

NATIONAL TECHNICAL UNIVERSITY OF UKRAINE
“IGOR SIKORSKY KYIV POLYTECHNIC INSTITUTE”

A.V.Petrashenko, O.P.Polishchuk

USER INTERFACE PROGRAMMING

PRACTICUM

Recommended by the Methodical Council of Igor Sikorsky Kyiv Polytechnic Institute as a study guideline for bachelor's degree holders in F7- "Computer Engineering" specialty.

Electronic network educational publication

Kyiv
Igor Sikorsky Kyiv Polytechnic Institute
2025

Reviewer Zabolotnia T.M., PhD, associate professor,
Igor Sikorsky Kyiv Polytechnic Institute

Responsible Radchenko K.O., senior lecturer,
editor Igor Sikorsky Kyiv Polytechnic Institute

*The classification was provided by the Methodical Council of Igor Sikorsky Kyiv Polytechnic
Institute*

*(protocol No. 3 dated 24-10-25)
on the proposal of the Academic Council of the Faculty of Applied Mathematics
(protocol No. 6 dated 01-12-25)*

*Andriy Vasylyovych Petrashenko, PhD, associate professor
Olena Pavlivna Polishchuk, assistant*

USER INTERFACE PROGRAMMING PRACTICUM

User Interface Programming. Practicum [Electronic resource]: education textbook for students of specialty F7 – Computer engineering. / A.V. Petrashenko; O.P.Polishchuk, Igor Sikorsky Kyiv Polytechnic Institute. – Electronic text data (1 file: 3.9 MB). – Kyiv: Igor Sikorsky Kyiv Polytechnic Institute, 2025. – 126 pp.

The practicum offers a comprehensive, hands-on guide to user interface (UI) programming, with a focus on three essential client technologies: web (ReactJS), mobile (Android), and cross-platform (Flutter). It begins with an exploration of the historical evolution of UI technologies and highlights the growing significance of user experience (UX) and visual design in modern applications. Through detailed chapters, readers will gain practical knowledge in building responsive and dynamic user interfaces using React components and hooks, managing state effectively with Redux and Redux Toolkit, designing native Android UIs with XML and Jetpack Compose, and developing cross-platform interfaces in Flutter using widgets, state management with Riverpod, and asynchronous data handling with streams.

The study guide is intended for full-time students of the specialty F7 - "Computer Engineering" of Applied Mathematics Faculty of Igor Sikorsky Kyiv Polytechnic Institute.

© A.V.Petrashenko, O.P.Polishchuk
©Igor Sikorsky Kyiv Polytechnic Institute, 2025

CONTENTS

1 The Evolution of UI Technologies.....	6
1.1 The Origins of User Interaction.....	6
1.2 Transition to Modern UIs.....	8
1.3 The Mobile Revolution	11
1.4 Current Trends in User Interfaces	13
1.5 Importance of UX and Design in UI.....	14
1.6 Overview of Client Technologies.....	16
2 Web UI Development Using React.....	18
2.1 React Essentials.....	18
2.2 ReactJS Installation and Run.....	19
2.3 The Default Folder and File Structure.....	20
2.4 React Components.....	22
2.4.1 Component Composition.....	23
2.4.2 Understanding JSX.....	25
2.4.3 Props in React.....	27
2.4.4 Component State	29
2.4.5 Component Lifecycle with <code>useEffect</code>	32
2.5 Hooks in React.....	35
2.5.1 <code>useState</code> : Managing State.....	35
2.5.2 <code>useEffect</code> : Managing Side Effects.....	36
2.5.3 <code>useContext</code> : Managing Context.....	36
2.5.4 <code>useReducer</code> : Managing Complex State Logic	37
2.5.5 <code>useCallback</code> : Memoizing Functions.....	38
2.5.6 <code>useMemo</code> : Memoizing Values.....	39
2.5.7 Custom Hooks: Reusable Logic.....	40
2.6 State Management.....	41
2.6.1 Types of State Management Libraries.....	42
2.6.1.1 Centralized State Management.....	42
2.6.1.2 Atomized State Management.....	43

2.6.2 Types of State.....	44
2.6.3 State Management Solutions.....	45
2.6.3.1 Local State (<i>useState</i>).....	45
2.6.3.2 Local Reducer State (<i>useReducer</i>).....	46
2.6.3.3 Context API.....	47
2.6.3.4 Redux.....	51
2.6.3.5 Redux Toolkit.....	54
2.6.3.6 REST API with Redux Toolkit.....	58
2.7 React Router.....	62
3 Mobile UI Development using Android.....	69
3.1 The Building Blocks of an Android App.....	70
3.1.1 Activity.....	71
3.1.1.1 The Lifecycle of an Activity.....	71
3.1.1.2 Activities interaction.....	72
3.1.2 Service: Background Worker.....	72
3.1.3 Broadcast Receiver: Listening to Events.....	74
3.1.4 Content Provider: Sharing Data.....	75
3.1.5 Intents.....	76
3.2 Android Manifest.....	77
3.3 Android XML Layout UI Basics.....	78
3.3.1 Layout XML-file.....	79
3.3.2 The basic Android Studio IDE Example.....	81
3.3.2.1 The project execution.....	87
3.4 Android Compose UI Layout.....	90
3.4.1 Common Layouts in Compose.....	91
3.5 State Management in Jetpack Compose.....	94
3.5.1 React vs Jetpack Compose.....	94
3.5.2 State in Jetpack Compose.....	95
3.5.2.1 Remembering State.....	95

3.5.2.2 State Hoisting.....	95
3.5.2.3 Using ViewModel for State Management.....	96
3.5.2.4 State and Side Effects.....	97
3.5.2.5 Derived State.....	98
4 Cross-platform UI Development using Flutter.....	100
4.1 Architectural layers.....	101
4.2 Setting Up the Environment.....	102
4.3 Dart language.....	103
4.4 Widgets.....	110
4.5 Stateful and Stateless Widgets.....	112
4.5.1 Widget Lifecycle.....	113
4.6 Flutter State Management. Revirepod.....	115
4.7 Streams in Flutter.....	117
References.....	120

1 The Evolution of UI Technologies

1.1 The Origins of User Interaction

In the early days of computing, user interfaces were entirely text-based, depending on command-line interfaces (CLI) for user interaction. Users needed to type precise commands to communicate with the computer, resulting in a significant learning curve. While CLIs were efficient for those familiar with the commands, they were not easily accessible to the general public. Early operating systems like UNIX, MS-DOS, and CP/M functioned exclusively through CLIs. Despite their challenging nature, CLIs offered robust access to system-level operations and continue to be utilized in various modern development and administrative tasks.

The development of graphical user interfaces (GUIs) in the 1970s and 1980s revolutionized human-computer interaction by introducing visual elements like windows, icons, menus, and pointers (WIMP model). GUIs made computers far more accessible by allowing users to interact with them through point-and-click navigation rather than memorizing commands. Here are some key milestones:

Xerox PARC (Palo Alto Research Center) (1973) developed the first true graphical user interface on the **Xerox Alto**, a pioneering research computer. The Alto introduced the WIMP paradigm, including overlapping windows, menus, and a mouse-driven pointer (Figure 1.1). Although never a commercial success, the Alto laid the foundation for the development of future graphical user interfaces. Xerox PARC's work would inspire a new generation of commercial GUIs.

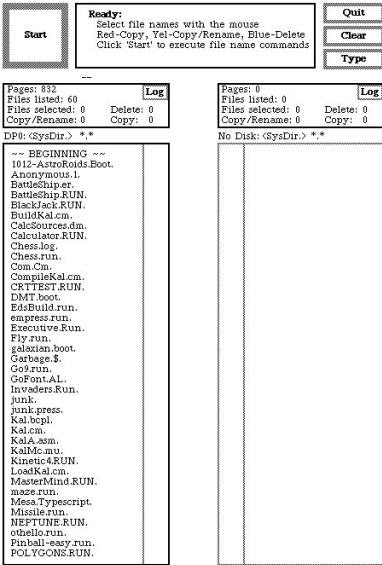


Figure 1.1. [Alto File Manager](#)

Apple Lisa (1983), heavily influenced by the Xerox PARC research, was one of the first personal computers to feature a graphical user interface aimed at business users (Figure 1.2). The Lisa introduced drag-and-drop capabilities, an icon-based file system, and a mouse for navigation. However, its high cost and slow performance hindered widespread adoption.

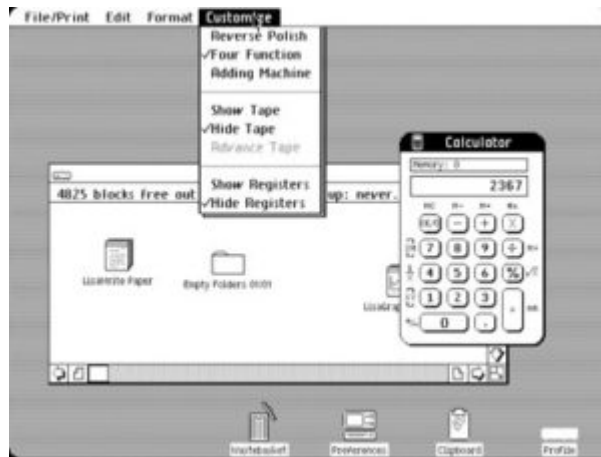


Figure 1.2. [Apple LISA GUI](#)

Apple Macintosh (1984). Shortly after the Lisa, Apple launched the **Macintosh**, which brought GUIs to a broader audience. It was much more affordable than the Lisa, and its GUI became a hallmark of Apple computers (Figure 1.3). The Macintosh popularized the use of icons, windows, and a mouse-driven interface, setting a new standard for user interaction in personal computing.



Figure 1.3. [Macintosh 1.1 User Interface](#)

Microsoft Windows released its first version in 1985, as a graphical extension of MS-DOS. Although primitive compared to the Macintosh, Windows eventually became the dominant GUI, evolving over time into a more sophisticated interface (Figure 1.4). By the 1990s, Windows 3.1 and Windows 95 introduced more advanced UI features like taskbars, file explorers, and the start menu, helping GUI adoption spread widely.

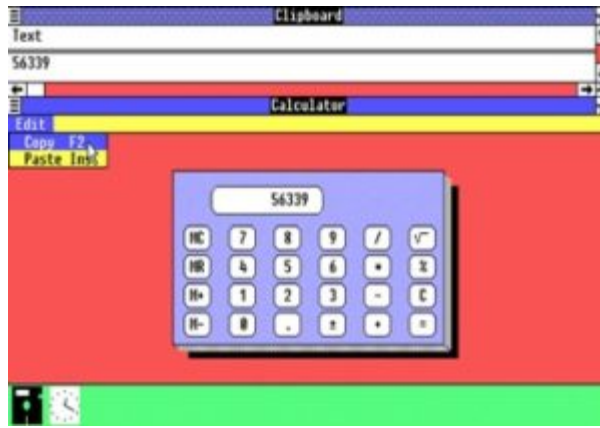


Figure 1.4. [*Windows 1.0 User Interface*](#)

These early graphical user interfaces made computing more accessible to non-technical users, transforming computers from specialized tools for professionals into essential devices for everyday life. They laid the groundwork for contemporary UI design principles, such as direct manipulation, consistency, and feedback, which still shape software interfaces today. The graphical user interface, initially developed by Xerox and further enhanced by companies like Apple and Microsoft, became the cornerstone for modern interfaces used in both desktop and mobile applications.

1.2 Transition to Modern UIs

As graphical user interfaces (GUIs) evolved, two dominant platforms, **Windows** and **macOS**, led the way in refining and advancing the capabilities of desktop environments.

Windows made significant strides in GUI design with the release of **Windows 95** in 1995, introducing the classic **Start menu**, taskbar, and system tray. This version enhanced multitasking, file management, and overall usability, setting a new standard for desktop environments. Later, **Windows XP**, launched in 2001, offered a more visually appealing interface with smoother graphics, an intuitive control panel, and customizable themes. Its user-friendly approach and stability contributed to its widespread popularity. Subsequent iterations, like **Windows 7**, **8**, and **10**, introduced improvements in window management, touch screen support, and integration with cloud services. **Windows 10** in particular streamlined the experience by combining the traditional Start menu with a modern tile-based interface and providing a consistent user experience across devices, including desktops, tablets, and phones.

On the other hand, **macOS** set early benchmarks in GUI design with its **Classic** interface (formerly known as System Software), which introduced consistent menu bars, windows, and icons, making user interaction intuitive. Its **drag-and-drop** functionality and desktop metaphor provided users with a clear visualization of

tasks. In 2001, **macOS X** marked a significant redesign, based on the NeXTSTEP OS, featuring the **Aqua** interface (Figure 1.5).



Figure 1.5. [*Mac OS X 10.1 Puma UI*](#)

This introduced visually appealing, fluid designs, including translucent elements, rounded buttons, and intuitive dock icons. Additionally, it brought advanced features like **Exposé** for window management, **Dashboard**, and **Spaces** for virtual desktops. Modern versions of **macOS** have continued to streamline interactions with features like **Mission Control**, **Launchpad**, and enhanced touchpad gestures, all while maintaining a clean, aesthetically pleasing design.

Both **Windows** and **macOS** advancements helped standardize the **WIMP** (windows, icons, menus, pointer) model across desktop systems, leading to greater ease of use and increased functionality for a wider audience.

As the **World Wide Web** gained prominence in the 1990s, web browsers became a critical part of the user interface landscape, offering a new way to access and interact with content. The rise of the web shifted the focus from local software to web-based applications. One of the first widely-used browsers, **Netscape Navigator**, introduced users to the concept of hyperlinks, web forms, and embedded media (Figure 1.6). With features like the **back button**, bookmarks, and multimedia support, Netscape set the standard for early web browsers. **Microsoft** quickly entered the space with **Internet Explorer**, which came bundled with **Windows 95**, giving millions of users easy access to the internet.



Figure 1.6. [Netscape Navigator 4.5 \(1998\)](#)

By the late 1990s and early 2000s, **Internet Explorer** had become the dominant browser, incorporating more advanced support for HTML, CSS, and JavaScript, thus enabling richer web experiences (Figure 1.7).

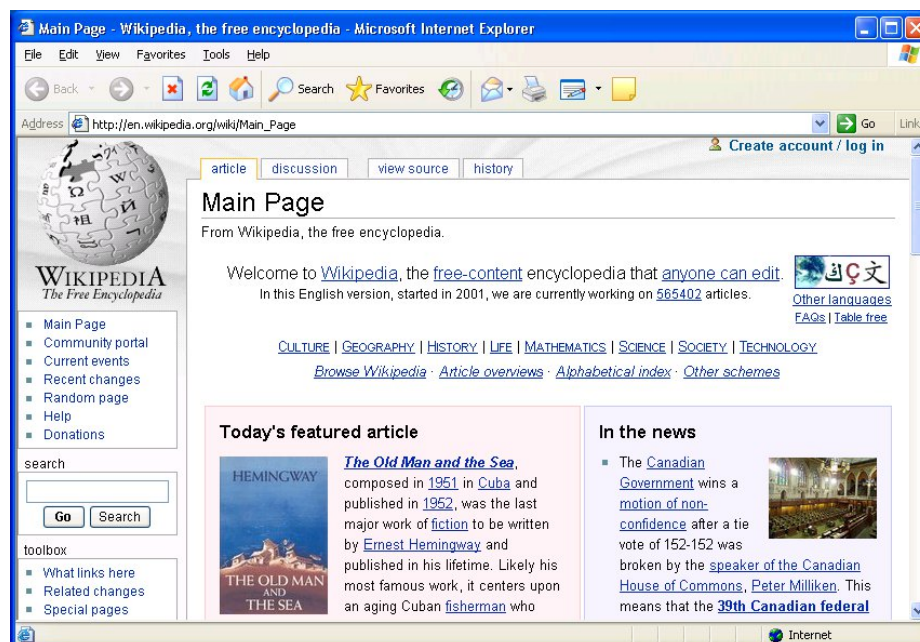


Figure 1.7. [Microsoft Internet Explorer 6](#)

The early 2000s saw the rise of modern browsers like **Mozilla Firefox**, which offered a faster, more secure alternative to Internet Explorer and popularized features such as tabbed browsing and extensive customization through extensions. In 2008, **Google Chrome** redefined the web browser with its minimalist interface, speed, and focus on stability. Chrome's emphasis on **web standards** and performance,

alongside the rise of **Web 2.0** technologies, like **AJAX**, paved the way for a new era of dynamic, JavaScript-driven web applications.

The web browser's evolution into a platform blurred the lines between desktop applications and web apps. Services like Google Docs, Slack, and Figma took advantage of web technologies to deliver rich, interactive experiences within the browser, reducing the need for native desktop software. As a result, modern web browsers have become essential tools, supporting cross-platform access, cloud-based services, and development tools.

The transition from desktop GUIs to web-based interfaces marked a significant shift in computing. Users no longer had to rely solely on local applications as the web evolved into a powerful platform for accessing software and services. This convergence of web and desktop UI capabilities laid the groundwork for today's cross-platform technologies, including **Progressive Web Apps (PWAs)**, **ElectronJS**, and mobile-friendly frameworks like **React Native** and **Flutter**. The evolution of the web into a powerful, universal platform has transformed how we interact with digital interfaces across various devices.

1.3 The Mobile Revolution

The mobile revolution began with the introduction of touch interfaces, changing the way people interacted with devices. Instead of using keyboards or styluses, users could now directly manipulate on-screen elements with gestures like tapping, swiping, and pinching, making mobile devices much more user-friendly.

In 2007, Apple launched the iPhone (Figure 1.8), a major turning point in user interface design. The iPhone replaced physical buttons and styluses with a capacitive multi-touch screen, using gestures like swipe-to-unlock and pinch-to-zoom to make interaction smooth and natural. A year later, the App Store allowed developers to create apps specifically for this new touch-based platform, further revolutionizing mobile interaction.

In response, Google launched Android in 2008 with the HTC Dream (G1), helping smartphones become widely available. Android, being open-source, allowed for a variety of devices with similar touch capabilities. It also introduced gesture-based navigation, like swiping between apps and pulling down notification panels, making touch interactions even more intuitive.

As mobile UIs advanced, multitouch gestures became more refined, allowing users to switch between apps or zoom with two fingers. Both iOS and Android developed touch-friendly UI components like buttons and sliders, designed for easy interaction. Haptic feedback (vibrations) also improved the sense of responsiveness during touch interactions.



Figure 1.8. [*iPhone 1 \(2007\)*](#)

With the rise of mobile devices, web developers needed to make websites work across many screen sizes. This led to the development of responsive web design (RWD) in the 2010s. Responsive design allows websites to adjust to different screen sizes and orientations, providing an optimal viewing experience on desktops, tablets, and smartphones. The term "responsive design" was coined by Ethan Marcotte in 2010, and it emphasized the use of fluid grids, flexible images, and media queries.

At the same time, the mobile-first approach became popular. This design philosophy prioritizes mobile devices, ensuring key content and features work well on smaller screens before enhancing the design for larger ones. Benefits of mobile-first design include simpler, more user-focused interfaces and faster page load times, which is important for mobile users with slower internet or limited data.

