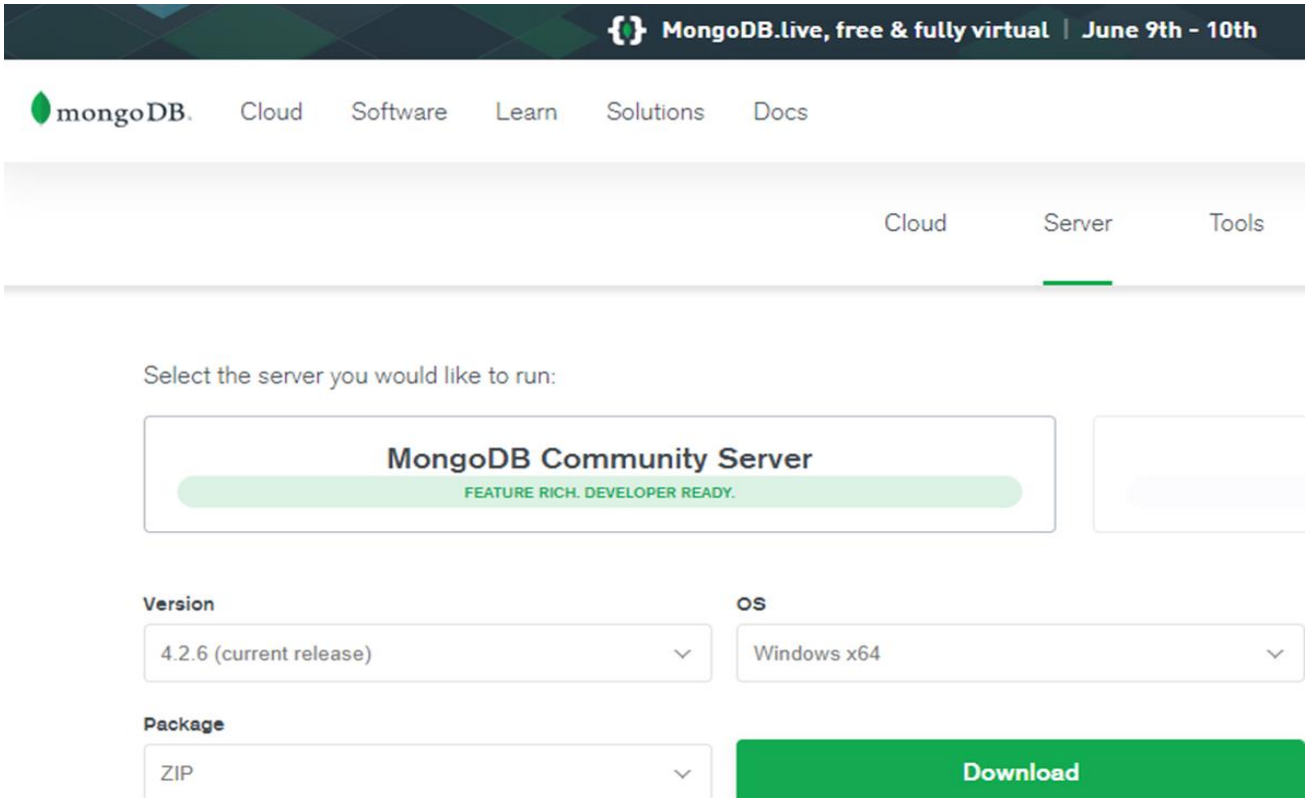
MongoDB implements a new approach to building databases where there are no tables, schemas, SQL queries, foreign keys, and many other things that are inherent in object-relational databases.

For storage, MongoDB uses a format called **BSON**, or short for binary JSON. BSON allows to work with data faster: faster search and processing.

Although it should be noted that BSON, in contrast to storing data in JSON format, has a small drawback: in general, data in JSON format takes up less space than in BSON format, on the other hand, this drawback is more than paying for speed.

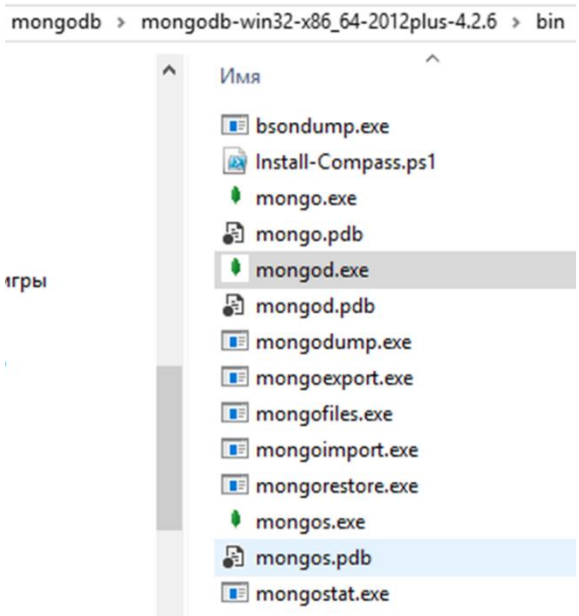
Installing MongoDB

To download the necessary functionality, we will select the necessary operating system and the appropriate type of package:



MongoDB Package Contents

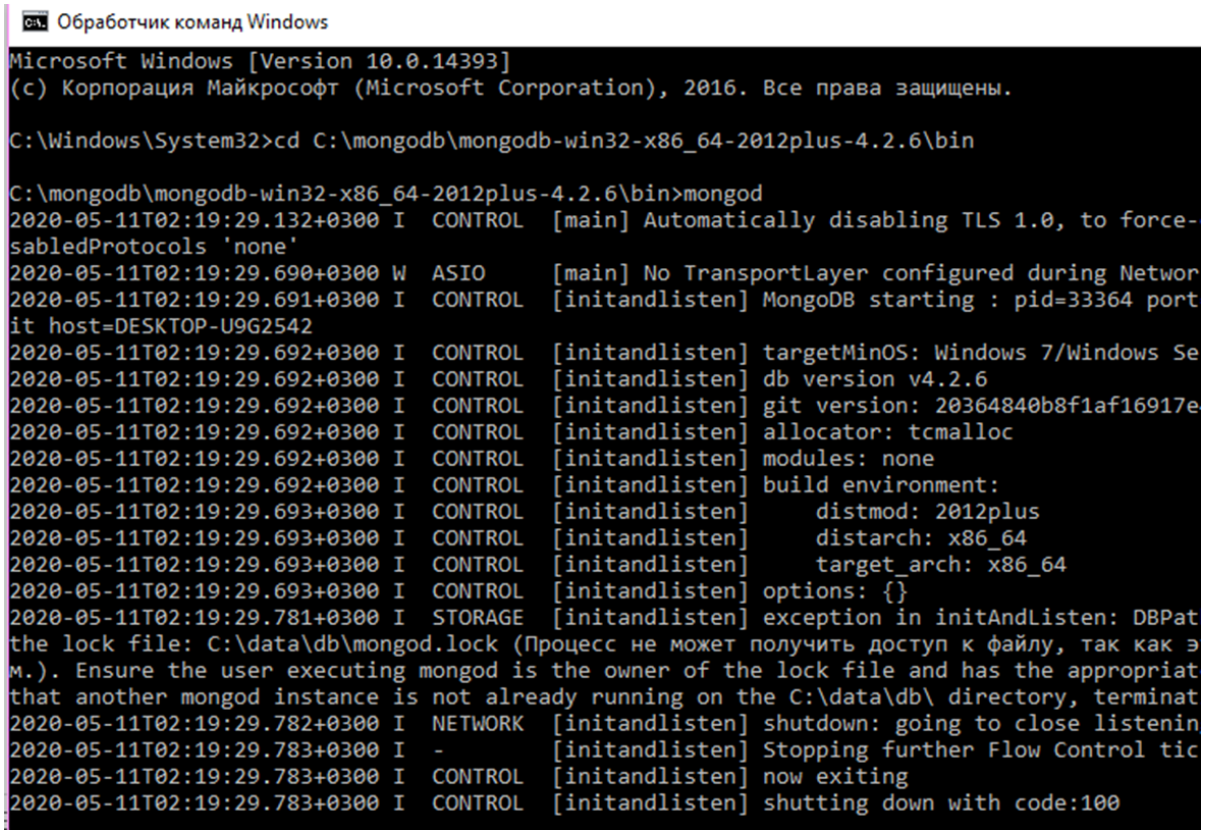
If after installation we open the folder **C:\mongodb\bin**, then we can find applications that play a certain role there:



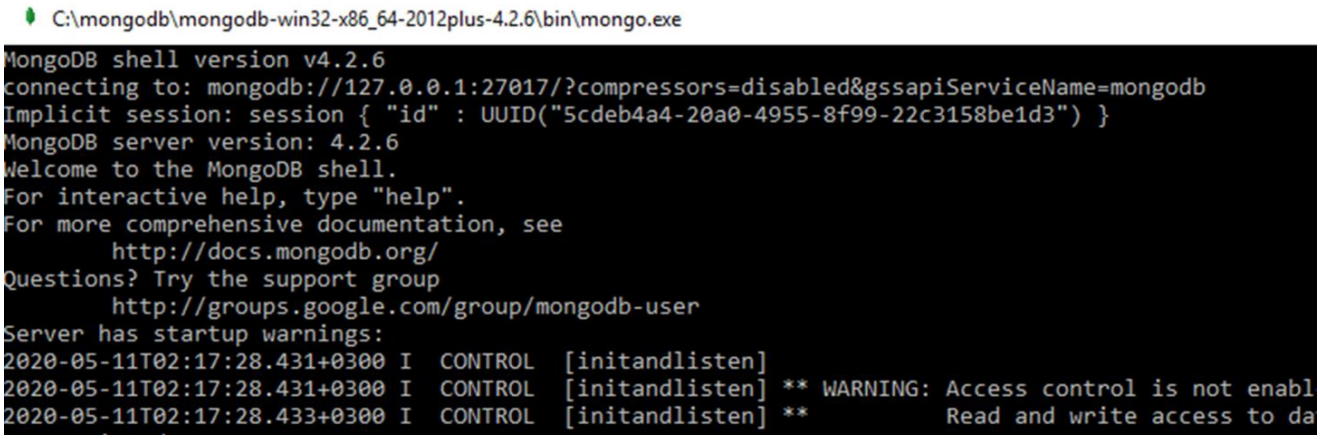
- **bsondump**: reads the contents of BSON files and converts them into a readable format, for example, in JSON;
- **mongo**: introduces a console interface for interacting with databases, a kind of console client;
- **mongod**: MongoDB database server, it processes requests, manages the data format and performs various operations in the background for database management;

- **mongodump**: database backup utility;
- **mongoexport**: utility for exporting data in JSON, TSV or CSV formats;
- **mongofiles**: a utility that allows to manage files on a GridFS system;
- **mongoimport**: utility importing JSON, TSV or CSV data into MongoDB database;
- **mongorestore**: allows to write data from a dump created by mongodump to a new or existing database;
- **mongos**: MongoDB routing service that helps process requests and locate data in a MongoDB cluster;
- **mongorestat**: represents database operation counters;
- **mongotop**: provides a way to calculate the time spent on read / write operations in the database.