Key techniques for responsive and mobile-first design include fluid grids, CSS Flexbox, and media queries that allow layouts to scale based on screen size. Developers also use responsive images and the viewport meta tag to make sure websites display correctly on mobile devices.

This shift to responsive and mobile-first design has had a lasting impact. Users now expect consistent experiences across devices, whether they're using a smartphone, tablet, or desktop. Native mobile UI elements like hamburger menus and card-based layouts have also influenced web design. The rise of Progressive Web Apps (PWAs) further blends web and app experiences, offering features like offline access and push notifications.

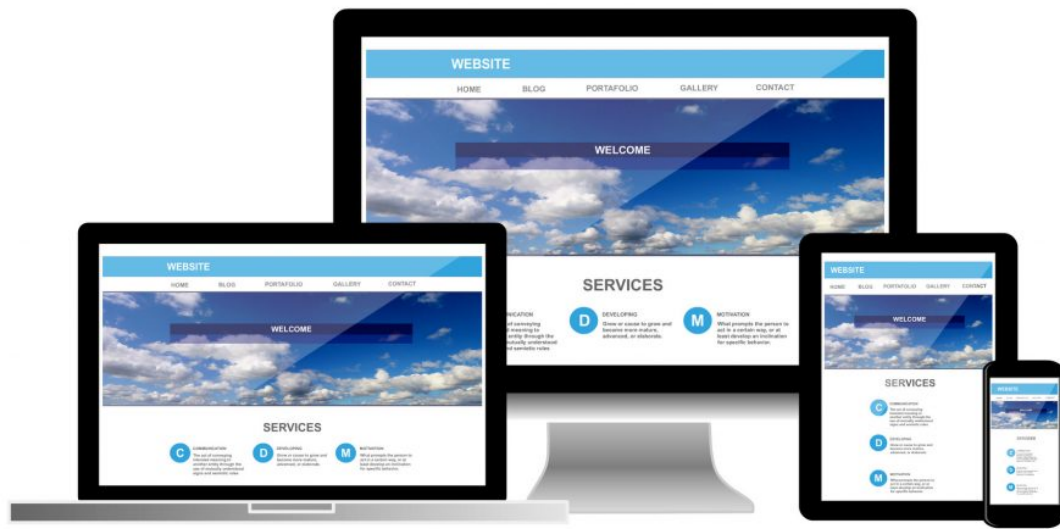


Figure 1.9. [Mobile-first design](#)

Overall, the mobile revolution, driven by touch interfaces and responsive design, transformed how we design user interfaces. Today, creating seamless experiences across devices is a core part of UI/UX design [1], ensuring that users can access content smoothly from anywhere, at any time.

1.4 Current Trends in User Interfaces

Single Page Applications (SPAs) are web apps that load a single HTML page and update content dynamically as users interact with them, avoiding full page reloads. This provides a more seamless and app-like experience, reducing server load and enhancing user engagement with faster interactions.

SPAs use technologies like AJAX or the fetch API to load content in the background, allowing smooth navigation without reloading the whole page. These apps rely on client-side rendering (CSR), meaning JavaScript frameworks like ReactJS, Vue.js, or Angular handle content display, routing, and managing app state directly on the user's device. The result is a fast, fluid interface that mimics the performance of native desktop or mobile apps.

Popular examples of SPAs include Gmail, Google Maps, Facebook, and Twitter, all offering quick, interactive experiences that keep users engaged. However, SPAs can face challenges with search engine optimization (SEO) since much of the content is loaded dynamically. Technologies like server-side rendering (SSR) and prerendering help mitigate this issue. Another potential downside is performance—

while SPAs reduce the need for full page reloads, the initial load time can be longer due to large JavaScript files, requiring techniques like code-splitting to improve performance.

Progressive Web Apps (PWAs) combine the advantages of both web and native apps, offering the speed, reliability, and engaging features of native apps directly through a browser. Built with standard web technologies like HTML, CSS, and JavaScript, PWAs feel like traditional mobile apps but come with extra features.

One of the key benefits of PWAs is offline access, made possible through service workers that cache content, allowing users to continue using the app even without an internet connection. They can also send push notifications, keeping users updated and engaged just like native apps. PWAs are installable on the user's home screen without needing to go through an app store, making them a lightweight alternative to traditional apps.

PWAs like Twitter Lite, Google Maps Go (Figure 1.10), and Spotify offer fast, responsive experiences across all devices, working offline and sending notifications.

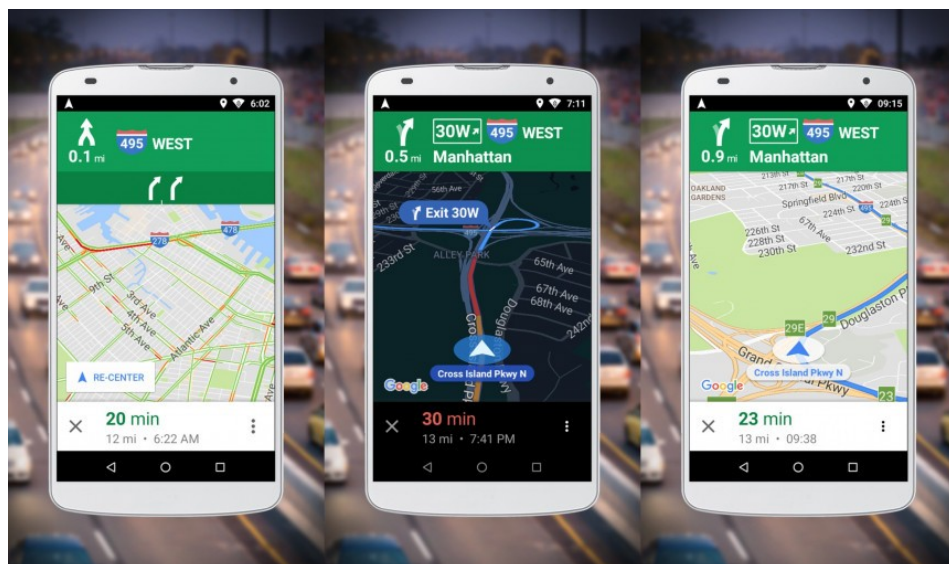


Figure 1.10. [Google Maps Go](#)

They provide nearly the same functionality as native apps through the web, while also reducing development costs since they work across all modern browsers and devices. PWAs are optimized to minimize data use, making them a great option for users with limited connectivity or slower networks. They deliver an app-like experience without the large downloads or the need for app store installations.

1.5 Importance of UX and Design in UI

User Experience, or UX, is about the overall feel of a product and how the user interacts with it. It focuses on making the product easy, efficient, and enjoyable to use. When designing UX, the goal is to understand what users need, what problems

they might face, and how to make their experience as smooth and satisfying as possible. Good UX means users can achieve their goals with minimal frustration, whether they are browsing a website, using an app, or interacting with any digital platform (Figure 1.11). It's about guiding users intuitively, so they know where to go, what to do, and feel pleased with the journey.

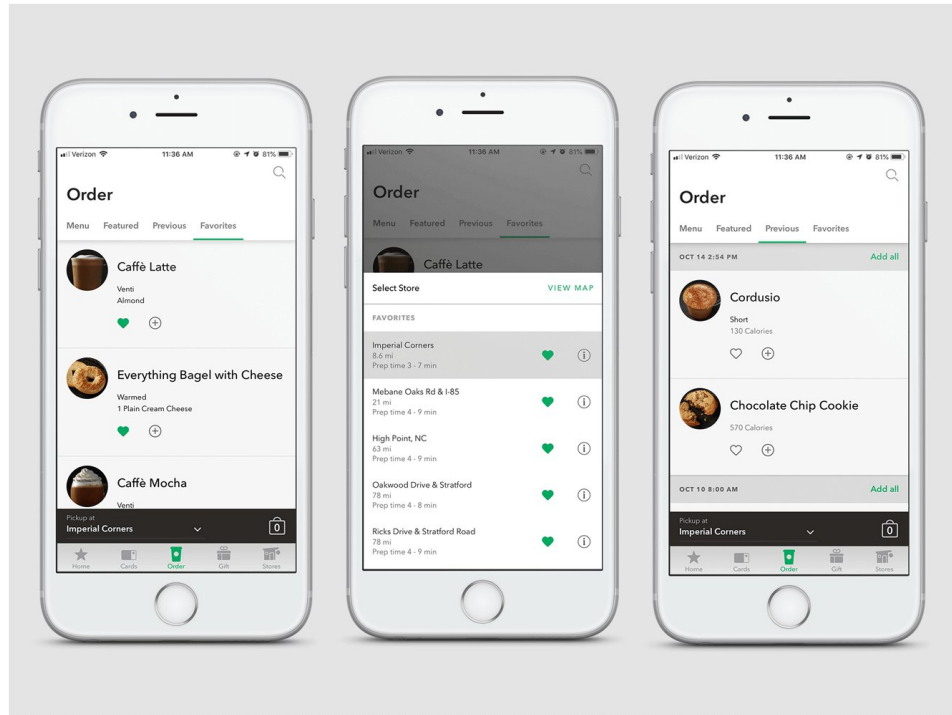


Figure 1.11. [*Starbucks: Making it Personal*](#)

User Interface, or UI, refers to the visual elements that users see and interact with. This includes buttons, icons, colors, layouts, and fonts—essentially, everything that makes up the appearance of the product. UI is responsible for making the product visually appealing and consistent, ensuring it aligns with the brand and is pleasant to look at. It's also about clarity; good UI helps users recognize actions and navigate the interface confidently, making the design feel effortless (Figure 1.12).

Together, UX and UI create a product that is both beautiful and functional. UX ensures that the product feels right and works smoothly, guiding users through their tasks without confusion [1]. Meanwhile, UI makes sure everything looks cohesive and appealing, allowing users to interact with the product easily. A well-designed product considers both, as UX without UI can feel bland or unappealing, and UI without UX can look good but be frustrating to use. By balancing UX and UI, designers create products that not only look great but also provide a positive and effective user experience [2].

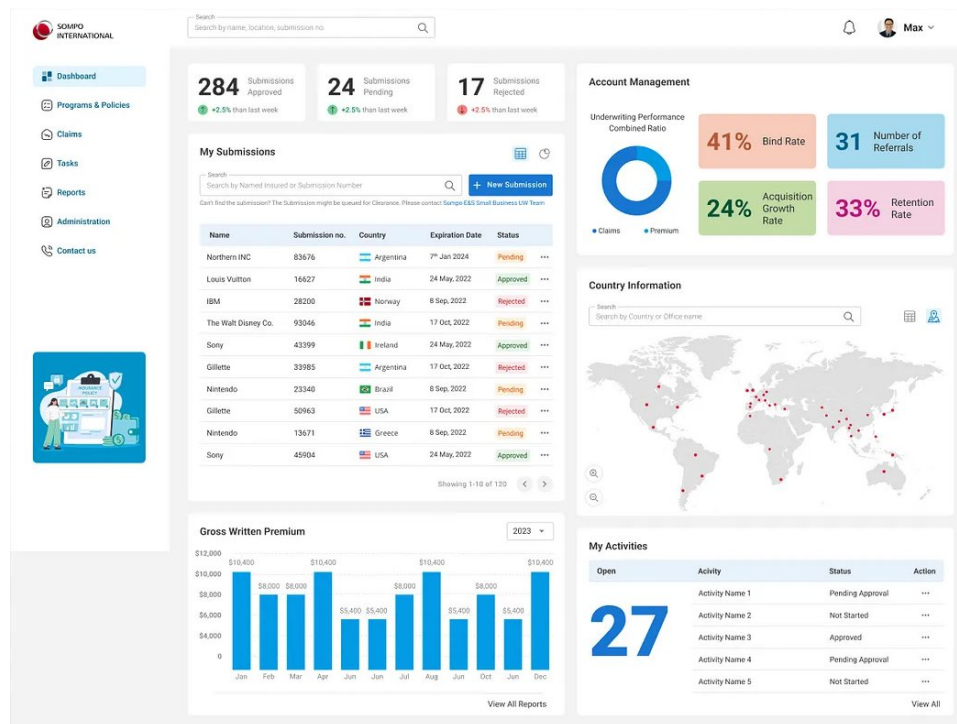


Figure 1.12. [UI Design: Dashboard](#)

1.6 Overview of Client Technologies

Client technologies refer to the tools, platforms, and frameworks used to develop applications and interfaces that run directly on users' devices, allowing them to interact with content and services [3]. These technologies are commonly divided into four main categories: Web, Mobile, Desktop, and Cross-platform. Each serves unique purposes but often overlaps in functionality and design principles.

Web technologies are foundational for creating applications accessed through browsers on various devices, from desktops to smartphones. Web development primarily involves HTML, CSS, and JavaScript, along with popular frameworks like React, Vue, and Angular, which simplify building interactive user interfaces. Web applications are inherently platform-agnostic and accessible on any device with a browser, making them widely compatible and easy to update. However, they may have limitations in accessing hardware features or running offline without special techniques like Progressive Web Apps (PWAs).

Mobile technologies are tailored for applications running on smartphones and tablets, with iOS and Android as the primary platforms. Mobile apps can be native, using languages like Swift (iOS) and Kotlin (Android), which allow optimal performance and full access to device capabilities, including GPS, cameras, and sensors. Alternatively, hybrid and cross-platform frameworks like React Native and Flutter enable developers to write a single codebase for both platforms, speeding up development while still delivering near-native performance and appearance.

Desktop technologies are used for applications on traditional computers, often providing more advanced functionality suited to professional or resource-intensive tasks. Historically built using languages like Java, C++, or C#, desktop applications can access powerful system resources and hardware. Recently, frameworks like Electron have allowed web technologies to be repurposed for desktop applications, enabling developers to use familiar languages like JavaScript, HTML, and CSS. This makes it possible to build cross-platform desktop apps with one codebase, though it may come at the cost of performance compared to fully native apps.

Cross-platform technologies aim to provide a unified development approach that works across Web, Mobile, and Desktop. Frameworks like Flutter and React Native enable developers to create a single codebase that can deploy to multiple platforms, significantly reducing development time and resources.

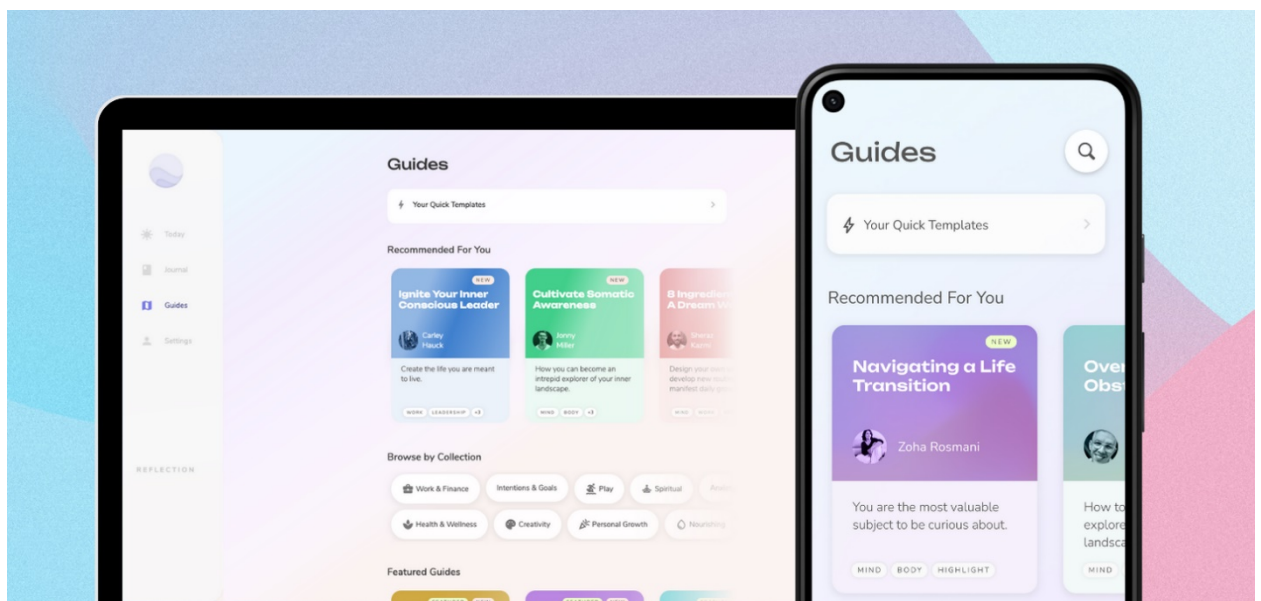


Figure 1.13. [Flutter Cross-platform application: reflection.app](#)

Although there may be compromises in platform-specific functionality or performance, cross-platform solutions are ideal for delivering a consistent experience across different devices and operating systems, particularly for applications where uniformity and efficiency are prioritized over highly specialized features.

2 Web UI Development Using React

Web UI development has evolved significantly, beginning with static HTML pages and gradually incorporating CSS for styling and JavaScript for interactivity. Early web interfaces required reloading entire pages with each interaction, which often led to slower, less seamless user experiences. The shift to Single Page Applications (SPAs) brought a new approach, allowing content to update dynamically without full page reloads, making websites feel faster and more like native applications. Today, HTML, CSS, and JavaScript remain the core technologies for building modern, interactive web interfaces, with frameworks and libraries further enhancing development speed and user experience.

2.1 React Essentials

React is a popular JavaScript library for building user interfaces, especially for web applications [4]. Developed by Facebook, it has become a go-to tool for developers because of how it simplifies the creation of complex and interactive user interfaces. At its core, React is all about making it easier to build responsive, dynamic websites and apps where things change on the screen without having to reload the entire page. This is a big deal because, traditionally, creating these dynamic experiences required complex coding and a lot of manual updates, but React streamlines it all.

One of the reasons React has become so widely used is its **component-based architecture**. Think of components as the building blocks of a web page—each part of a site, like buttons, forms, or images, can be built as a separate piece, making them easy to work with individually and reusable across different parts of an application. When a developer builds a new feature in React, they can quickly create, test, and even reuse these components across various sections of the application. This makes development not only faster but also more organized and efficient.

Another defining feature of React is how it manages changes on a page. It uses so called a “**virtual DOM**” (Document Object Model), which is like a simplified copy of the actual webpage structure in memory. When something changes, React updates only the necessary parts on the screen without refreshing the whole page, resulting in faster and smoother user experiences. This approach is efficient and gives React a performance edge, making websites feel quick and responsive even with a lot of interactive elements.