After creating the directory for storing the database, you can start the MongoDB server. The server represents the mongod application, which is located in the bin folder. To do this, run the command line (on Windows) or the console on Linux and enter the appropriate commands there. For Windows, it will look like this:

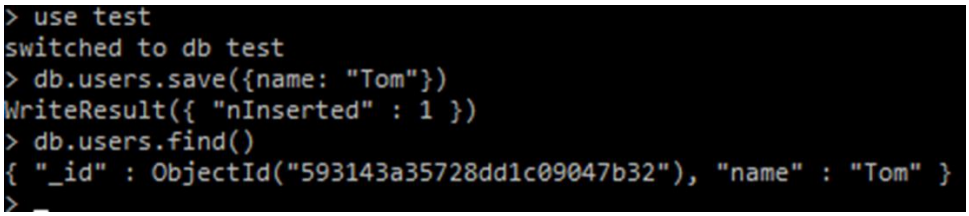


The command line will show us a series of service information, for example, that the server starts on localhost on port **27017**. And after the server starts successfully, we can perform operations with the mongo shell. This shell represents the mongo.exe file, which is located in the above installation folder. Run this file:



We will enter the following commands in mongo sequentially and after each command press Enter:

```
use test
db.users.save( { name: "Tom" } )
db.users.find()
```



The use test command sets the **test** database to be used. Even if there is no such database, then it is created automatically. And then db will represent the current database - that is, the **test** database. After db comes **users** - this is a collection into which we then add a new object. If in SQL we need to create tables in advance, then MongoDB creates collections ourselves if they are not available.

Using the **db.users.save ()** method, the **{name: "Tom"}** object is added to the users collection of the test database. The description of the added object is defined in JSON format.

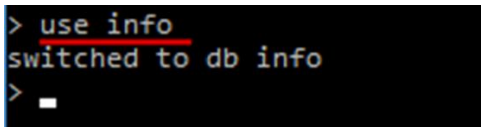
In this case, the object has one key named "name", which is associated with the value "Tom". That is, we add a user named Tom.

If the object was successfully added, then the console will output the result in the form of an expression **WriteResult ({"nInserted": 1}).**

The **db.users.find ()** command displays all objects from the test database.

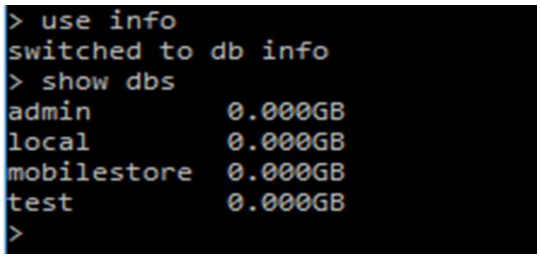
Database installation and administration

Starting to work with MongoDB, the first thing is to set the database we need as the current one, so that we can use it later. To do this, use the use command, followed by the name of the database. It doesn't matter if such a database exists or not. If it is not, then MongoDB will automatically create it when data is added to it. For example, run **mongo.exe** and enter the following command there:



Now the info database will be installed as the current one.

If you are suddenly not sure if a database with that name already exists, then using the show **dbs** command you can display the names of all available databases on the console:



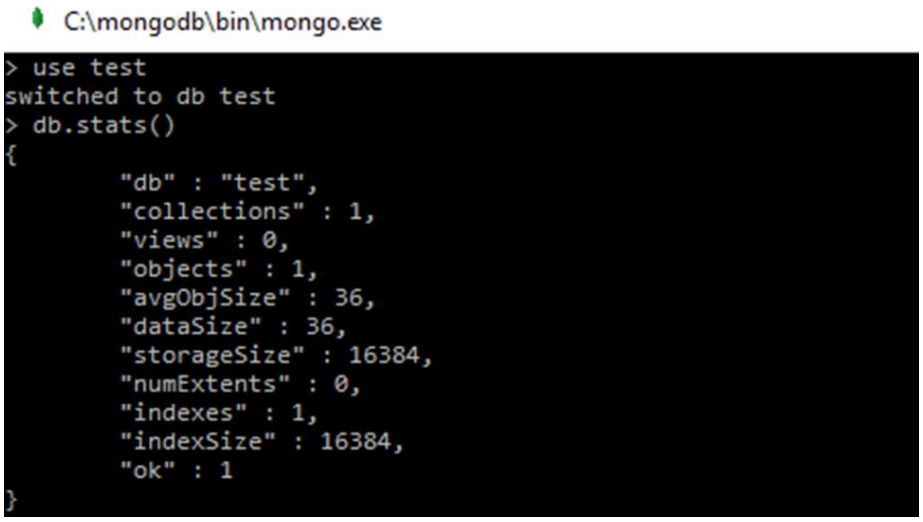
You can specify any name for the database, but there are some restrictions. For example, the name should not contain the characters **/, \,., ", *, <,>,:, |,?, $.** In addition, database names are limited to 64 bytes. There are also reserved names that cannot be used: **local, admin, config**. Moreover, as you can see, the info database is not in this list, since I have not added data to it yet. If we want to find out which database is currently in use, then we can use the db command:

> db

info

Getting statistics

Using the **db.stats ()** command, we can get statistics on the current database. For example, we have the **test** database installed as the current one:



To work with the MongoDB platform, you first need to install the MongoDB server itself. More details on how to do this are described here [1]. In addition to the Mongo server itself, we need a driver [2] to interact with Node.js.

When connecting and interacting with a database in MongoDB, the following steps can be distinguished:

• Connection to the server;

• Getting the database object on the server;

• Getting the collection object in the database;

• Interaction with the collection (add, delete, receive, modify data).