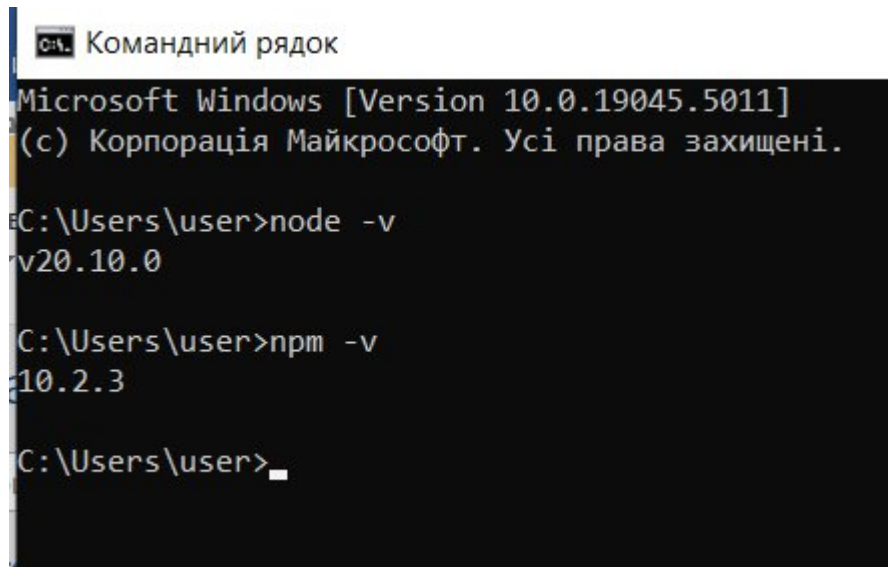
React also allows developers to think about their interfaces as a flow of data, where each piece of the UI automatically updates when the underlying data changes. For example, if a shopping cart count updates in the background, React takes care of changing the displayed count on the screen instantly without extra coding. This data-

driven approach simplifies the coding process and keeps the user experience seamless.

2.2 ReactJS Installation and Run

To install and run a basic ReactJS application, follow these steps.

1. Install Node.js: Ensure that Node.js and npm (Node Package Manager) are installed. You can download it from nodejs.org. Then check them by running the following commands in the terminal (Figure 2.1):

A screenshot of a Windows Command Prompt window. The title bar reads 'Командний рядок' (Command Prompt). The window content shows the following text: 'Microsoft Windows [Version 10.0.19045.5011] (c) Корпорація Майкрософт. Усі права захищені.' followed by a prompt 'C:\Users\user>'. The user has entered 'node -v' and the output is 'v20.10.0'. Then the user has entered 'npm -v' and the output is '10.2.3'. The prompt 'C:\Users\user>' is shown again at the bottom.

```
Командний рядок
Microsoft Windows [Version 10.0.19045.5011]
(c) Корпорація Майкрософт. Усі права захищені.

C:\Users\user>node -v
v20.10.0

C:\Users\user>npm -v
10.2.3

C:\Users\user>
```

Figure 2.1. Node.js and NPM versions

2. Create a New React Application: Use the Create React App command-line tool to set up a new React application. Open the terminal and run:

```
npx create-react-app polling-app
```

This command creates a new directory called `polling-app` with a default React application setup.

3. Navigate to the Application Directory: Change into the newly created application directory:

```
cd polling-app
```

4. Run the Application: Start the development server to run the application:

```
npm start
```

This will start the app and open it in a web browser, typically at `http://localhost:3000`. The default app should display a welcome message with the React logo.

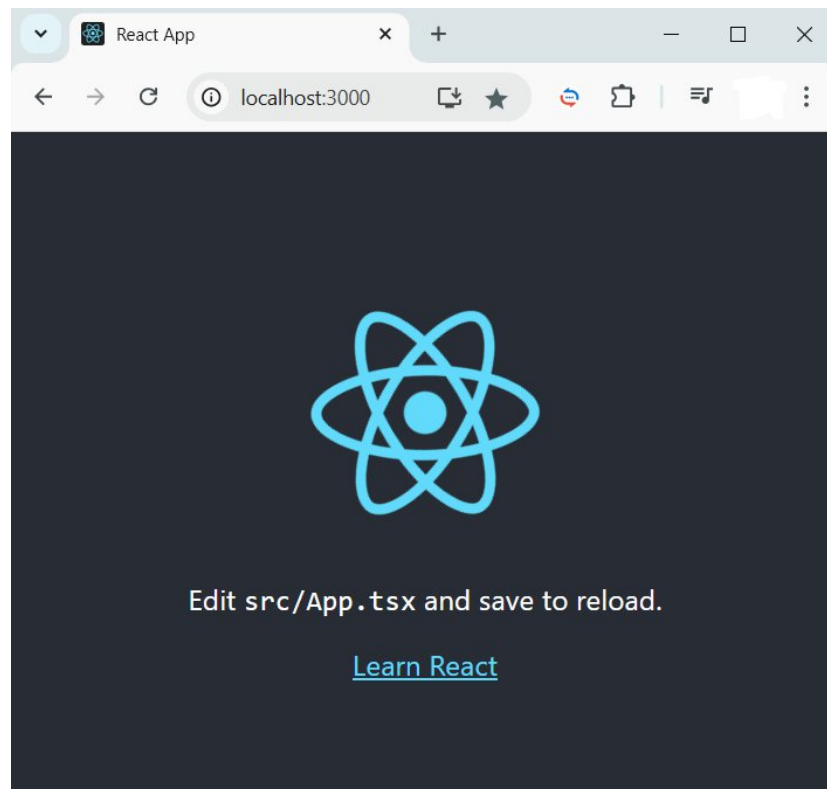


Figure 2.2. The default React application

2.3 The Default Folder and File Structure

When you create a new React app using `npx create-react-app`, the default folder and file structure provides an organized setup for building, styling, and deploying the application. Here's an overview of the main folders and files:

node_modules/ Contains all the installed dependencies for the React project. Don't modify this folder directly.

public/ Holds static files that do not change during the app's lifecycle.

index.html The main HTML file that serves as the entry point for the app. React components are injected here.

favicon.ico The website's favicon.

manifest.json Describes the app's metadata (like name, icons) for Progressive Web Apps.

Other assets: Any other static assets (e.g., images) that should be accessible from the root.

src/ The main folder for the React application code.

index.js The entry JavaScript file where React DOM renders the `App` component into `index.html`. It's also where libraries and global styles can be imported.

App.js The root component for the application. Typically includes high-level layout and routes.

App.css Default styling for the App component.

index.css Global styles that apply across the app.

Key Files

.gitignore Specifies files and folders that should be ignored by Git (e.g., `node_modules`, environment files).

package.json Lists dependencies, scripts, and metadata for the app. Defines how to run and build the project.

README.md A basic introduction to the app. It's customizable and often includes project instructions and documentation.

Typical Modifications

src/components/ A folder to hold reusable components. Developers commonly create a `components/` directory to organize React components.

src/assets/ Stores images, icons, and other static files used within the application.

src/hooks/ Custom hooks to manage specific logic that can be reused across components.

src/context/ Holds context providers for managing global state with React Context API.

src/utils/ Utility functions or helper files.

As it was stated above **index.js** is the main entry point of a React application (created with Create React App). It has two main responsibilities:

- 1) Mounting the React application to the DOM:

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

This code finds an HTML element with `id="root"` and renders the main App component inside it, wrapped in React's Strict Mode for additional development checks.

- 2) Setting up performance monitoring through `reportWebVitals()` (optional feature that can be removed if you don't need performance monitoring).

```
reportWebVitals();
```

2.4 React Components

Components are the building blocks of React applications. They are reusable pieces of UI (User Interface) that can be combined to create complex applications. Each component is a self-contained module that can render some output and manage its own logic and state.

Let's take a simple example of a React component: a **button** that keeps track of the number of times it's been clicked.

In React, each component is typically a function or class that returns some JSX (JavaScript XML) code, which looks similar to HTML. This JSX is then rendered as UI. Components can also manage their own **state**, which allows them to keep track of data that changes over time.

```
import React, { useState } from 'react';

function CounterButton() {
  // useState is a hook that allows us to add state to this functional component
  const [count, setCount] = useState(0);
  // Function to increment count
  const handleClick = () => {
    setCount(count + 1); // Update the state by incrementing count
  };
  return (
    <button onClick={handleClick}>
      Clicked {count} times
    </button>
  );
}
```

```
export default CounterButton;
```

When we start off, we use `useState(0)` to set up a piece of state called `count`, and we give it a starting value of 0. This count will keep track of how many times the button has been clicked. We also get a handy function, `setCount`, that lets us update count whenever we want.

Now, each time the button is clicked, we want the count to go up by one. So, we create a function called `handleClick` that runs every time the button is clicked. Inside this function, we call `setCount(count + 1)`, which tells React to update count to its current value plus one.

As for the button itself, it's showing some text that says, "Clicked {count} times." The {count} part is replaced by the actual value of count, so if it's been clicked three times, it will show "Clicked 3 times." We make the button clickable by adding `onClick={handleClick}`, which means every click will trigger our `handleClick` function, updating the count on the button.

2.4.1 Component Composition

Component composition in React is all about building complex interfaces by nesting components within each other. This way, we can create a "component tree" where each component is responsible for a specific piece of the UI.

Imagine we're building a **Profile Card** that shows a user's profile picture, name, and bio. Instead of putting all the code for this card in one large component, we can break it down into smaller, reusable components: one for the profile picture, one for the user's name, and another for the bio.

Here's how this composition might look (Figure 2.3):

1. **ProfileCard**: The main component that holds everything together.
2. **ProfilePicture**: A component to display the user's profile picture.
3. **ProfileName**: A component for displaying the user's name.
4. **ProfileBio**: A component to show the bio text.

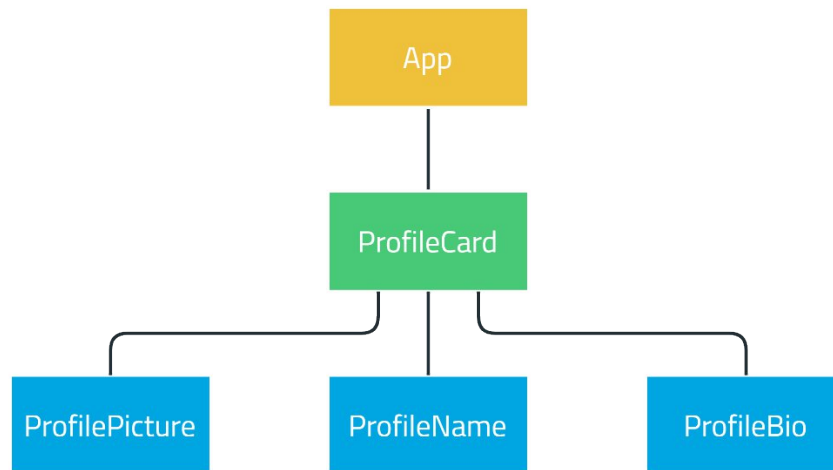


Figure 2.3. The components tree

The updated App.js includes reference to the ProfileCard component and will look like this:

```
import logo from './logo.svg';
import './App.css';
import ProfileCard from './components/ProfileCard';
function App() {
  return (
    <div className="App">
      <ProfileCard imageUrl={logo} name="John Smith" bio="I am a software engineer" />
    </div>
  );
}
export default App;
```

and **ProfileCard** component (ProfileCard.js) in turn contains references to the rest of the components (**ProfilePicture**, **ProfileName** and **ProfileBio**):

```
import React from 'react';
// Small components
function ProfilePicture({ imageUrl }) {
  return <img src={imageUrl} alt="Profile" style={{ width: 100, borderRadius: '50%' }} />;
}
```

```

function ProfileName({ name }) {
  return <h2>{name}</h2>;
}

function ProfileBio({ bio }) {
  return <p>{bio}</p>;
}

// Main ProfileCard component
function ProfileCard({ imageUrl, name, bio }) {
  return (
    <div style={{ border: '1px solid #ddd', padding: '20px', borderRadius: '10px',
textAlign: 'center' }}>
      <ProfilePicture imageUrl={imageUrl} />
      <ProfileName name={name} />
      <ProfileBio bio={bio} />
    </div>
  );
}

export default ProfileCard;

```

The final result will look like the following (Figure 2.4). The colorful borders around the components show their boundaries and locations.

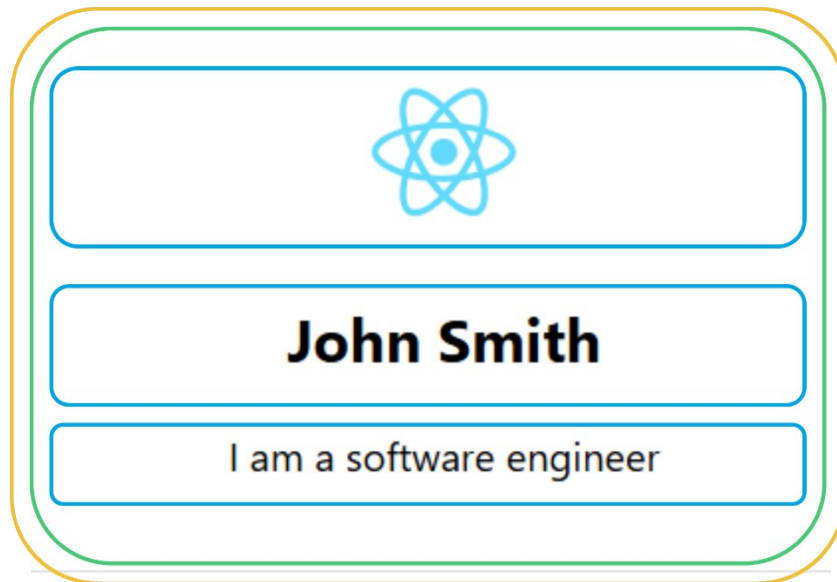


Figure 2.4. The final result

2.4.2 Understanding JSX

JSX (JavaScript XML) is a syntax extension for JavaScript that allows us to write HTML-like code within your JavaScript code. It was developed by Facebook for React and makes writing React components more intuitive and readable.

At its core, JSX looks similar to HTML. Here's a basic example of JSX in action:

```
const element = <h1>Hello, world!</h1>;
```

This line creates a React element, an object that React will eventually turn into a DOM element. JSX lets us directly write tags like `<h1>`, keeping our code clear and resembling the final UI.

One of the powerful features of JSX is its ability to embed JavaScript expressions directly within markup. By wrapping JavaScript in curly braces `{}`, we can dynamically render variables, execute functions, or use any valid JavaScript expression. For instance:

```
const name = "Alice";  
const element = <h1>Hello, {name}!</h1>;
```

Here, `{name}` is evaluated as a JavaScript expression, so the resulting element will display "Hello, Alice!"

In React, we use JSX to define components, the building blocks of our application. Let's say we're creating a simple `Welcome` component that greets a user by name:

```
function Welcome(props) {  
  return <h1>Hello, {props.name}!</h1>;  
}
```

```
}  
const element = <Welcome name="Alice" />;
```

In this example, `<Welcome name="Alice" />` passes the name "Alice" as a prop to the `Welcome` component, which then displays it inside the heading.

JSX supports attributes similar to HTML, with a few key differences:

- Use `className` instead of `class`, as `class` is a reserved word in JavaScript.
- Use `camelCase` for attributes like `onClick` and `tabIndex`.

For example:

```
const button = <button className="primary-button" onClick={handleClick}>Click  
me</button>;
```

Here, `className` applies a CSS class to the button, and `onClick` triggers a function when the button is clicked

Conditional rendering allows us to display different content based on conditions. Using JavaScript expressions in JSX, we can write inline conditions directly in our components. For instance:

```
const isLoggedIn = true;  
const message = <h1>{isLoggedIn ? "Welcome back!" : "Please sign in"}</h1>;
```

This code displays "Welcome back!" if `isLoggedIn` is `true`; otherwise, it shows "Please sign in."

To render lists of items, JSX lets us loop over arrays with JavaScript's `map()` function. Each item in the list needs a unique `key` to help React track changes efficiently:

```
const fruits = ["Apple", "Banana", "Cherry"];  
const listItems = fruits.map((fruit) => <li key={fruit}>{fruit}</li>);  
const element = <ul>{listItems}</ul>;
```

In this example, each fruit is rendered as a `li` inside an unordered list. The `key` prop is required for list elements to maintain the order and identity of each item.

JSX elements can be nested to form more complex structures. Here's an example of a simple JSX layout:

```
const element = (  
  <div>  
    <h1>Welcome</h1>
```

```
<p>This is a sample nested JSX structure.</p>
</div>
);
```

JSX expressions must have a single parent element. If multiple elements are at the top level, they need to be wrapped in a single container. We can use a `div` or, if we don't want extra markup, a `React.Fragment` or empty tags `<>` `</>`:

```
const element = (
  <>
    <h1>Title</h1>
    <p>This paragraph is part of the same parent.</p>
  </>
);
```

2.4.3 Props in React

Props, short for "properties," are a core concept in React used to pass data from one component to another. They allow us to create dynamic, reusable components that adapt based on the data they receive. We can think of props as a component's way of receiving inputs—data or functions—from its parent component.

Props are passed to a component much like attributes in HTML. For instance, if we want to pass a name and an age to a `UserProfile` component, it would look like this:

```
<UserProfile name="Alice" age={25} />
```

In this example `name` and `age` are props. `"Alice"` and `25` are the values of these props, which `UserProfile` will receive and use.

Props are accessed in the component as an object. If you're using a functional component, props are passed as the first argument:

```
function UserProfile(props) {
  return (
    <div>
      <h1>{props.name}</h1>
      <p>Age: {props.age}</p>
    </div>
  );
}
```

```
}
```

Alternatively, we can use object destructuring to access props more cleanly:

```
function UserProfile({ name, age }) {  
  return (  
    <div>  
      <h1>{name}</h1>  
      <p>Age: {age}</p>  
    </div>  
  );  
}
```

With this approach, `name` and `age` are extracted directly from props, which can make the code easier to read.

It's important to remember that props are **read-only**. This means that a component should never try to modify its own props. Props are meant to be received and displayed or used to influence the component's behavior, not changed. If you need to change data, it should be done in a parent component's state or through other data management techniques.

Props are particularly useful when creating reusable components that need to vary slightly based on their inputs. For example, consider a `Button` component that might receive a `color` prop and a `label` prop:

```
function Button({ color, label }) {  
  return <button style={{ backgroundColor: color }}>{label}</button>;  
}
```

// Using the Button component with different props:

```
<Button color="blue" label="Save" />  
<Button color="green" label="Submit" />  
<Button color="red" label="Delete" />
```

Here, each button will have a different color and label based on the props it receives, making it highly reusable.

Props are not limited to simple data; we can also pass functions as props. This is especially useful for handling events or updating state in a parent component. For

instance, if we want a button component to trigger an action in its parent component, we could pass a function as a prop:

```
function ParentComponent() {  
  function handleClick() {  
    alert("Button clicked!");  
  }  
  return <Button onClick={handleClick} label="Click Me" />;  
}  
  
function Button({ onClick, label }) {  
  return <button onClick={onClick}>{label}</button>;  
}
```

While both props and state hold data, they serve different purposes:

- **Props** are passed from parent to child and are fixed from the child component's perspective.
- **State** is managed within the component and can be updated to reflect changes over time.

2.4.4 Component State

State in React is a built-in way to keep track of changing data within a component. Unlike props, which are received from a parent component and remain fixed from the child's perspective, state is internal and can be updated by the component itself. When state changes, the component re-renders to reflect the updated values, allowing React to create interactive and dynamic user interfaces.

In functional components, we use the `useState` hook to define state. The `useState` function lets us declare a state variable and a function to update it. Here's how it works:

```
import React, { useState } from 'react';  
  
function Counter() {  
  const [count, setCount] = useState(0);  
  return (  
    <div>  
      <p>Current count: {count}</p>  
      <button onClick={() => setCount(count + 1)}>Increase</button>  
    </div>  
  );  
}
```

```
</div>

);
}
```

In this example, we have a variable called `count` that we use to keep track of something. At the beginning, `count` is set to 0.

We also have a function called `setCount` that we use to change the value of `count`. Whenever we click a button, `setCount(count + 1)` is called, which adds 1 to the current value of `count`.

The component re-renders with the updated value of `count`, displaying the new count in real-time.

When we update state in React, it doesn't happen instantly. Instead, state updates are asynchronous, meaning React batches multiple updates to improve performance. This approach means that if you try to rely on the current state value for multiple updates in one function, you might get unexpected results. To handle this, `setState` can also accept a function that takes the previous state as an argument:

```
<button onClick={() => setCount((prevCount) => prevCount + 1)}>
  Increase
</button>
```

This pattern ensures that each update uses the latest state, making it reliable even in complex scenarios.

We can use multiple `useState` calls to handle different pieces of state in the same component. For instance, if we're tracking a user's `name` and `age`, we can set up state for each:

```
function UserInfo() {
  const [name, setName] = useState("Alice");
  const [age, setAge] = useState(25);
  return (
    <div>
      <h1>{name}</h1>
      <p>Age: {age}</p>
    </div>
  );
}
```

```
}
```

Each call to `useState` creates an independent piece of state, allowing us to update name and age separately.

State is central to React's reactivity. Whenever state changes, React triggers a re-render of the component to ensure the UI reflects the new data. This process is efficient; React only re-renders the affected parts, not the entire app.

Using state, we can control what a component displays based on the current values. For instance, we might want to show different messages based on whether a user is logged in or not:

```
function Greeting() {  
  const [isLoggedIn, setIsLoggedIn] = useState(false);  
  return (  
    <div>  
      <h1>{isLoggedIn ? "Welcome back!" : "Please sign in"}</h1>  
      <button onClick={() => setIsLoggedIn(!isLoggedIn)}>  
        {isLoggedIn ? "Sign out" : "Sign in"}  
      </button>  
    </div>  
  );  
}
```

Here, the greeting message and button text update based on the `isLoggedIn` state. When the button is clicked, `setIsLoggedIn` toggles the user's status, switching the message and button label.

State is often used to manage form inputs, as it lets us capture and respond to user input. In a form component, we might track the value of an input field in state and update it as the user types:

```
function Form() {  
  const [inputValue, setInputValue] = useState("");  
  return (  
    <form>  
      <input  
        type="text"
```

```

    value={inputValue}
    onChange={(e) => setInputValue(e.target.value)}
  />
  <p>Input: {inputValue}</p>
</form>
);
}

```

Here, `inputValue` is bound to the input field. As the user types, `setInputValue` updates the state, which re-renders the component to display the latest input.

2.4.5 Component Lifecycle with `useEffect`

In React, components have lifecycles—phases they go through from creation, updating, and removal. In functional components, we use the `useEffect` hook to manage these lifecycle events. `useEffect` allows us to perform side effects, like fetching data, updating the DOM, or setting up timers, in a clean and organized way.

The `useEffect` hook takes two arguments:

1. A **callback function** where we place our side effect logic.
2. A **dependency array** (optional), which determines when the effect should run.

Here's the basic syntax:

```

useEffect(() => {
  // Side effect code here
}, [/* dependencies */]);

```

If **the dependency array is empty**, the effect runs only once, after the initial render.

If **there are dependencies**, the effect runs whenever one of them changes.

If **no dependency array is provided**, the effect will run after every render.

Using `useEffect`, we can control the behavior of the three main lifecycle phases in components:

1. **Mounting**
2. **Updating**
3. **Unmounting**

Mounting: Running an Effect Once

To run an effect only once when a component mounts, we provide an empty dependency array [] to `useEffect`. This setup is useful for tasks like fetching data or setting up subscriptions that only need to happen once.

Example:

```
import React, { useEffect, useState } from 'react';

function Profile() {
  const [user, setUser] = useState(null);

  useEffect(() => {
    // Fetch data only once on component mount
    fetch('/api/user')
      .then(response => response.json())
      .then(data => setUser(data));
  }, []); // Empty dependency array

  return (
    <div>
      {user ? <h1>{user.name}</h1> : <p>Loading...</p>}
    </div>
  );
}
```

Here, the effect runs once after the initial render to fetch user data. The empty dependency array ensures it doesn't re-run on subsequent renders.

Updating: Running an Effect on Dependency Change

When we want the effect to run whenever certain variables change, we list those variables in the dependency array. This approach is useful for effects that depend on dynamic values, like responding to user inputs or changes in props.

Example:

```
function Counter({ step }) {  
  const [count, setCount] = useState(0);  
  
  useEffect(() => {  
    console.log(`Count changed to ${count}`);  
  }, [count]); // Effect depends on `count`  
  
  return (  
    <div>  
      <p>Count: {count}</p>  
      <button onClick={() => setCount(count + step)}>Increase</button>  
    </div>  
  );  
}
```

In this example, we want to do something every time the count variable changes. Specifically, we want to print a message to the console.

We use a special tool called `useEffect` to achieve this. By telling `useEffect` to watch the count variable, it will automatically run the code inside the effect whenever count changes. This ensures that our message is logged each time count updates.

Unmounting: Cleaning Up with `useEffect`

To clean up resources or cancel ongoing processes when a component unmounts, we return a cleanup function from the `useEffect` callback. This function runs automatically before the component is removed from the DOM, or whenever dependencies change (if the effect is set to re-run on updates).

Example:

```
function Timer() {  
  const [seconds, setSeconds] = useState(0);  
  
  useEffect(() => {  
    const interval = setInterval(() => {
```

```

    setSeconds((prev) => prev + 1);
  }, 1000);

  // Cleanup function to clear the interval on unmount
  return () => clearInterval(interval);
}, []); // Empty dependency array: only set up once on mount

return <p>Time elapsed: {seconds} seconds</p>;
}

```

In this example, we want to create a timer that counts seconds. We set up a repeating task, or "interval," that adds 1 to the seconds count every second.

To avoid unnecessary work and potential problems, we need to make sure the timer stops when we're no longer using the component. We do this by including a "cleanup function" that cancels the interval. This prevents the interval from running indefinitely, which could lead to memory issues.

2.5 Hooks in React

Hooks are special functions introduced in React to **allow state and lifecycle features** in functional components. Before hooks, only class components could manage state and side effects. With hooks, functional components can be more powerful and flexible, making code cleaner and easier to understand [5].

Key Benefits of Using Hooks

1. **State and Side Effects in Functional Components:** Hooks allow us to manage component state and side effects directly in functional components.
2. **Reusable Logic:** With hooks, we can extract and share logic across components without changing the component structure.
3. **Simpler Code:** Hooks reduce the need for complex lifecycle methods, especially in class components, and allow us to handle component updates more effectively.

2.5.1 useState: Managing State

The `useState` hook allows us to add state to a functional component.

```
import React, { useState } from 'react';
```

```
function Counter() {
  // Declare a state variable "count" initialized to 0
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increase</button>
    </div>
  );
}
```

In this example `useState(0)` initializes `count` with a value of 0 and `setCount` is the function we use to update count.

2.5.2 `useEffect`: Managing Side Effects

The `useEffect` hook allows us to perform side effects like data fetching, updating the DOM, or setting up subscriptions (see also the section 2.4.5).

2.5.3 `useContext`: Managing Context

The `useContext` hook allows us to access context directly in a functional component without needing to use a context consumer.

```
import React, { useContext } from 'react';
// Assuming ThemeContext is created in another file
const ThemeContext = React.createContext('light');

function DisplayTheme() {
  const theme = useContext(ThemeContext); // Accesses context value

  return <p>Current theme: {theme}</p>;
}
```

In this example `useContext(ThemeContext)` gets the current value of `ThemeContext`, which can be shared among components without prop drilling.

2.5.4 `useReducer`: Managing Complex State Logic

The `useReducer` hook is an alternative to `useState` for managing more complex state logic, especially when the state updates depend on previous states or involve multiple sub-values. It's similar to the `reducer` function pattern in Redux.

```
const [state, dispatch] = useReducer(reducerFunction, initialState);
```

Here, `reducerFunction` is a function that takes in the current state and an action, and returns the updated state, `initialState` is the initial state value and `dispatch` is a function that triggers state changes based on actions.

Let's use `useReducer` to manage the state of a counter with different actions for increment, decrement, and reset.

```
import React, { useReducer } from 'react';

const initialState = { count: 0 };

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    case 'reset':
      return initialState;
    default:
      throw new Error(`Unknown action: ${action.type}`);
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
```

```

<div>
  <p>Count: {state.count}</p>
  <button onClick={() => dispatch({ type: 'increment' })}>+</button>
  <button onClick={() => dispatch({ type: 'decrement' })}>-</button>
  <button onClick={() => dispatch({ type: 'reset' })}>Reset</button>
</div>

);
}

```

In this example *reducer* handles the logic for each action type and *dispatch* triggers updates to count by dispatching action objects with a specific type.

2.5.5 `useCallback`: Memoizing Functions

The `useCallback` hook is used to memoize functions, preventing them from being re-created on every render. This is particularly helpful when passing functions to child components that may otherwise cause unnecessary re-renders. The basic syntax is like this:

```

const memoizedCallback = useCallback(() => {
  // Function logic here
}, [dependencies]);

```

Here, `dependencies` is an array of variables that trigger the function to re-create when they change.

Consider a component that renders a list of items. We'll use `useCallback` to memoize a function that handles the selection of an item in the list.

```

import React, { useState, useCallback } from 'react';

function ItemList({ items, onItemSelect }) {
  return (
    <ul>
      {items.map((item) => (
        <li key={item.id} onClick={() => onItemSelect(item.id)}>
          {item.name}
        </li>
      ))}
    </ul>
  );
}

```

```

    ))}
  </ul>
);
}

function App() {
  const [selectedItemId, setSelectedItemId] = useState(null);

  const handleItemSelect = useCallback((id) => {
    setSelectedItemId(id);
  }, []);

  const items = [{ id: 1, name: 'Item 1' }, { id: 2, name: 'Item 2' }];

  return (
    <div>
      <ItemList items={items} onItemSelect={handleItemSelect} />
      <p>Selected Item ID: {selectedItemId}</p>
    </div>
  );
}

```

In this example *handleItemSelect* is memoized with `useCallback` and only re-created if its dependencies (none in this case) change. This prevents unnecessary re-renders of `ItemList` if `App` re-renders for reasons unrelated to `handleItemSelect`.

2.5.6 `useMemo`: Memoizing Values

The `useMemo` hook is used to memoize the result of a computation, preventing it from being re-calculated on every render unless its dependencies change. `useMemo` is beneficial for expensive calculations or operations.

The basic syntax is the following:

```

const memoizedValue = useMemo(() =>
  computeExpensiveValue(someDependency), [someDependency]);

```

Here, *computeExpensiveValue* is the function containing the logic for the calculation and *someDependency* is the variables that trigger re-calculation when they change.

Let's look at an example where we compute an expensive value based on the count. We'll use `useMemo` to ensure that this computation only happens when necessary.

```
import React, { useState, useMemo } from 'react';

function App() {
  const [count, setCount] = useState(0);
  const [show, setShow] = useState(true);

  const expensiveCalculation = (count) => {
    console.log('Calculating...');
    return count * 2; // Example of a computationally intensive calculation
  };

  const memoizedValue = useMemo(() => expensiveCalculation(count), [count]);

  return (
    <div>
      <p>Count: {count}</p>
      <p>Result: {memoizedValue}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
      <button onClick={() => setShow(!show)}>Toggle Show</button>
    </div>
  );
}
```

In this example *expensiveCalculation* runs only when count changes. *useMemo* stores the result of *expensiveCalculation*, avoiding unnecessary recalculations if count remains the same.

2.5.7 Custom Hooks: Reusable Logic

Custom hooks let us create and reuse custom logic in multiple components. A custom hook is just a JavaScript function whose name starts with "use."

```
import { useState, useEffect } from 'react';

function useWindowWidth() {
  const [width, setWidth] = useState(window.innerWidth);

  useEffect(() => {
    const handleResize = () => setWidth(window.innerWidth);
    window.addEventListener('resize', handleResize);

    return () => window.removeEventListener('resize', handleResize);
  }, []);

  return width;
}

// Example usage of custom hook
function WindowSizeDisplay() {
  const width = useWindowWidth();

  return <p>Window width: {width}px</p>;
}
```

Here *useWindowWidth* is a custom hook that tracks the window's width. By calling *useWindowWidth* in any component, we can easily reuse this logic.

2.6 State Management

State management is one of the most crucial aspects of React applications. It determines how data flows through an application and how components

communicate with each other. As applications grow in size and complexity, choosing the right state management solution becomes increasingly important.

We can think of state as a **storage** for dynamic data. Every time state changes, React determines which parts of the UI need to update to reflect that change. So, managing state effectively is about organizing, accessing, and responding to these changes across an app.

There are two core actions tied to state:

1. **Storing Data:** State is a storage mechanism for any dynamic data.
2. **Reacting to Changes:** When state changes, components that rely on it re-render to display the new data.

2.6.1 Types of State Management Libraries

The complexity of an app and the types of data it handles play a big role in determining the best state management approach. Two main categories of state management libraries have emerged to meet these needs: **centralized** and **atomized** state management.

2.6.1.1 Centralized State Management

In centralized state management, **all state is stored in a single global store**. This approach works well for apps where many components need to access and interact with shared data. Instead of having individual states spread across many components, centralized management consolidates state into a single source of truth, making data easier to organize and update consistently.

Common Examples of Centralized State Management Libraries are Redux [6] and MobX. **Redux** is one of the most widely used libraries, it provides a single store where state is managed through actions and reducers. Another library is **MobX** which commonly used to create centralized stores where multiple components can observe and react to state changes (Figure 2.5).

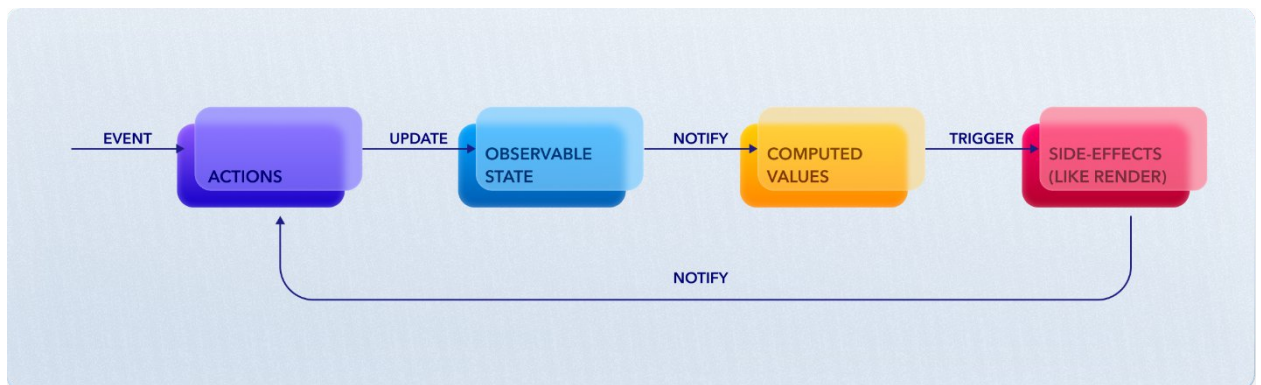


Figure 2.5. [*Centralized State Management using MobX*](#)

Advantages of Centralized State Management are the following.

1. **Consistency:** All components refer to one place for state, reducing redundancy and ensuring data consistency.
2. **Debugging and Predictability:** With centralized actions and reducers, it's easier to track what changed and why.
3. **Suitability for Larger Apps:** As applications grow, centralized state management scales well because data is centrally organized.

While centralized state management offers many benefits, it also presents some drawbacks, such as:

1. **Setup Complexity:** Setting up a centralized state management system (e.g., Redux) can introduce additional boilerplate and concepts, such as actions, reducers, and middlewares.
2. **Re-render Challenges:** When a state change happens in the store, it can trigger re-renders for all components that rely on that state, even if the change only impacts one component. This can require optimizations to avoid performance issues.

2.6.1.2 Atomized State Management

In atomized (or decentralized) state management, **state is broken into smaller, isolated pieces (atoms)** that each component can independently observe and modify. This approach is more flexible, allowing components to track only the specific pieces of state they need, rather than the entire global state.

Atomized state management works well for apps that don't require a single global store and where many components rely on independent or granular pieces of state. This approach is reactive, meaning each atom can independently update without affecting unrelated parts of the app.

Common Examples of Atomized State Management Libraries are **Recoil** which is built around the concept of "atoms" (individual state units) and selectors, allowing each atom to be reactive and scoped to the components that depend on it (Figure

2.6), **Zustand** which provides a minimalist, flexible approach to managing state, where each piece of state can be independently accessed and modified without a centralized structure and **Jotai** which enables us to build state as a collection of atoms, making it easy to manage and organize by specific needs.

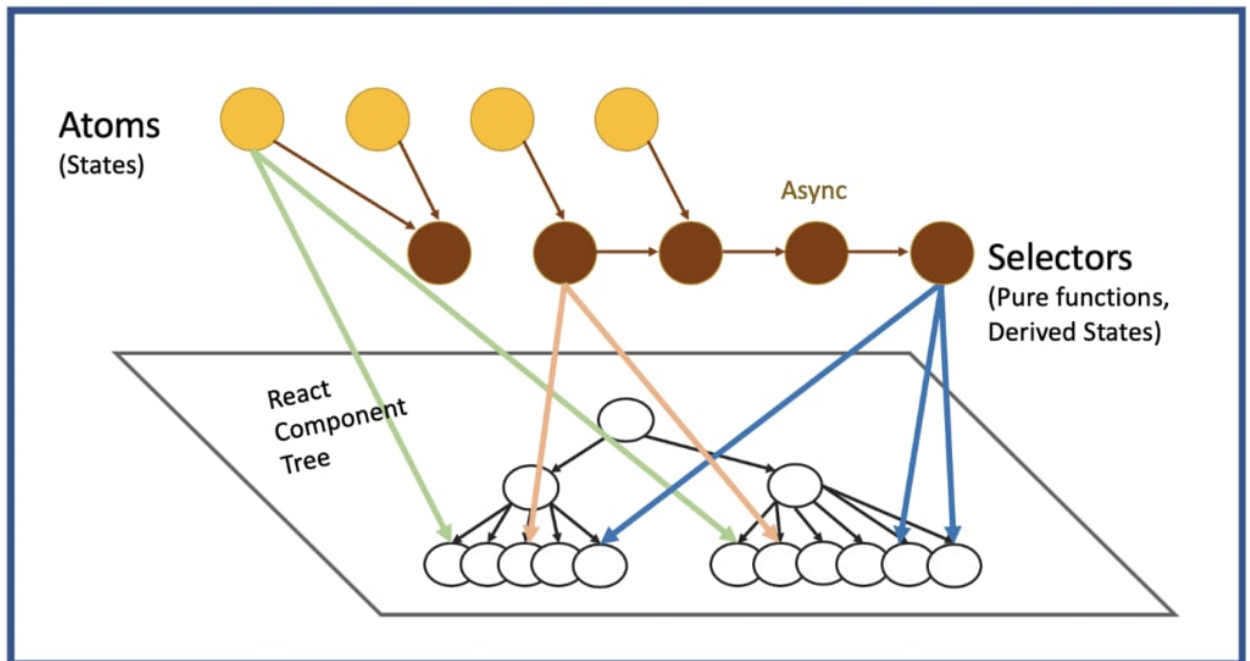


Figure 2.6. Atomized State Management using Recoil

Advantages of Atomized State Management

1. **Localized State Access:** Components subscribe only to the specific atoms they need, reducing unnecessary re-renders.
2. **Flexibility:** Each component or group of components can create and manage its own atoms without relying on a single centralized store.
3. **Efficient for Fine-Grained State:** Atomized state is efficient for complex applications where certain components need highly specific data that doesn't need to be shared widely across the app.