So, create a new project. To do this, define a new directory, which will be called **mongoapp**. Next, define a new **package.json** file in this directory:

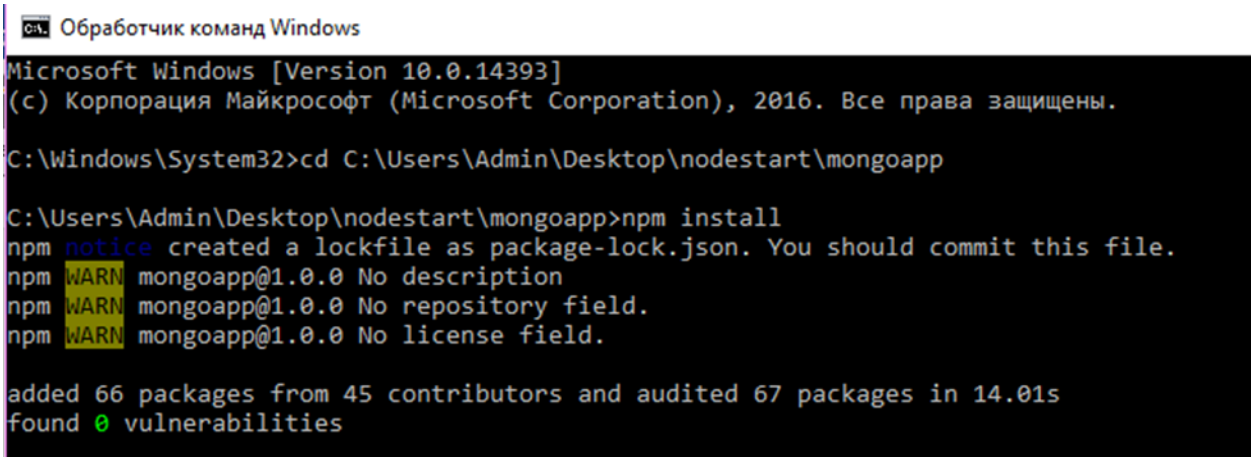
```
{
  "name": "mongoapp",
  "version": "1.0.0",
  "dependencies": {
     "express": "^ 4.16.0",
     "body-parser": "^ 1.18.0",
```

```
    "mongodb": "^ 3.1.0"
  }
}
```

In this case, the last dependency "mongodb" represents the driver. All the necessary help information specifically for this driver can be found at https://mongodb.github.io/node-mongodb-native/

Next, we will go to this directory at the command line / terminal and to add all the necessary packages, execute the command:

npm install



Database connection

The key class for working with MongoDB is the MongoClient class, and all interactions with the data warehouse will go through it. Accordingly, we must first get MongoClient:

```
const MongoClient = require ("mongodb"). MongoClient;
```

To connect to the mongodb server, the **connect ()** method is used:

```
const MongoClient = require ("mongodb"). MongoClient;
// create a MongoClient object and pass it a connection string
const mongoClient = new MongoClient ("mongodb://localhost:27017/", {useNewUrlParser: true});
mongoClient.connect (function (err, client) {
    if (err) {
        return console.log (err);
```

```
    }
    // interaction with the database
    client.close ();
});
```

First, a MongoClient object is created. To do this, two parameters are passed to its constructor. The first parameter is the server address. The address protocol is set to "mongodb://". On the local machine, the address will be localhost, followed by the port number. The default port number is **27017**.

The second parameter is an optional configuration object. MongoDb is constantly evolving. In this case, the configuration object that has the useNewUrlParser property is used: **true** – it indicates to the mongodb infrastructure that it is necessary to use the new server address parcel.

Next, using the connect method, a connection is made to the server. As a parameter, the method takes a callback function, which is triggered when a connection is established. This function takes two parameters: err (the error that occurred while establishing the connection) and client (link to the client connected to the server).

If errors occurred while connecting, then we can use the err value to get the error.

If there is no error, then we can interact with the server through the client object.

At the end of the work with the database, we need to close the connection using the **client.close ()** method.

Database, collections and documents

Having received the object of the connected client, we can access the database on the server. To do this, use the method

client.db ("DB_name");

The name of the database to which we want to connect is passed as a parameter to the method.

The database in MongoDB has no tables. Instead, all data falls into collections. And within the framework of node.js, to interact with the database (add, delete, read data), we need to get a collection object. To do this, the **db.collection ("collection_name")** method is used, to which the name of the collection is passed.

Unlike tables in relational systems, where all data is stored as rows, in collections in MongoDB, data is stored as documents. For example, add one document to the database. To do this, define the following **app.js** file in the project directory:

```
const MongoClient = require("mongodb").MongoClient;
  const url = "mongodb://localhost:27017/";
const mongoClient = new MongoClient(url, { useNewUrlParser: true });
 mongoClient.connect(function(err, client){
        const db = client.db("usersdb");
    const collection = db.collection("users");
    let user = {name: "Tom", age: 23};
    collection.insertOne(user, function(err, result){
            if(err){
        return console.log(err);
      }
      console.log(result.ops);
      client.close();
    });
});
```

The database used here is "usersdb". It doesn’t matter that by default there is no such database on the MongoDB server. At the first access to it, the server will automatically create it.