Limitations of Atomized State Management

1. **Potential for Fragmentation:** As atoms are created for each specific need, managing and coordinating them can become challenging in large apps.
2. **Scalability in Complex Apps:** Atomized state management works well for medium-sized applications, but in large applications, the high degree of state fragmentation can make it hard to maintain and debug.

2.6.2 Types of State

In general, UI state controls the visual elements and interactions within the application. It's often temporary, changing based on user actions, and doesn't need

to be saved once a user leaves the page or reloads it. These are the main types of state—*UI*, *server cache*, *client*, and *URL*.

UI State primarily handles the temporary, interactive aspects of the app’s interface. This type of state is used for elements like form inputs, toggle switches, loading indicators, and modals. For example, form fields that capture a user’s name and email are managed within UI state. When the user enters text, this data is stored temporarily, clearing when the form is submitted or the page is refreshed.

UI state is typically managed with `useState` or `useReducer` within components.

Server Cache State refers to data fetched from a server, such as API responses or results from asynchronous operations, which may need periodic updating or synchronization with the server. This state is essential for ensuring that users view the most current data without needing constant refetching. For instance, if you build a blog that retrieves posts from an API, those posts are stored in server cache state, allowing users to view them without immediately fetching them again. If the server updates, this data can be revalidated to reflect changes.

Libraries such as *React Query* and *SWR* are excellent tools for handling server cache state, providing efficient caching and refetching mechanisms.

Client State encompasses data that remains on the client side, impacting the user’s experience but not requiring synchronization with the server. This state type includes user preferences, authentication status, and theme settings. Consider a language setting in an application; if a user selects French as their preferred language, this preference is stored in client state, allowing the app to load in French every time without repeatedly asking for it. Authentication status, which identifies whether a user is logged in or logged out, also resides in client state. By maintaining authentication as client state, the app controls what content is visible and which areas are accessible based on the user’s session. Theme settings are another instance of client state—when a user switches to dark mode, this preference is saved in the client state, ensuring the theme remains consistent across the app.

Managing client state efficiently can become complex as applications grow, and tools like *Redux* and *Recoil* offer robust solutions for sharing this state across multiple components.

URL State is used to represent data embedded within the URL, enabling shareable, bookmarkable links and easier navigation. This includes route parameters, query parameters, and navigation state. In a product catalog, for instance, the URL might look like `/product/123`, with 123 serving as a route parameter indicating which product to display. Query parameters are also part of URL state and are especially useful for searches and filters; `/search?query=shoes&color=black` directly encodes a user’s search preferences, allowing them to share or revisit the search results.

Similarly, URL state can represent navigation state, like specifying the active tab in a user profile with a URL like `/profile?tab=settings`.

React Router simplifies URL state management, and frameworks like *Next.js* offer integrated support for handling routing and query parameters efficiently in server-rendered applications.

2.6.3 State Management Solutions

2.6.3.1 Local State (*useState*)

Best for: Simple, component-level state management.

The `useState` hook is perfect for managing local state **within individual components**. It's ideal for storing simple data such as form input values, toggles, and counters. `useState` is lightweight and allows quick access to data that doesn't need to be shared across multiple components. The example:

```
function Counter() {  
  const [count, setCount] = useState(0);  
  return (  
    <button onClick={() => setCount(count + 1)}>  
      Count: {count}  
    </button>  
  );  
}
```

2.6.3.2 Local Reducer State (*useReducer*)

Best for: More complex state logic within a component.

When state management within a component involves complex updates or multiple actions, `useReducer` provides a more structured approach than `useState`. It is ideal for managing state where multiple, interconnected variables need updating, such as form fields with validation. `useReducer` is also helpful for state that could grow or change frequently within a single component.

Here's an example of using `useReducer` for a simple form with a name, email and password fields:

```
const initialState = {  
  name: "",
```

```

email: "",
password: "",
errors: { name: "", email: "", password: "" }
};

function formReducer(state, action) {
  switch (action.type) {
    case 'SET_FIELD':
      return {
        ...state,
        [action.field]: action.value,
        errors: { ...state.errors, [action.field]: "" }
      };
    case 'SET_ERROR':
      return {
        ...state,
        errors: { ...state.errors, [action.field]: action.error }
      };
    case 'RESET':
      return initialState;
    default:
      return state;
  }
}

const [state, dispatch] = useReducer(formReducer, initialState);

const handleInputChange = (e) => {
  dispatch({ type: 'SET_FIELD', field: e.target.name, value: e.target.value });
}

```

```

};

const handleSubmit = () => {
  if (!state.name) dispatch({ type: 'SET_ERROR', field: 'name', error: 'Name is required' });
  if (!state.email.includes('@')) dispatch({ type: 'SET_ERROR', field: 'email', error: 'Invalid email' });
  if (state.password.length < 6) dispatch({ type: 'SET_ERROR', field: 'password', error: 'Too short' });
};

```

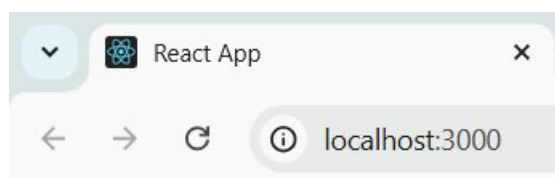
In this example we have two form's event handlers: *handleInputChange(e)* and *handleSubmit()*. The first one is triggered once any of fields is edited and the second works when the form is submitted. The *handleInputChange* just dispatches 'set value' event in order to update the local state. In other words in send a command to *formReducer()* to perform this action. On the other hand, the *handleSubmit()* tells the *formReducer()* to check all fields for errors.

2.6.3.3 Context API

Best for: Sharing state across deeply nested components.

The Context API is good solution for passing state down multiple levels of the component tree without prop drilling. It works well for global data that changes infrequently, such as themes, user authentication status, or language preferences. However, since it can lead to performance issues if the data changes frequently, it's often best combined with *useReducer* to manage state updates more efficiently.

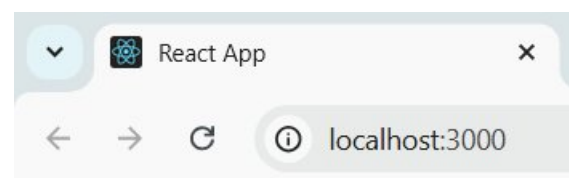
Let's consider an example of creating a component for language selection and another component to display text based on the selected language (Figure 2.7 and Figure 2.8).



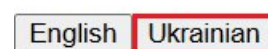
Current Language: English



Hello, welcome!



Current Language: Ukrainian



Вітаю, привіт!

Figure 2.7. The App UI for English Language *Figure 2.8. The App UI for Ukrainian Language*

App.js acts as the root of the application, where all actions begin. It wraps all child components within a central provider called `LanguageProvider`, ensuring every part of the app can access the language preferences. It renders two key components: **LanguageSelector** and **LocalizedText**.

```
import React from 'react';
import { LanguageProvider } from './LanguageContext.js';
import LanguageSelector from './LanguageSelector.js';
import LocalizedText from './LocalizedText.js';

function App() {
  return (
    <LanguageProvider>
      <LanguageSelector />
      <LocalizedText />
    </LanguageProvider>
  );
}

export default App;
```

LanguageProvider (defined in `LanguageContext.js`) plays a crucial role by creating a React Context to hold the language state. This component serves as a central store, managing the language setting and making both the current language and the function to change it available throughout the entire app. Child components don't need to manage their own language state — they simply rely on this global context. The code looks as the following.

```
import React, { createContext, useReducer, useContext } from 'react';
// Define available languages and actions
const initialState = { language: 'en' }; // default to English
const LanguageContext = createContext();
const languageReducer = (state, action) => {
```

```

switch (action.type) {
  case 'SET_LANGUAGE':
    return { ...state, language: action.payload };
  default:
    return state;
}
};

// Create a provider component
export const LanguageProvider = ({ children }) => {
  const [state, dispatch] = useReducer(languageReducer, initialState);

  const setLanguage = (language) => {
    dispatch({ type: 'SET_LANGUAGE', payload: language });
  };

  return (
    <LanguageContext.Provider value={{ language: state.language, setLanguage }}>
      {children}
    </LanguageContext.Provider>
  );
};

// Custom hook to use Language context
export const useLanguage = () => useContext(LanguageContext);

```

In the code example above, we create a language context (LanguageContext) and a language reducer (languageReducer). The reducer handles the SET_LANGUAGE action, enabling organized state updates. The LanguageProvider is a special component for state management and logic encapsulation, separating the concern. It acts as a bridge between client components and the specific context.

In **LanguageSelector.js**, the current language is accessed through the useLanguage hook, which taps into the Context. This component provides a user interface for changing the language. When a user selects a new language, the LanguageProvider state updates, automatically propagating the change to any component that relies on this value.

```

import React from 'react';
import { useLanguage } from './LanguageContext';

const LanguageSelector = () => {
  const { language, setLanguage } = useLanguage();

  return (
    <div>
      <p>Current Language: {language === 'en' ? 'English' : 'Ukrainian'}</p>
      <button onClick={() => setLanguage('en')}>English</button>
      <button onClick={() => setLanguage('ua')}>Ukrainian</button>
    </div>
  );
};

export default LanguageSelector;

```

LocalizedText.js also connects to the `LanguageProvider` using the `useLanguage` hook. It consumes the current language value from the `Context` to determine what text should be displayed. Additionally, it shows the corresponding country flag using the language code, giving users a visual cue of the selected language.

```

import React from 'react';
import { useLanguage } from './LanguageContext.js';

const translations = {
  en: {
    greeting: "Hello, welcome!",
  },
  ua: {
    greeting: "Вітаю, привіт!",
  },
}

```

```

};

const LocalizedText = () => {
  const { language } = useLanguage();
  const languageFlagId = (language === 'en') ? 'GB' : language.toUpperCase();
  return <div style={{display:'flex', alignItems:'center', gap:'10px'}}>
    <img src={`https://flagsapi.com/${languageFlagId}/flat/64.png`} alt='flag'/>
    {translations[language].greeting}</div>;
};

export default LocalizedText;

```

The core of this application is the **shared state** managed by `LanguageProvider`. By lifting the state up and using React Context, `LanguageSelector` and `LocalizedText` can access the language data without needing to communicate directly. This pattern optimizes state management across the app, ensuring consistent updates and a responsive user experience.

2.6.3.4 Redux

When building large-scale React applications, managing state across multiple components can quickly become challenging. As the application grows, so does the complexity of state management, making it harder to track data flow and ensure consistent updates. This is where **Redux** comes into play. Redux offers a robust solution for managing global state, helping you organize data flow and ensure predictable updates throughout your application.

Redux [7] is a state management library for JavaScript applications, commonly used with React but not limited to it. It follows a **unidirectional data flow** and centralizes an application state in a single, global store. This approach helps maintain consistency, making it easier to reason about how data is shared and updated across an application.

Redux is built around three fundamental principles:

1. **Single Source of Truth:** The entire state of your application is stored in a single JavaScript object called the "store." This central store provides a single point of access for state, making it easier to manage and debug.
2. **State is Read-Only:** The state can only be modified by dispatching actions. Instead of changing the state directly, components describe what happened

by dispatching an action. This helps prevent unpredictable changes and keeps state updates predictable.

3. **Changes are Made with Pure Functions:** State updates in Redux are handled by **reducers**, which are pure functions that take the current state and an action as arguments and return a new state. Reducers ensure that state changes are predictable and do not produce side effects.

To understand how Redux works, let's break down its key concepts:

Store: The store is the central hub of Redux, holding the entire state of your application. It provides methods to get the current state, dispatch actions, and subscribe to state changes.

```
import { createStore } from 'redux';  
const store = createStore(reducer);
```

Action: An action is a plain JavaScript object that describes a change in the application. It has a `type` property (a string that defines the action) and an optional `payload` containing any additional data.

```
const addItemAction = {  
  type: 'ADD_ITEM',  
  payload: { item: 'Apple' }  
};
```

Reducer: A reducer is a pure function that takes the current state and an action as arguments, processes the action, and returns a new state. It determines how the state should change based on the action type.

```
function cartReducer(state = [], action) {  
  switch (action.type) {  
    case 'ADD_ITEM':  
      return [...state, action.payload.item];  
    default:  
      return state;  
  }  
}
```

Dispatch: The `dispatch` method is used to send actions to the store, triggering state updates.

```
store.dispatch(addItemAction);
```

Selectors: Selectors are functions that extract specific pieces of state from the store. They help keep your component logic clean and focus on reading data.

```
const selectCartItems = (state) => state.cart.items;
```

```
const items = selectCartItems(store.getState());
```

Redux follows a unidirectional data flow (Figure 2.9):

1. A **component** triggers an action (e.g., `ADD_ITEM`).
2. The **action** is dispatched to the Redux store using `dispatch()`.
3. The **reducer** function processes the action and returns a new state.
4. The Redux **store** updates its state based on the new data.
5. The **component** re-renders with the updated state.

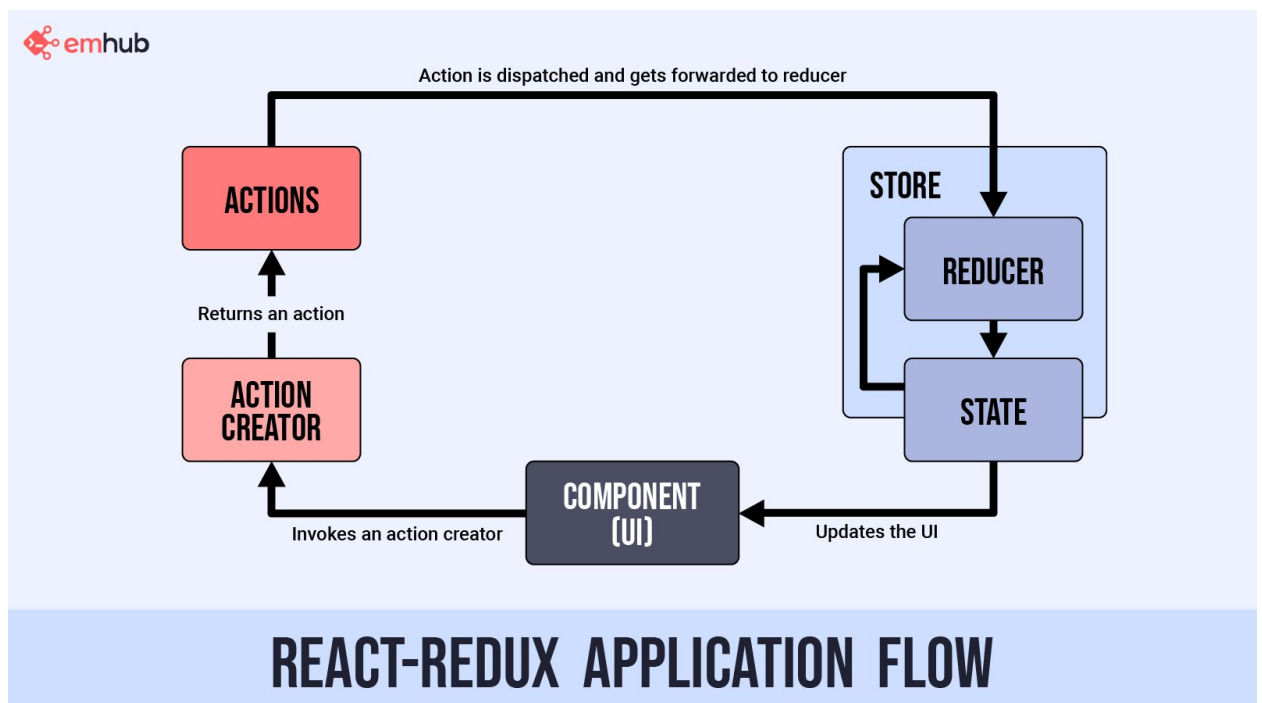


Figure 2.9. [Unidirectional Data Flow in React](#)

Redux DevTools

One of the remarkable features of Redux is its integration with **Redux DevTools**, a browser extension that allows us to inspect every action and state change in your application. With DevTools, we can **rewind and replay state** changes to see exactly how your app got to its current state, **inspect every action dispatched** in an app as well as **examine the state** before and after each action. This level of visibility is especially useful in large applications, where tracking down bugs can be difficult.

2.6.3.5 Redux Toolkit

Redux is powerful, but the boilerplate code (creating actions, reducers, and store configuration) can be overwhelming, especially for beginners. Redux Toolkit is now the recommended way to use Redux, and it addresses some of the complexity and boilerplate code that made "plain" Redux harder to learn and more cumbersome to set up. It simplifies the Redux setup with built-in functions and conventions, making it easier for students to understand and apply state management in React apps. Redux Toolkit simplifies this process by providing convenient helper functions and sensible defaults like **createSlice** which automatically generates action creators and a reducer and **configureStore** which sets up the store with good defaults, like middleware for handling async logic.

Each slice includes the **initial state** for that feature, a set of **reducer functions** that define how the state changes in response to dispatched actions, the **actions** themselves, which are automatically generated for each reducer function and the **slice reducer**, which is exported for combining in the Redux store.

Let's consider a simple counter example. It allows a user to increment and decrement a value as well as increment the value by a constant (Figure 2.10).