After connecting, we turn to the "users" collection:

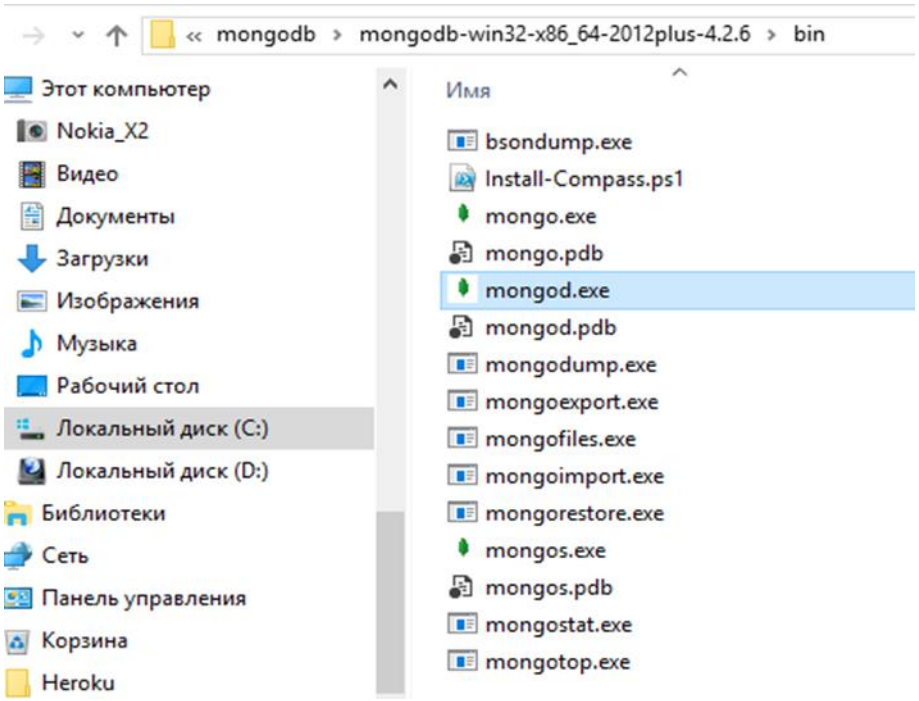
```
const collection = db.collection ("users");
```

Again, it doesn’t matter that such a collection is not in the usersdb database by default, it will also be created upon first access.

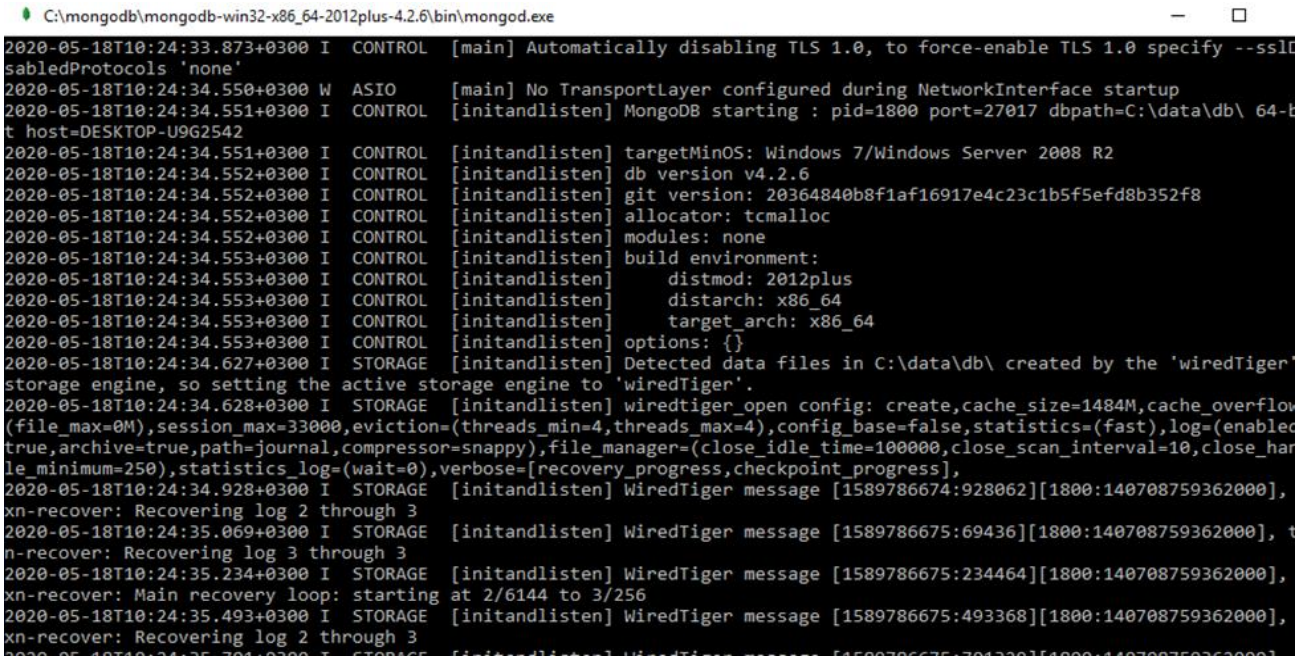
Having received the collection, we can use its methods. In this case, to add a single document – the user object, the **insertOne ()** method is used. This method has two parameters – the added object itself and the callback function, which is executed after the addition. Two parameters are used in this function: err (an error that may occur during the operation) and result (the result of the operation is an added object).

In the callback function, the added object is inspected using the **result.ops** property. Moreover, this is not just a user object, but an object that is retrieved back from the database and which contains the identifier set when adding it.

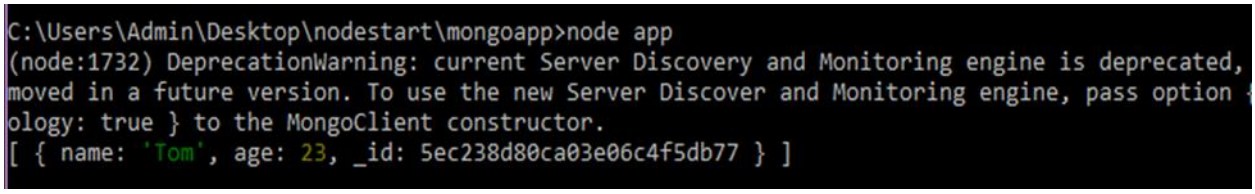
Now, on the hard drive, go to the directory where **mongodb** is installed, and in this directory, go to the **bin** folder:



Run the mongodb server, which is located in this directory and which is the mongod console program.



Then run our **app.js** file:



As we see, in addition to the initial properties, the document here also has the additional property **_id** - this is the unique identifier of the document that is assigned by the server when it is added.

Adding and Retrieving Data in MongoDB

For adding we can use various methods. If you need to add a single object, the **insertOne ()** method is used. When adding a set of objects, you can use the **insertMany ()** method.

We use the **insertMany ()** method. Add a set of objects and for this change the application file **app.js**:

```
const MongoClient = require ("mongodb"). MongoClient;
  const url = "mongodb://localhost:27017/";
const mongoClient = new MongoClient (url, {useNewUrlParser: true});
 let users = [{name: "Bob", age: 34}, {name: "Alice", age: 21}, {name: "Tom", age: 45}];
  mongoClient.connect (function (err, client) {
  const db = client.db ("usersdb");
  const collection = db.collection ("users");
  collection.insertMany (users, function (err, results) {
   console.log (results);
   client.close ();      });  });
```

Like **insertOne**, the **insertMany ()** method accepts the data to be added as the first parameter – an array of objects, and as the second parameter – a callback function that is executed when the data is added. Upon successful addition, the second parameter of the function – results will contain the added data. Launch the application:

```
C:\Users\Admin\Desktop\nodestart\mongoapp>node app
(node:8956) DeprecationWarning: current Server Discovery and Monitoring engine is
moved in a future version. To use the new Server Discover and Monitoring engine,
ology: true } to the MongoClient constructor.
{
  result: { ok: 1, n: 3 },
  ops: [
    { name: 'Bob', age: 34, _id: 5ec23bdf2822c322fcc03f6f },
    { name: 'Alice', age: 21, _id: 5ec23bdf2822c322fcc03f70 },
    { name: 'Tom', age: 45, _id: 5ec23bdf2822c322fcc03f71 }
  ],
  insertedCount: 3,
  insertedIds: {
    '0': 5ec23bdf2822c322fcc03f6f,
    '1': 5ec23bdf2822c322fcc03f70,
    '2': 5ec23bdf2822c322fcc03f71
  }
}
```

In addition to the data itself, the **results** parameter will contain some additional information about the performed add operation.

Data retrieval

The **find ()** method is used to get data from the collection:

```
const MongoClient = require ("mongodb"). MongoClient;
  const url = "mongodb: // localhost: 27017 /";
const mongoClient = new MongoClient (url, {useNewUrlParser: true});
 mongoClient.connect (function (err, client) {
        const db = client.db ("usersdb");
    const collection = db.collection ("users");
     if (err) return console.log (err);
        collection.find (). toArray (function (err, results) {
```

```
            console.log (results);
        client.close ();
    });
});
```

The **find** method returns a special object Cursor, and the **toArray ()** method is called to get all the data from this object. A callback function with standard parameters is passed to this method: **err** (information about the error, if any) and **result** (the actual result of the selection).

And if we run the application, we will see all the previously added data:

```
C:\Users\Admin\Desktop\nodestart\mongoapp>node app
(node:9896) DeprecationWarning: current Server Discovery and Monitoring engine i
moved in a future version. To use the new Server Discover and Monitoring engine,
ology: true } to the MongoClient constructor.
[
  { _id: 5ec238d80ca03e06c4f5db77, name: 'Tom', age: 23 },
  { _id: 5ec23bdf2822c322fcc03f6f, name: 'Bob', age: 34 },
  { _id: 5ec23bdf2822c322fcc03f70, name: 'Alice', age: 21 },
  { _id: 5ec23bdf2822c322fcc03f71, name: 'Tom', age: 45 },
  { _id: 5ec23de3c6e857258435aa4e, name: 'Bob', age: 34 },
  { _id: 5ec23de3c6e857258435aa4f, name: 'Alice', age: 21 },
  { _id: 5ec23de3c6e857258435aa50, name: 'Tom', age: 45 }
]
```

Reading data in Node.js from a MongoDB database

Using the **find ()** method, we can additionally filter the extracted documents. For example, we need to find all users whose name is Tom:

```
const MongoClient = require("mongodb").MongoClient;
   const url = "mongodb://localhost:27017/";
const mongoClient = new MongoClient(url, { useNewUrlParser: true });
 mongoClient.connect(function(err, client){
        const db = client.db("usersdb");
    const collection = db.collection("users");
     if(err) return console.log(err);
```

```
        collection.find({name: "Tom"}).toArray(function(err, results){
                console.log(results);
        client.close();
    });
});
```

An object is passed to find as a parameter, which sets filtering parameters. In particular, that the name property must be equal to "Tom".

```
C:\Users\Admin\Desktop\nodestart\mongoapp>node app
(node:5036) DeprecationWarning: current Server Discovery and Monitorin
moved in a future version. To use the new Server Discover and Monitori
ology: true } to the MongoClient constructor.
[
  { _id: 5ec238d80ca03e06c4f5db77, name: 'Tom', age: 23 },
  { _id: 5ec23bdf2822c322fcc03f71, name: 'Tom', age: 45 },
  { _id: 5ec23de3c6e857258435aa50, name: 'Tom', age: 45 }
]
```

Filtering in MongoDB and Node.js

We can set additional filtering criteria, for example, add age filtering:

```
collection.find({name: "Tom", age: 23}).toArray(function(err, results){
            console.log(results);
    client.close();
});
```

The **findOne ()** method works similarly, only it allows to get one document:

```
const MongoClient = require("mongodb").MongoClient;
   const url = "mongodb://localhost:27017/";
const mongoClient = new MongoClient(url, { useNewUrlParser: true });
 mongoClient.connect(function(err, client){
         if(err) return console.log(err);
        const db = client.db("usersdb");
    db.collection("users").findOne(function(err, doc){
                console.log(doc);
```

```
        client.close();
    });
});
```