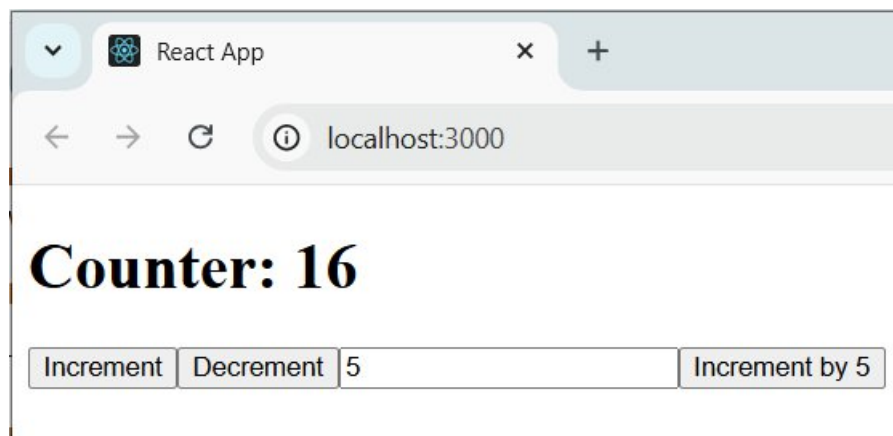


Figure 2.10. Counter Application UI using React Toolkit

To set up the React project with Redux Toolkit, follow these steps.

1. Create a New React Project:

```
npx create-react-app redux-toolkit-counter  
cd redux-toolkit-counter
```

2. Install Redux Toolkit and React-Redux:

```
npm install @reduxjs/toolkit react-redux
```

3. Create the Redux Slice: a "slice" in Redux Toolkit represents a specific part of the state and the logic to manage it.

Create a file named `counterSlice.js` in a `features/counter/` folder structure. Organizing files by feature makes it easier to manage as the app grows.

```
import { createSlice } from '@reduxjs/toolkit';

// Step 1: Define the initial state for the counter
const initialState = {
  value: 0,
};

// Step 2: Create the slice, defining state and reducer logic
const counterSlice = createSlice({
  name: 'counter', // a unique name for the slice
  initialState, // the initial state
  reducers: {
    increment: (state) => {
      state.value += 1; // direct mutation, handled internally by Redux Toolkit
    },
    decrement: (state) => {
      state.value -= 1;
    },
    incrementByAmount: (state, action) => {
      state.value += action.payload; // action.payload allows passing a custom value
    },
  },
});

// Export actions generated by createSlice
export const { increment, decrement, incrementByAmount } = counterSlice.actions;

// Export the reducer to configure the store
```

```
export default counterSlice.reducer;
```

4. **Set Up the Redux Store:** The store combines the slices and provides the entire state tree for the app. **Create a `store.js` file** in a new `app` folder.

```
// src/app/store.js

import { configureStore } from '@reduxjs/toolkit';
import counterReducer from '../features/counter/counterSlice';

// Step 1: Configure the store
const store = configureStore({
  reducer: {
    counter: counterReducer, // add the counter slice reducer here
  },
});

export default store;
```

5. **Provide the Redux Store to the React App:** Use the `Provider` component from `react-redux` to make the store available to all components. **Wrap the root of the app** in `index.js`.

```
// src/index.js

import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import App from './App';
import store from './app/store';

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
```

```
</Provider>,  
document.getElementById('root')  
);
```

Create a counter component that displays the current count and provides buttons to increase, decrease, or increment the count by a specified amount:

```
// src/features/counter/Counter.js  
  
import React, { useState } from 'react';  
import { useSelector, useDispatch } from 'react-redux';  
import { increment, decrement, incrementByAmount } from './counterSlice';  
  
const Counter = () => {  
  const count = useSelector((state) => state.counter.value); // Selects counter value  
  from the store  
  const dispatch = useDispatch(); // Dispatches actions  
  
  const [incrementAmount, setIncrementAmount] = useState(2); // Local state for  
  input  
  
  return (  
    <div>  
      <h1>Counter: {count}</h1>  
      <button onClick={() => dispatch(increment())}>Increment</button>  
      <button onClick={() => dispatch(decrement())}>Decrement</button>  
      <input  
        type="number"  
        value={incrementAmount}  
        onChange={(e) => setIncrementAmount(Number(e.target.value))}  
      />  
      <button onClick={() => dispatch(incrementByAmount(incrementAmount))}>
```

```

    Increment by {incrementAmount}
  </button>
</div>
);
};

export default Counter;

```

6. Use the Counter Component in App.js

```

// src/App.js

import React from 'react';
import Counter from '../features/counter/Counter';

function App() {
  return (
    <div className="App">
      <Counter />
    </div>
  );
}

export default App;

```

2.6.3.6 REST API with Redux Toolkit

Redux Toolkit simplifies managing REST API calls with its built-in `createAsyncThunk` function and `createSlice` utility, which together provide a streamlined way to handle async logic, caching, and loading states. The `createAsyncThunk` generates a function for fetching data from a REST API, handling the promise lifecycle automatically (e.g., loading, success, error) whereas the `createSlice` defines the slice's initial state and reducers. Inside the slice, we can add extra reducers to handle the async actions generated by the thunk.

Here's a simple example of setting up Redux Toolkit to fetch and manage posts from a REST API. We use well-known service [JSON Placeholder](#) to get list of blog posts with “Get More” button allowing us to achieve “infinite loop” of items ()

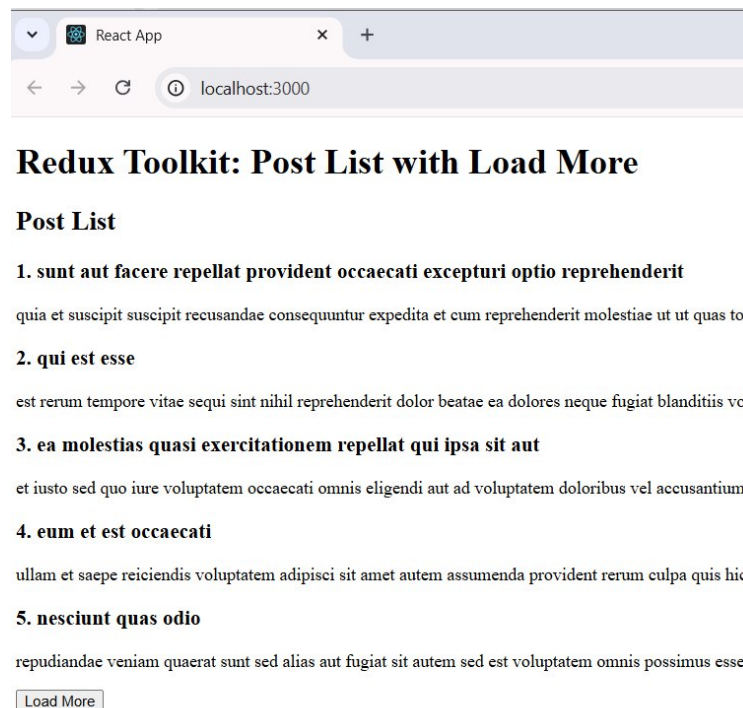


Figure 2.11. Redux Toolkit REST API Example

We need to implement 3 files: *postsSlice.js* which uses *createAsyncThunk*, *store.js* which defines a storage and *PostList.js* which is the REST API consuming component.

```
// postsSlice.js
import { createSlice, createAsyncThunk } from "@reduxjs/toolkit";
import axios from "axios";

export const fetchPosts = createAsyncThunk("posts/fetchPosts", async (page) => {
  const response = await axios.get(
    `https://jsonplaceholder.typicode.com/posts?_limit=10&_page=${page}`
  );
  return response.data;
});

const postsSlice = createSlice({
  name: "posts",
  initialState: {
```

```

    posts: [],
    status: "idle",
    error: null,
    page: 1,
  },
  reducers: {
    incrementPage(state) {
      state.page += 1;
    },
  },
  extraReducers: (builder) => {
    builder
      .addCase(fetchPosts.pending, (state) => {
        state.status = "loading";
      })
      .addCase(fetchPosts.fulfilled, (state, action) => {
        state.status = "succeeded";
        state.posts = [...state.posts, ...action.payload];
      })
      .addCase(fetchPosts.rejected, (state, action) => {
        state.status = "failed";
        state.error = action.error.message;
      });
  },
});

export const { incrementPage } = postsSlice.actions;
export default postsSlice.reducer;

```

In this code we set up a Redux slice for managing posts, including fetching them from an API. It handles the loading state, success, and error states using Redux Toolkit's features. The **extraReducers** allow us to handle actions defined outside of the slice (like the async thunk). It uses a builder pattern to add cases for the different states of the fetchPosts thunk.

The next step is to configure the Store, we need to add the slice's reducer to the Redux store so it can manage the posts state globally:

```
import { configureStore } from "@reduxjs/toolkit";
import postsReducer from "./postsSlice";

const store = configureStore({
  reducer: {
    posts: postsReducer,
  },
});

export default store;
```

The following code shows how to use Thunk in components. We need to **dispatch the Thunk** to start fetching data and apply **useSelector** to retrieve the data and loading state from the Redux store:

```
import React, { useEffect } from "react";
import { useSelector, useDispatch } from "react-redux";
import { fetchPosts, incrementPage } from "./postsSlice";

const PostList = () => {
  const dispatch = useDispatch();
  const posts = useSelector((state) => state.posts.posts);
  const status = useSelector((state) => state.posts.status);
  const error = useSelector((state) => state.posts.error);
  const page = useSelector((state) => state.posts.page);
```

```

useEffect(() => {
  if (status === "idle") {
    dispatch(fetchPosts(page));
  }
}, [dispatch, status, page]);

const loadMorePosts = () => {
  dispatch(incrementPage());
  dispatch(fetchPosts(page + 1));
};

if (status === "loading") {
  return <p>Loading...</p>;
}

if (status === "failed") {
  return <p>Error: {error}</p>;
}

return (
  <div>
    <h2>Post List</h2>
    {posts.map((post) => (
      <div key={post.id}>
        <h3>{post.id}. {post.title}</h3>
        <p>{post.body}</p>
      </div>
    ))}
    <button onClick={loadMorePosts}>Load More</button>
  </div>
)

```

```
</div>

);
};

export default PostList;
```

2.7 React Router

React Router is a standard library for routing in React applications. Routing enables us to navigate between different views in your application and is key to creating a seamless single-page experience for users.

To add React Router to your project, start by installing the `react-router-dom` package:

```
npm install react-router-dom
```

Once installed, import the necessary components from `react-router-dom` in your React application.

React Router provides a set of components to define and control routes. Here's an overview of the main ones:

- **BrowserRouter:** This component wraps your entire app and provides the routing context.
- **Routes:** A container for all route definitions.
- **Route:** Defines the path and component that should render when the path matches the current URL.
- **Link:** A component used for navigation without reloading the page.
- **Navigate:** Redirects users to a different route programmatically.

A basic React Router setup includes a `BrowserRouter`, `Routes`, and multiple `Route` components. Here is an example:

```
// App.js

import React from "react";
import { BrowserRouter, Routes, Route } from "react-router-dom";
import Home from "../pages/Home";
import About from "../pages/About";
import Contact from "../pages/Contact";

function App() {
```

```

return (
  <BrowserRouter>
    <Routes>
      <Route path="/" element={<Home />} />
      <Route path="/about" element={<About />} />
      <Route path="/contact" element={<Contact />} />
    </Routes>
  </BrowserRouter>
);
}
export default App;

```

In this example `Home`, `About`, and `Contact` are components that render when the URL matches the specified path and the root path `/` loads the `Home` component, `/about` loads the `About` component, and `/contact` loads the `Contact` component.

Using `<Link>` instead of `<a>` prevents page reloads and keeps the app as a single-page application (SPA).

```

// Navbar.js
import React from "react";
import { Link } from "react-router-dom";

function Navbar() {
  return (
    <nav>
      <Link to="/">Home</Link>
      <Link to="/about">About</Link>
      <Link to="/contact">Contact</Link>
    </nav>
  );
}
export default Navbar;

```

Here, `to` specifies the destination path, and clicking these links will navigate between routes without reloading the page.

React Router supports nested routes, allowing for hierarchical organization of an app's structure.

```
// App.js
import React from "react";
import { BrowserRouter, Routes, Route } from "react-router-dom";
import Layout from "../components/Layout";
import Home from "../pages/Home";
import About from "../pages/About";
import Team from "../pages/Team";

function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Layout />}>
          <Route index element={<Home />} />
          <Route path="about" element={<About />} />
          <Route path="about/team" element={<Team />} />
        </Route>
      </Routes>
    </BrowserRouter>
  );
}

export default App;
```

In this example nested routes like `about/team` render the `Team` component inside the `Layout`.

The `Navigate` component is useful for programmatically redirecting users.

```
import React from "react";
```

```
import { Navigate } from "react-router-dom";

function ProtectedRoute({ isAuthenticated }) {
  return isAuthenticated ? <Dashboard /> : <Navigate to="/login" />;
}
```

In this example if the `isAuthenticated` prop is `false`, the user is redirected to `/login`, otherwise the `Dashboard` component is rendered.

URL parameters allow capturing dynamic segments in the URL, which is great for routes like `/product/:id`.

```
// Product.js
import React from "react";
import { useParams } from "react-router-dom";

function Product() {
  const { id } = useParams();
  return <div>Product ID: {id}</div>;
}

// App.js
<Route path="/product/:id" element={<Product />} />
```

In this setup navigating to `/product/5` would render the `Product` component with `id` equal to 5.

To use query parameters, use the `useLocation` hook to access the current URL and parse query strings.

```
// Search.js
import React from "react";
import { useLocation } from "react-router-dom";

function useQuery() {
  return new URLSearchParams(useLocation().search);
}
```

```

}

function Search() {
  const query = useQuery();
  const searchTerm = query.get("q");

  return <div>Search Term: {searchTerm}</div>;
}

```

// App.js

```
<Route path="/search" element={<Search />} />
```

Accessing `/search?q=react` will render Search Term: react.

A `*` path can be used to catch any route that doesn't match.

// NotFound.js

```
import React from "react";
```

```

function NotFound() {
  return <h2>Page Not Found</h2>;
}

```

// App.js

```
<Route path="*" element={<NotFound />} />
```

If the user navigates to a non-existent route, the `NotFound` component is displayed.

Sometimes we want to redirect the user based on certain conditions using `Navigate` within components.

```
import React from "react";
```

```
import { useNavigate } from "react-router-dom";
```

```
function Login({ isLoggedIn }) {
```

```

const navigate = useNavigate();

const handleLogin = () => {
  // Log in logic
  if (isLoggedIn) {
    navigate("/dashboard");
  }
};

return <button onClick={handleLogin}>Log in</button>;
}

```

In this example useNavigate is used to redirect to /dashboard if the user is logged in.

To protect routes for authenticated users, create a wrapper component.

```

import React from "react";
import { Navigate } from "react-router-dom";

function ProtectedRoute({ children, isAuthenticated }) {
  if (!isAuthenticated) {
    return <Navigate to="/login" />;
  }
  return children;
}

// App.js
<Route
  path="/dashboard"
  element={
    <ProtectedRoute isAuthenticated={isLoggedIn}>
      <Dashboard />
    </ProtectedRoute>
  }
 />

```

```
}  
/>
```

Only authenticated users can access the dashboard; unauthenticated users are redirected to `/login`.

3 Mobile UI Development using Android

Mobile app development took off in the late 2000s with the release of the iPhone in 2007 and Android in 2008, which introduced smartphones capable of running complex, user-friendly applications. Unlike web development, which builds apps accessible through browsers, mobile development focuses on creating apps tailored to specific devices, using features like touch input, sensors, and push notifications to deliver richer, more interactive experiences. This shift opened a new era of development, expanding possibilities beyond the browser [8].

When designing user interfaces, it's important to recognize the differences between web and mobile platforms. Both serve unique contexts and user needs, which influence how their UIs are structured and function. While web UIs prioritize flexibility and adaptability across devices, mobile UIs emphasize efficiency, simplicity, and device integration. Let's explore the core distinctions with a focus on mobile features.

Limited Energy and Resource Constraints

Mobile devices operate on battery power and have limited processing capabilities compared to desktops. This means mobile UIs must be designed with energy efficiency in mind. Animations, background tasks, and network usage must be optimized to prevent excessive battery drain or performance issues. For example, apps using background services (e.g., syncing data) need to be carefully managed to avoid unnecessary power consumption.

Heavy App Interactions

Unlike web apps that often operate in isolation, mobile apps frequently interact with other applications and system features. For instance, apps can share data (e.g., sharing a poll result via messaging apps) or notifications from one app can trigger actions in another (e.g., a calendar app opening directly to a scheduled meeting).

These interactions are enabled by features like **intents** and **content providers**, which allow mobile apps to communicate seamlessly. For example, in a poll app, we can share results via email or social media directly from the app.

One Window per Screen

Mobile UIs are constrained to a single window (or screen) at a time, unlike web interfaces that can leverage multiple tabs or split-screen experiences. This single-screen focus requires a **linear navigation structure** and clear, concise UI elements. Designing for one screen at a time, we need to prefer minimalism, as messy screens can overwhelm users. For that reason, it requires creation of efficient use of space, often achieved through gestures (swiping, pinching) and modals.

Touch-First Interaction

Unlike web UIs, which rely on keyboards and mice, mobile UIs are designed for touch inputs. This brings unique challenges and opportunities like buttons, sliders, and other touch targets must be large enough to interact with comfortably as well as gestures like swiping and pinching are used for navigation and interaction.

Sensors and Hardware Features

Mobile devices are equipped with sensors like accelerometers, gyroscopes, GPS, and cameras, which enable apps to create rich, context-aware experiences. Web UIs can't leverage these features as extensively. For example, in a mobile poll app, the UI could **rotate** or adapt dynamically based on device orientation (portrait vs. landscape), use location data to filter polls relevant to a specific area.

Push Notifications

Push notifications are a key feature of mobile apps, providing real-time updates directly on the user's device. These notifications keep users engaged and informed, but they also require careful integration with the UI to avoid being intrusive. For example, a notification might alert users about a new poll and clicking the notification opens the relevant screen in the app.

While these differences highlight the unique aspects of mobile UIs, many principles—like responsive design and performance optimization—are shared across both Web and Mobile platforms.

3.1 The Building Blocks of an Android App

Every Android application consists of a collection of essential components. These components work together to create a cohesive and functional app. Android defines four primary app components:

1. **Activity:** Represents a single screen in the app's UI.
2. **Service:** Runs in the background to perform long-running tasks without a user interface.
3. **Broadcast Receiver:** Responds to system-wide events or broadcasts, like receiving a message or detecting network changes.
4. **Content Provider:** Manages shared data between apps, such as contacts or media

Each of these components plays a specific role, and they communicate through an **intent-based messaging system** to share information or trigger actions.

3.1.1 Activity

An **Activity** is one of the fundamental building blocks of an Android app. We can think of it as the **screen a user can see when opening an app**—each screen he/she interacts with is typically an activity. Activities serve as the entry points for user interaction and manage the user interface (UI) components like buttons, text fields, and lists.