And also in the **findOne ()** method, we can apply filtering:

```
db.collection ("users"). findOne ({name: "Bob"}, function (err, doc) {
        console.log (doc);
    client.close ();
});
```

Deleting Documents in MongoDB

There are several ways to delete documents in MongoDB. The following collection methods should be noted here:

• **deleteMany ():** deletes all documents that match a specific criterion

• **deleteOne ():** deletes one document that meets a specific criterion

• **findOneAndDelete ():** retrieves and deletes a single document that meets a specific criterion

• **drop ():** deletes the entire collection

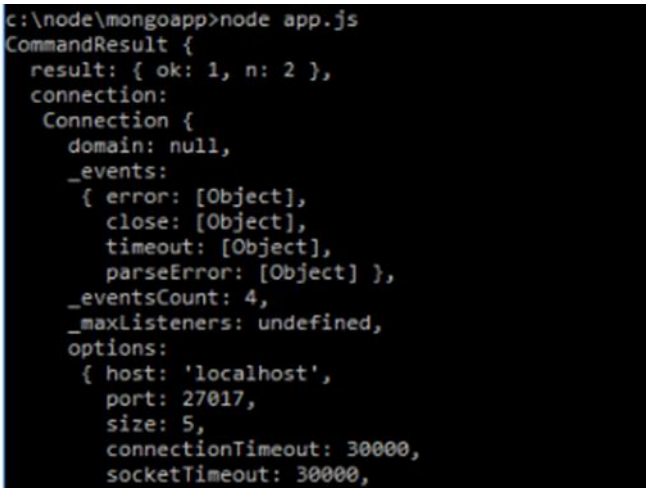
deleteMany

Remove all users with the name "Tom":

```
const MongoClient = require ("mongodb"). MongoClient;
   const url = "mongodb: // localhost: 27017 /";
const mongoClient = new MongoClient (url, {useNewUrlParser: true});
 mongoClient.connect (function (err, client) {
        if (err) return console.log (err);
         const db = client.db ("usersdb");
```

```
db.collection ("users"). deleteMany ({name: "Tom"}, function (err, result) {
        console.log (result);
    client.close ();
});
});
```

The first parameter in the method is a document filter, and the second is a callback function in which we can get the result of deletion. In this case, the result of the removal will be a complex object containing detailed information:



deleteOne

The **deleteOne()** method is similar to the **deleteMany()** method, except that it deletes only one object:

```
const MongoClient = require ("mongodb"). MongoClient;
  const url = "mongodb: // localhost: 27017 /";
const mongoClient = new MongoClient (url, {useNewUrlParser: true});
 mongoClient.connect (function (err, client) {
      if (err) return console.log (err);
       const db = client.db ("usersdb");
```

```
    db.collection ("users"). deleteOne ({name: "Bob"}, function (err, result) {
                console.log (result);
        client.close ();
    });
});
```

findOneAndDelete

The **findOneAndDelete()** method deletes a single document by a specific criterion, but compared to the **deleteOne()** method, it returns a deleted document:

```
const MongoClient = require ("mongodb"). MongoClient;
  const url = "mongodb://localhost: 27017/";
const mongoClient = new MongoClient (url, {useNewUrlParser: true});
 mongoClient.connect (function (err, client) {
       if (err) return console.log (err);
        const db = client.db ("usersdb");
    db.collection ("users"). findOneAndDelete ({age: 21}, function (err, result) {
                console.log (result);
        client.close ();
    });
});
```

drop

The **drop ()** method deletes the entire collection:

```
const MongoClient = require ("mongodb"). MongoClient;
  const url = "mongodb: // localhost: 27017 /";
```

```
const mongoClient = new MongoClient (url, {useNewUrlParser: true});
  mongoClient.connect (function (err, client) {
        if (err) return console.log (err);
         const db = client.db ("usersdb");
     db.collection ("users"). drop (function (err, result) {
                  console.log (result);
        client.close ();
     });
});
```

Updating documents in MongoDB

There are several methods for updating items in MongoDB:

• **updateOne**: updates one document that meets the filtering criteria and returns information about the update operation

• **updateMany**: updates all documents that meet the filtering criteria and returns information about the update operation

• **findOneAndUpdate**: updates one document that matches the filter criteria and returns an updated document

findOneAndUpdate

The **findOneAndUpdate()** method updates a single item. It takes the following parameters:

1. Filter criteria for the document to be updated

2. Update option

3. Additional upgrade options that are null by default

4. The callback function that is performed when updating

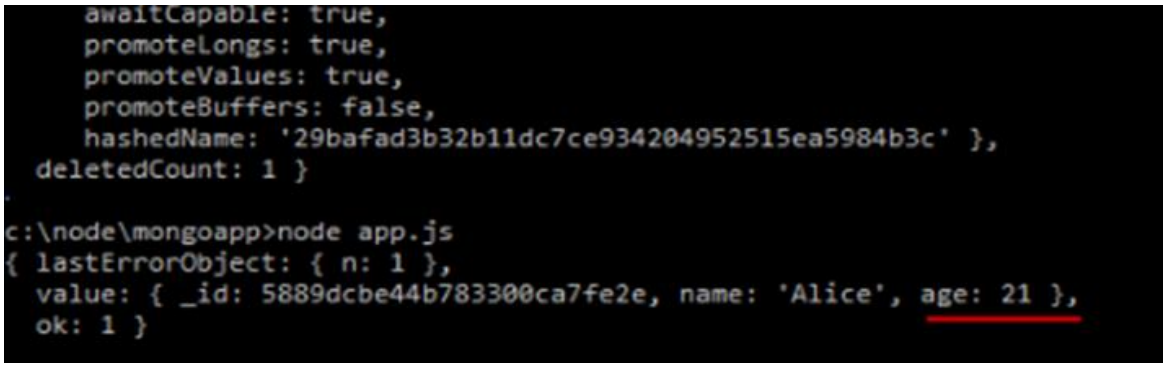
For example, update the first user in the database whose age is 21:

```
const MongoClient = require ("mongodb"). MongoClient;
  const url = "mongodb: // localhost: 27017 /";
const mongoClient = new MongoClient (url, {useNewUrlParser: true});
  let users = [{name: "Bob", age: 34}, {name: "Alice", age: 21}, {name: "Tom", age: 45}];
mongoClient.connect (function (err, client) {
        if (err) return console.log (err);
         const db = client.db ("usersdb");
    const col = db.collection ("users");
    col.insertMany (users, function (err, results) {
                col.findOneAndUpdate (
          {age: 21}, // selection criteria
          {$ set: {age: 25}}, // update option
          function (err, result) {
                  console.log (result);
             client.close ();
          }
        );
    });
});
```

First, 3 users are added to the database here, and after the addition, an update is taking place.

For updating, the object {$ set: {age: 25}} is used. The $ set parameter allows to update values for a single field or group of fields. In this case, the age field changes.

The third parameter is the callback function, which displays the update result. By default, this is the old state of the modified document:



But, let's say, after the update we want to get not the old, but the new state of the changed document. To do this, we can set additional update options.

```
const MongoClient = require ("mongodb"). MongoClient;
  const url = "mongodb: // localhost: 27017 /";
const mongoClient = new MongoClient (url, {useNewUrlParser: true});
 mongoClient.connect (function (err, client) {
       if (err) return console.log (err);
        const db = client.db ("usersdb");
   const col = db.collection ("users");
   col.findOneAndUpdate (
      {name: "Bob"}, // selection criteria
      {$ set: {name: "Sam"}}, // update parameter
      {// add. update options
         returnOriginal: false
      },
```

```
function (err, result) {
        console.log (result);
    client.close ();
}       );    });
```

```
c:\node\mongoapp>node app.js
{ value: { _id: 588a0f8c9151d23ce47bf98d, name: 'Sam', age: 34 },
  lastErrorObject: { updatedExisting: true, n: 1 },
  ok: 1 }
```

updateMany

The **updateMany()** method allows to update all documents from the collection that meet the filtering criteria:

```
const MongoClient = require ("mongodb"). MongoClient;
  const url = "mongodb: // localhost: 27017 /";
const mongoClient = new MongoClient (url, {useNewUrlParser: true});
 mongoClient.connect (function (err, client) {
      if (err) return console.log (err);
       const db = client.db ("usersdb");
   const col = db.collection ("users");
   col.updateMany (
      {name: "Sam"}, // filter criteria
      {$ set: {name: "Bob"}}, // update parameter
      function (err, result) {
          console.log (result);
         client.close ();          }          );   });
```

updateOne

The **updateOne()** method is similar to the updateMany method, except that it updates only one item. Unlike the **findOneAndUpdate()** method, it does not return the modified document:

```
const MongoClient = require ("mongodb"). MongoClient;
  const url = "mongodb: // localhost: 27017 /";
const mongoClient = new MongoClient (url, {useNewUrlParser: true});
 mongoClient.connect (function (err, client) {
        if (err) return console.log (err);
         const db = client.db ("usersdb");
    const col = db.collection ("users");
    col.updateOne (
       {name: "Tom"},
       {$ set: {name: "Tom Junior", age: 33}},
       function (err, result) {
          console.log (result);
          client.close ();
       }
    ); });
```

Tasks for laboratory work 6

1. Connect the database to the server.

2. Get database objects on the server.

3. Get the collection object in the database.

4. Interact with the collection (add, delete, receive, modify data).

Create a project app.js. To do this, define the directory, mongoapp. Define the package.json file in this directory, add the dependencies "express": "^ 4.16.0", "body-parser": "^ 1.18.0", "mongodb": "^ 3.1.0".

Create a usersdb database and a "users" collection. Add to the collection of 10 different documents – user objects. You need to enter the following information about users: name, age, hobby – an array from one, two or three languages (music, sports, drawing) and foreign languages – an array from two or three languages (english, french, german).

Run the following queries in the database:

1. Display the first five users in the database.

2. Get a selection of documents that have two languages simultaneously in the languages array: "english" and "french".

3. Print all documents in which "english" in the languages array is in first place.

4. Print all documents that contain the hobby "music".

5. Display all documents in which the age of users is more than 24 and less than 30.

Report requirements for laboratory work

The report should include:

1. Title page.
2. Tasks for laboratory work.
3. Description of development steps. This section consists of a sequential description of the steps performed according to the instructions for laboratory work.
4. Conclusions.

Questions for self-assessment

1. What is MongoDB?
2. How we can get statistics on the current database?
3. Describe methods for adding a single object and a set of objects to MongoDB.
4. How we can read data in Node.js from a MongoDB database?
5. Describe several ways to delete documents in MongoDB.
6. Describe methods for updating items in MongoDB.

References

1. Installing and getting started with MongoDB on Windows https://metanit.com/nosql/mongodb/1.2.php
2. MongoDB Node.JS Driver https://www.npmjs.com/package/mongodb
3. Node.js and MongoDB https://metanit.com/web/nodejs/6.1.php
4. Adding and receiving data in MongoDB https://metanit.com/web/nodejs/6.2.php
5. Deleting documents in MongoDB https://metanit.com/web/nodejs/6.3.php
6. A selection from the database https://metanit.com/nosql/mongodb/2.4.php
7. Sampling operators https://metanit.com/nosql/mongodb/2.8.php