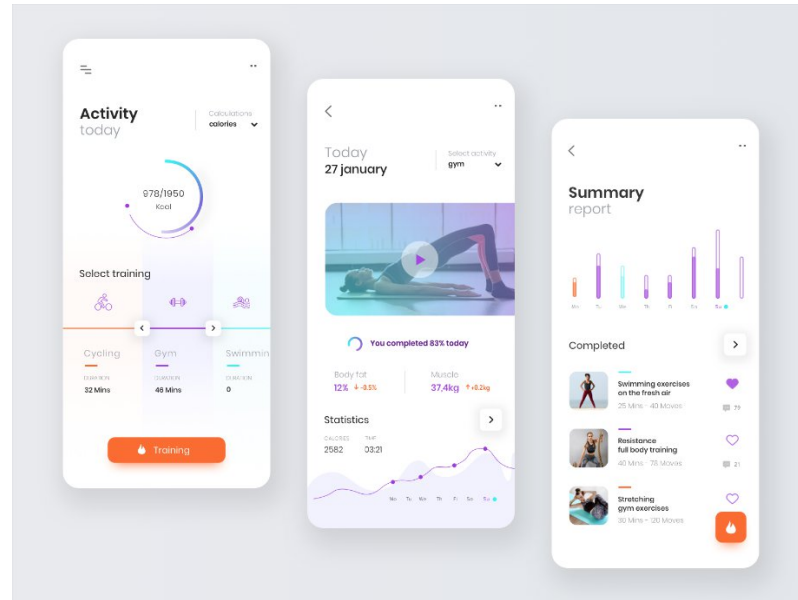


Figure 3.1. [Android Activity Screens](#)

Each Activity has many responsibilities: it creates and manages the **UI layout** for a specific screen, responds to **user actions** like button clicks or gestures, coordinates transitions between different screens (activities) within the app and works with Android's **Activity Lifecycle**, which ensures the app performs efficiently by managing what happens as the activity starts, pauses, resumes, or stops.

3.1.1.1 The Lifecycle of an Activity

An activity goes through a series of states during its lifetime, and Android provides lifecycle methods to respond to these changes. Here's a list of the most important ones:

- **onCreate()**. Called when the activity is created for the first time. It's where you initialize the UI.
- **onStart()**. Called when the activity becomes visible to the user.
- **onResume()**. Called when the activity is in the foreground and ready for interaction.
- **onPause()**. Called when the user navigates away but the activity is still partially visible.
- **onStop()**. Called when the activity is no longer visible.
- **onDestroy()**. Called when the activity is destroyed and resources are released.

For example, we can provide a lifecycle event handler using Kotlin:

```
class MainActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main) // Set the layout for this activity  
    }  
  
    override fun onResume() {  
        super.onResume()  
        // Code to update the UI or restart tasks  
    }  
}
```

Every time the activity is being recreated this *onCreate()* event handler will be triggered.

3.1.1.2 Activities interaction

Activities work in a **stack-like manner**, meaning that when we open a new activity, it sits on top of the previous one. We can navigate back to the previous activity using the back button.

For example, in a poll app, **MainActivity** might show a list of polls. Clicking on a poll Android opens **PollDetailsActivity**, which shows details about that specific poll. From there, the user can navigate back to the poll list.

Navigating to a new activity can be illustrated by the following code:

```
val intent = Intent(this, PollDetailsActivity::class.java)  
startActivity(intent)
```

We created an intent, a special form of command to the system, providing a class which is related to the PollDetails activity.

3.1.2 Service: Background Worker

In Android, a **Service** is a component that performs tasks in the background without showing a user interface. Unlike activities, which are user-facing, services are designed to handle operations that need to continue running even when the user isn't directly interacting with the app.

A service is ideal for tasks that run for a long time, like downloading a file or playing music, perform background tasks that need to continue even if the user switches to another app or synchronize data between the app and a server. For example, in a poll app, a service could periodically check for new polls or updates on a server or sync the results of a user's votes while the app isn't open.

There are three main types of services in Android:

1. **Foreground Service:** Runs actively and shows a notification (e.g., music players).
2. **Background Service:** Runs tasks in the background, but it can be stopped by the system to free resources.
3. **Bound Service:** Allows an app component to bind to the service and interact with it directly.

To create a service, we define it as a class that extends `Service` and override its key methods:

- **`onStartCommand()`:** Called when the service is started.
- **`onBind()`:** Used for bound services (not needed for background or foreground services).
- **`onDestroy()`:** Called when the service is stopped.

Here's a simple example of a service in a poll app that syncs poll data.

Firstly, we need to create a service:

```
class PollSyncService : Service() {  
    override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {  
        // Perform background syncing  
        Thread {  
            syncPollData() // Custom method to handle syncing  
        }.start()  
  
        // Keep the service running until explicitly stopped  
        return START_STICKY  
    }  
}
```

```

override fun onBind(intent: Intent?): IBinder? {
    return null // Not a bound service
}

override fun onDestroy() {
    super.onDestroy()
    // Cleanup resources
}

private fun syncPollData() {
    // Example: Simulate syncing with a server
    Thread.sleep(5000) // Simulating a delay
    println("Poll data synced!")
}
}

```

Then we can start the service, for example, from an activity:

```

val intent = Intent(this, PollSyncService::class.java)
startService(intent)

```

And we need to close it we can use the following code:

```

stopService(Intent(this, PollSyncService::class.java))

```

By default, services run on the main thread, meaning that long-running tasks could freeze the app's interface if not handled properly. To prevent this, developers should create separate threads for intensive tasks. While services remain valuable for certain use cases, modern alternatives like `WorkManager` or `JobScheduler` are often preferred for managing background tasks more efficiently and with better lifecycle awareness.

3.1.3 Broadcast Receiver: Listening to Events

Broadcast Receiver is an Android component that listens for and responds to system-wide or app-specific events. These events, called **broadcasts**, are sent out

by the Android system or other apps, and a Broadcast Receiver acts as a listener, waiting to respond when a specific event occurs.

Broadcast Receivers are lightweight and do not include a user interface. Instead, they run in the background, waiting to execute small, focused tasks. For example, a system broadcast might notify apps when the device's battery level is low or when the user connects to a Wi-Fi network.

For example, in a polling app we could use a broadcast receiver to detect changes in network connectivity and alert the user if they're offline.

```
class NetworkChangeReceiver : BroadcastReceiver() {  
    override fun onReceive(context: Context, intent: Intent) {  
        val isConnected = intent.getBooleanExtra("isConnected", false)  
        if (!isConnected) {  
            Toast.makeText(context, "No Internet Connection",  
                Toast.LENGTH_SHORT).show()  
        }  
    }  
}
```

3.1.4 Content Provider: Sharing Data

A **Content Provider** is an Android component that manages access to structured data and allows it to be shared between apps. It acts as a standardized interface that enables apps to securely query or modify data without having direct access to the underlying storage or database.

Content Providers are especially useful when data needs to be shared across different parts of your own app or between entirely separate apps. For example, the Android system itself uses Content Providers to manage shared data, like contacts, images, or audio files. Apps can query this data through a Content Provider without needing to know how it's stored internally.

A Content Provider exposes data through a **URI (Uniform Resource Identifier)**, which acts like an address for the data. Apps use this URI to query or update the data using standardized methods such as `query()`, `insert()`, `update()`, and `delete()`. These operations ensure that data access remains organized and secure.

For example, assume we have a database of poll questions and results. A Content Provider can expose this data so other components or apps can access it. Here's a simplified example of a Content Provider class:

```

class PollContentProvider : ContentProvider() {
    override fun onCreate(): Boolean {
        // Initialize database or data source
        return true
    }

    override fun query(uri: Uri, projection: Array<out String>?, selection: String?,
        selectionArgs: Array<out String>?, sortOrder: String?): Cursor? {
        // Handle querying the database
        return database.query("polls", projection, selection, selectionArgs, null, null,
sortOrder)
    }

    override fun insert(uri: Uri, values: ContentValues?): Uri? {
        // Handle inserting data into the database
        val id = database.insert("polls", null, values)
        return ContentUris.withAppendedId(uri, id)
    }

    // Similarly, implement update() and delete() methods
}

```

In this example we define globally available methods `query()`, `insert()`, `update()` and `delete()` allowing to access private data located in a SQLite database local storage or at a remote server.

Another app or component could use a `ContentResolver` to query the data provided by the Content Provider:

```

val cursor = contentResolver.query(
    Uri.parse("content://com.example.pollapp/polls"),
    null, null, null, null
)

```

3.1.5 Intents

The Android framework uses **intents** as the primary mechanism for communication between app components. Intents allow us to start an activity or a service, broadcast a message for other apps to respond to or transfer data between components.

There are two types of intent: implicit and explicit. **Explicit Intent** directly specifies the component to interact with. For example, Navigating from a poll list activity to a poll detail activity.

```
val intent = Intent(this, PollDetailActivity::class.java)
startActivity(intent)
```

Implicit Intent, on the other hand, specifies an action without naming a specific component, leaving the system to decide the best match. For example, sharing poll results:

```
val shareIntent = Intent(Intent.ACTION_SEND).apply {
    type = "text/plain"
    putExtra(Intent.EXTRA_TEXT, "Check out this poll!")
}
startActivity(Intent.createChooser(shareIntent, "Share via"))
```

3.2 Android Manifest

The **Android Manifest** file, named `AndroidManifest.xml`, is one of the most critical components of an Android app. It acts as a **blueprint** that defines essential information about the app for the Android operating system. Without it, the system wouldn't know how to launch or interact with an app. Every Android app must include a manifest file in its root directory.

The manifest serves several key purposes:

1. **Declares the App's Components:** It lists all the activities, services, broadcast receivers, and content providers that the app uses. This ensures the system knows what components exist and how to interact with them.
2. **Grants Permissions:** It specifies the permissions the app needs, such as accessing the internet, reading contacts, or using location services.
3. **Defines App Metadata:** It provides general information like the app's name, icon, theme, and version.
4. **Handles App Entry Points:** It specifies which activity should be launched when the user starts the app.

5. **Sets Hardware and Software Requirements:** It declares features or APIs the app depends on, like requiring a camera or a minimum Android version.

Content of the Manifest File

A typical AndroidManifest.xml file contains several key sections:

1. **Root Element:** The `<manifest>` element is the container for the entire file and includes the app's package name, which serves as a unique identifier.
2. **Application Element:** The `<application>` element includes app-level attributes like the app's icon, theme, and any global configurations. Inside it, you declare components like activities, services, and broadcast receivers.

```
<application
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme">
    <!-- Activities -->
    <activity android:name=".MainActivity">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
</application>
```

3. **Permissions:** The manifest lists permissions the app needs. For example, an app that requires internet access would include:

```
<uses-permission android:name="android.permission.INTERNET" />
```

Hardware and Software Features: Use `<uses-feature>` and `<uses-sdk>` elements to specify requirements like a camera or a minimum Android version:

```
<uses-sdk android:minSdkVersion="21" android:targetSdkVersion="34" />
```

3.3 Android XML Layout UI Basics

The traditional XML-based UI approach in Android involves defining the user interface in XML files separate from the application logic. These XML layouts describe the structure, appearance, and behavior of UI components like `TextView`,

Button, and `LinearLayout`. Developers link these XML layouts to activities or fragments in the code using `setContentView()`. In the picture below, we can see the Kotlin language code of an Activity which is related to a XML-file which describe its user interface as a XML-tree of UI-elements (Figure 3.2).

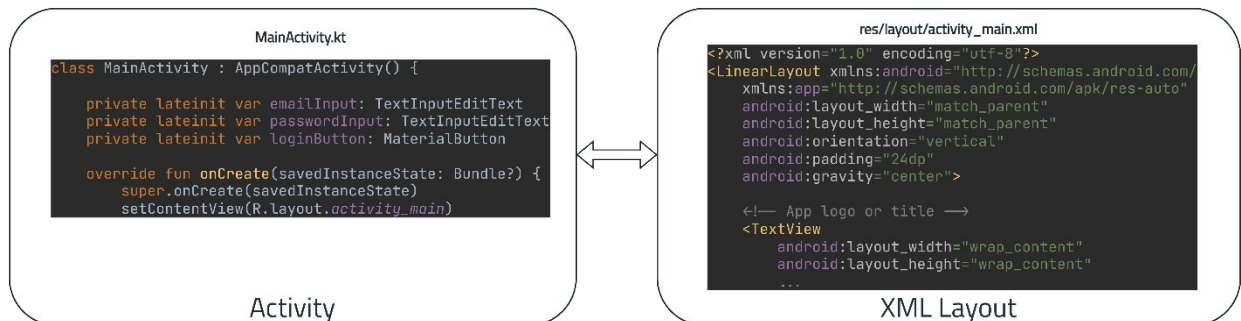


Figure 3.2. Activity and its XML UI Representation

3.3.1 Layout XML-file

In Android, a layout XML file defines the visual structure of a user interface for an activity or fragment. These files are stored in the `res/layout` directory and describe the arrangement of UI elements using a hierarchical structure.

Every layout XML file starts with a **root element** that serves as the container for other UI components. Common root elements include:

- **LinearLayout**: Aligns child views in a vertical or horizontal direction.
- **RelativeLayout** or **ConstraintLayout**: Allows more flexible and complex positioning of child views.
- **FrameLayout**: A simple layout to stack views on top of each other.
- **ScrollView**: For scrollable content.

The example:

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <!-- Child views go here -->
</LinearLayout>
```

Child UI Elements

- The root layout contains child elements such as `TextView`, `Button`, `ImageView`, `EditText`, and other widgets.
- Each element specifies its properties (e.g., size, color, text) using attributes.

```
<TextView
    android:id="@+id/textView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello, World!"
    android:textSize="18sp" />
```

Attributes

- **layout_width and layout_height:** Define the size of elements (`match_parent`, `wrap_content`, or specific dimensions).
- **id:** Uniquely identifies a UI component for reference in Kotlin/Java code.
- Other commonly used attributes:
- **text:** Text to display in a widget like `TextView`.
- **padding and margin:** Control spacing inside or outside a view.
- **gravity and layout_gravity:** Align content within or relative to the parent.

Resource References

Attributes often reference external resources like strings, dimensions, and drawables using the `@` symbol.

- `@string/app_name:` Fetches a string defined in `res/values/strings.xml`.
- `@drawable/icon:` Refers to an image file in the `res/drawable` folder.
- `@color/primary:` References a color defined in `res/values/colors.xml`.

Event Attributes

- XML can directly declare event listeners for UI elements, though it is generally better to set these in Kotlin/Java.
 - Example: `android:onClick="onButtonClick"` links to a method in the activity.

Commonly Nested Views

- **Layouts:** Child layouts for nested structures (e.g., `ConstraintLayout` inside `ScrollView`).
- **Widgets:** Interactive elements like `Button`, `EditText`.

- **Containers:** Group or organize elements, such as RecyclerView or ViewPager.

For example:

```
<ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <Button
        android:id="@+id/buttonSubmit"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Submit" />
</ConstraintLayout>
```

The XML-based approach promotes a clear separation between the UI and the code, making layouts reusable and easier to visualize. However, it can lead to challenges such as verbose syntax, managing complex UI updates, and difficulties in creating dynamic or stateful UIs.

3.3.2 The basic Android Studio IDE Example

[Android Studio](#) is the official Integrated Development Environment (IDE) for Android app development, providing a comprehensive suite of tools to streamline the development process. It offers a smart code editor that supports code completion, syntax highlighting, and linting for languages like Java, Kotlin, and XML, which helps developers write clean and error-free code. The IDE also includes powerful refactoring tools that simplify code organization while adhering to Android best practices, and it comes with an integrated debugger to inspect variables, set breakpoints, and analyze the app's behavior in real-time.

Android Studio integrates seamlessly with the Android Emulator, allowing developers to test their apps on virtual devices with varying screen sizes, Android versions, and other configurations. This feature is essential for simulating real-world conditions, such as network or battery states, while fine-tuning app performance.

Let's create a simple Android application showing Login Page Layout.

After Android Studio is shown we need to open *File-New-New Project* menu item (Figure 3.3) and choose *Basic Views Activity*.

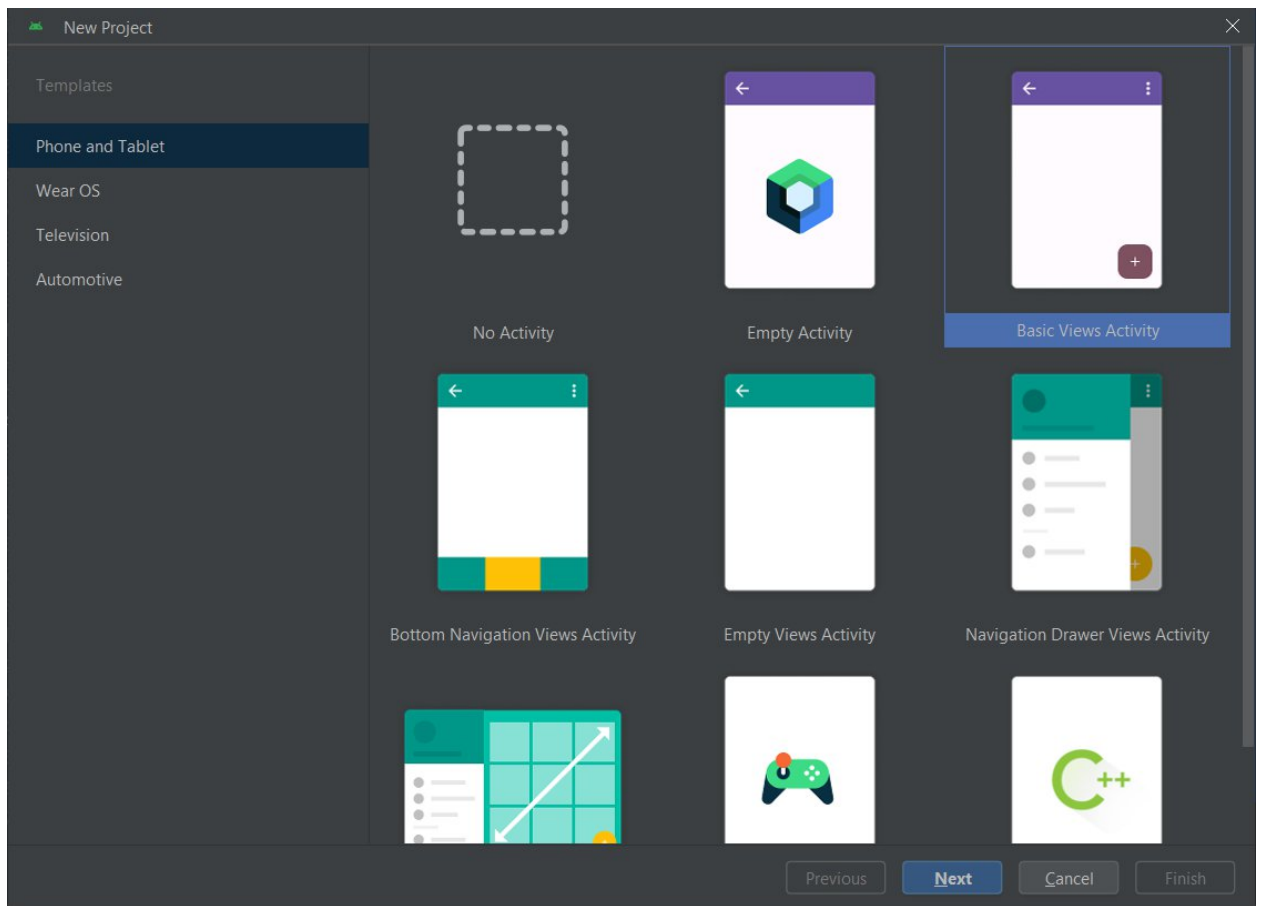


Figure 3.3. Android Studio New Project Screen

After clicking the *Next* button, a user will see the following screen (Figure 3.4). Among the parameters to choose there are the name of the project, programming language (Java or Kotlin), and minimum SDK supported by the application. Then the *Finish* button should be clicked.

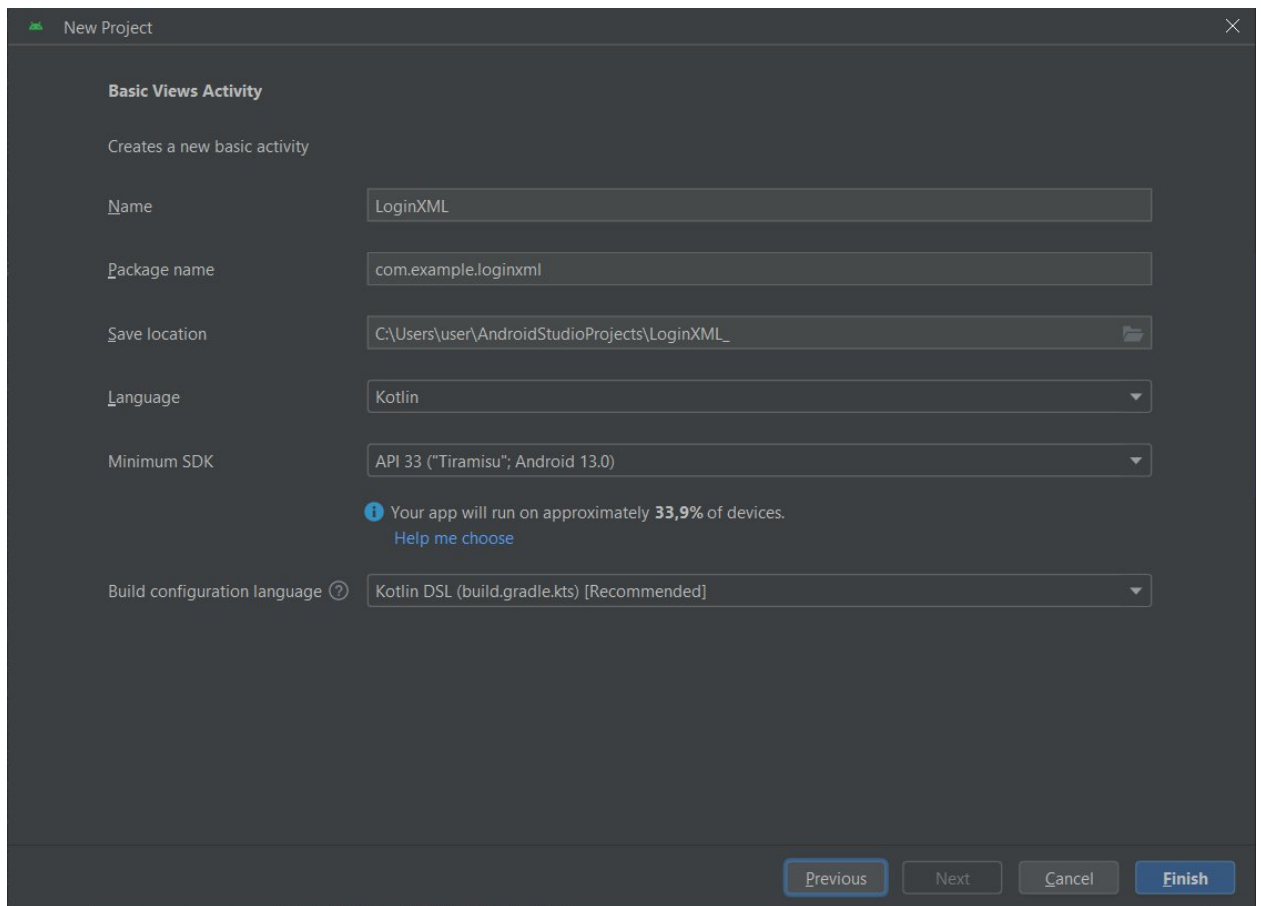


Figure 3.4. New Project Parameters

It'll create a set of core application files needed for its successful execution. Among the files there is the `AndroidManifest.xml` which provides essential information to the Android operating system about the app's structure, components, permissions, and other metadata. The key attributes in the file are inside `<application>` and its `<activity>` elements. They clarify general settings of the application and define the activity available in the project. For instance, `<intent-filter>` (MAIN, LAUNCHER) in the below example indicates that the activity is the main one and should be shown firstly:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportRtl="true"
```

```

        android:theme="@style/Theme.LoginXML"
        tools:targetApi="31">
        <activity
            android:name=".MainActivity"
            android:exported="true"
            android:theme="@style/Theme.LoginXML">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>

```

The `android:name` property of the activity shows the link to the Kotlin-file of the Activity, which includes the behavior of the activity: how it connects to the XML Layout file, defines various event handles like on-click for buttons etc.

The next file is *MainActivity.kt* located in the `java/com/example/loginxml/` folder of the application. It defines the main functionality of the login screen in an Android app. For our login page example let's replace the default file content with the relevant code.

It extends `AppCompatActivity` and uses modern UI components like `TextInputEditText` for email and password inputs and `MaterialButton` for the login button:

```

package com.example.loginxml

import android.os.Bundle
import android.widget.Toast
import androidx.appcompat.app.AppCompatActivity
import com.google.android.material.button.MaterialButton
import com.google.android.material.textfield.TextInputEditText

class MainActivity : AppCompatActivity() {

    private lateinit var emailInput: TextInputEditText
    private lateinit var passwordInput: TextInputEditText
    private lateinit var loginButton: MaterialButton

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
    }
}

```

```

setContentView(R.layout.activity_main)

// Initialize views
emailInput = findViewById(R.id.emailInput)
passwordInput = findViewById(R.id.passwordInput)
loginButton = findViewById(R.id.loginButton)

// Set click listener for login button
loginButton.setOnClickListener {
    val email = emailInput.text.toString()
    val password = passwordInput.text.toString()

    if (validateInput(email, password)) {
        // Perform login logic here
        performLogin(email, password)
    }
}

private fun validateInput(email: String, password: String): Boolean {
    var isValid = true

    // Validate email
    if (email.isEmpty()) {
        emailInput.error = "Email is required"
        isValid = false
    } else if (!android.util.Patterns.EMAIL_ADDRESS.matcher(email).matches()) {
        emailInput.error = "Invalid email format"
        isValid = false
    }

    // Validate password
    if (password.isEmpty()) {
        passwordInput.error = "Password is required"
        isValid = false
    } else if (password.length < 6) {
        passwordInput.error = "Password must be at least 6 characters"
        isValid = false
    }

    return isValid
}

```

```

private fun performLogin(email: String, password: String) {
    // Add your login logic here (e.g., API call, database check, etc.)
    Toast.makeText(this, "Login successful!", Toast.LENGTH_SHORT).show()
}
}

```

In the `onCreate()` method, the UI components are initialized using `findViewById()`. A click listener is set for the login button to capture the input values. The `validateInput()` method checks if the email is valid and if the password is at least 6 characters long, showing error messages if the inputs are incorrect.

If the input is valid, the `performLogin()` method is called, which currently simulates a successful login with a toast message. In a real app, this method would handle the authentication logic, such as making an API call or checking credentials.

The next file is XML-Layout file *activity_main.xml* located in the *res/layout/* folder. Its content is provided below:

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="24dp"
    android:gravity="center">

    <!-- App logo or title -->
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Login"
        android:textSize="24sp"
        android:textStyle="bold"
        android:layout_marginBottom="32dp"/>

    <!-- Email input field -->
    <com.google.android.material.textfield.TextInputLayout
        android:id="@+id/emailLayout"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginBottom="16dp"
        style="@style/Widget.MaterialComponents.TextInputLayout.OutlinedBox">

        <com.google.android.material.textfield.TextInputEditText

```

```

        android:id="@+id/emailInput"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Email"
        android:inputType="textEmailAddress"/>
</com.google.android.material.textfield.TextInputLayout>

<!-- Password input field -->
<com.google.android.material.textfield.TextInputLayout
    android:id="@+id/passwordLayout"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginBottom="24dp"
    app:enablePasswordToggle="true"
    style="@style/Widget.MaterialComponents.TextInputLayout.OutlinedBox">

    <com.google.android.material.textfield.TextInputEditText
        android:id="@+id/passwordInput"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Password"
        android:inputType="textPassword"/>
</com.google.android.material.textfield.TextInputLayout>

<!-- Login button -->
<com.google.android.material.button.MaterialButton
    android:id="@+id/loginButton"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Login"
    android:padding="12dp"/>

</LinearLayout>

```

The layout file defines a simple login screen using a `LinearLayout` with a vertical orientation to arrange UI components. It starts with a `TextView` for the app title, displaying "Login" in bold with a font size of 24sp and a margin at the bottom.

Next, there are two `TextInputLayout` containers, each wrapping a `TextInputEditText`. The first one is for the email input field, styled with an outlined box and configured to accept email input. The second is for the password input field, with a password toggle feature enabled to show or hide the password. Both input fields include hints and are sized to match the parent layout width.

Finally, a `MaterialButton` is provided as the login button, which spans the full width of the screen and includes padding for better touch accessibility.

3.3.2.1 The project execution

In Android Studio, there are two main ways to execute and test Android applications: **on a real device** and **using an emulator**. Each has its own setup and advantages:

Real Device Execution

This involves running the application on an actual Android smartphone or tablet. The device must be connected to the development computer using a USB cable. In order to setup the Android Device we need

- Enable **Developer Options** on the device (found in Settings, often by tapping "Build Number" in About Phone multiple times).
- Turn on **USB Debugging** within Developer Options.
- Connect the device to your computer via USB.
- Android Studio will detect the device and list it as a deployment option.

Such approach allows to tests the app in a real-world environment with actual hardware and system behavior as well as to be more accurate for performance analysis and interaction with physical sensors (like GPS, accelerometer, etc.).

Emulator Execution

An emulator is a virtual Android device that mimics the functionality of a real device on the computer. This approach is more convenient for quick testing without additional hardware.

To setup this approach we need:

Open the Device Manager in Android Studio (Figure 3.5) and create a virtual device by selecting a hardware profile, e.g., Pixel 8 (Figure 3.6).

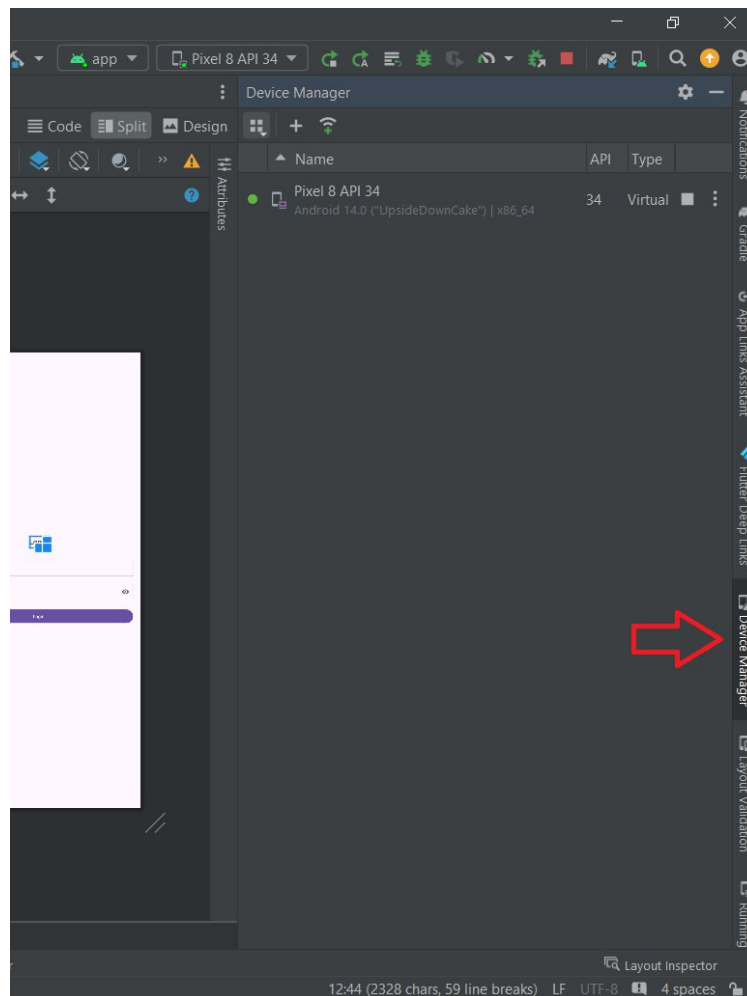


Figure 3.5. Device Manager

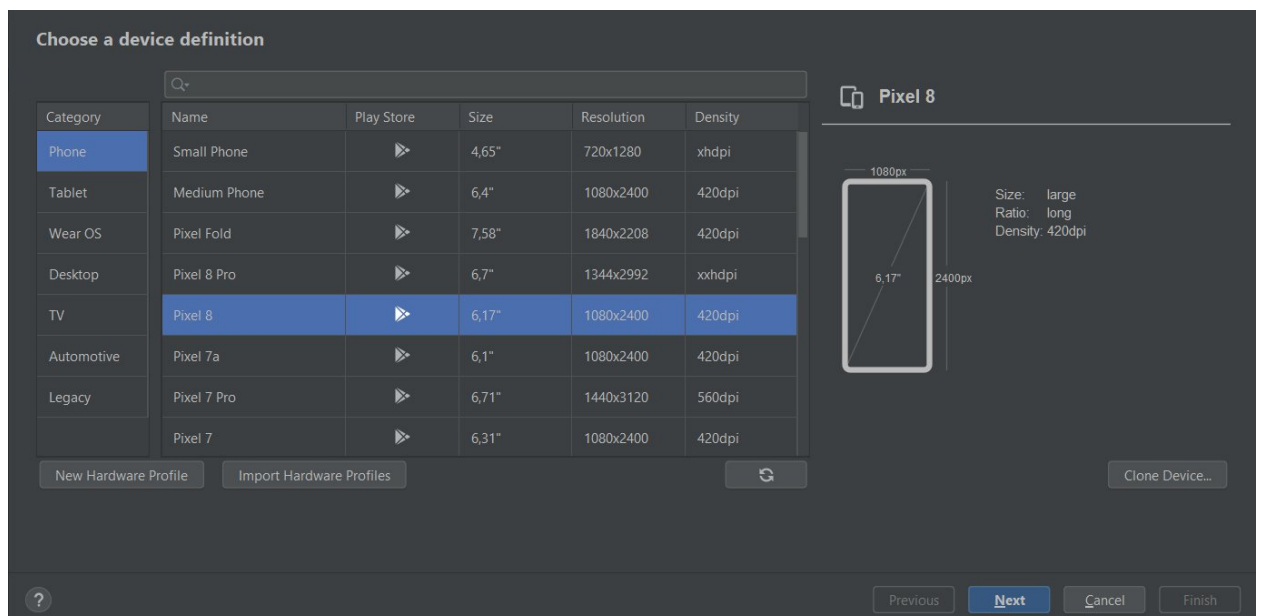


Figure 3.6. Emulator Hardware Profile

Then we need to choose an Android system image (e.g., API Level 34). See Figure 3.7.

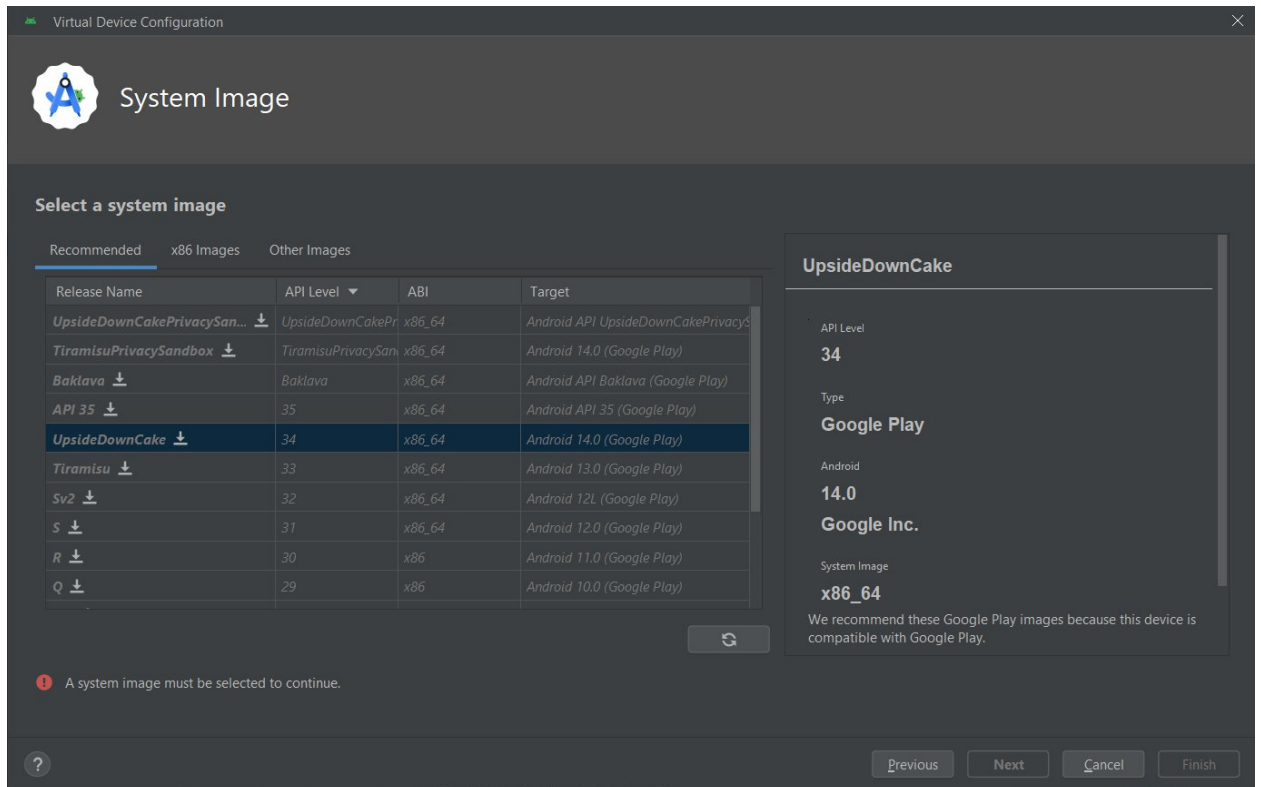


Figure 3.7. Emulator System Image

After that we can start the emulator and run the app (Figure 3.8).

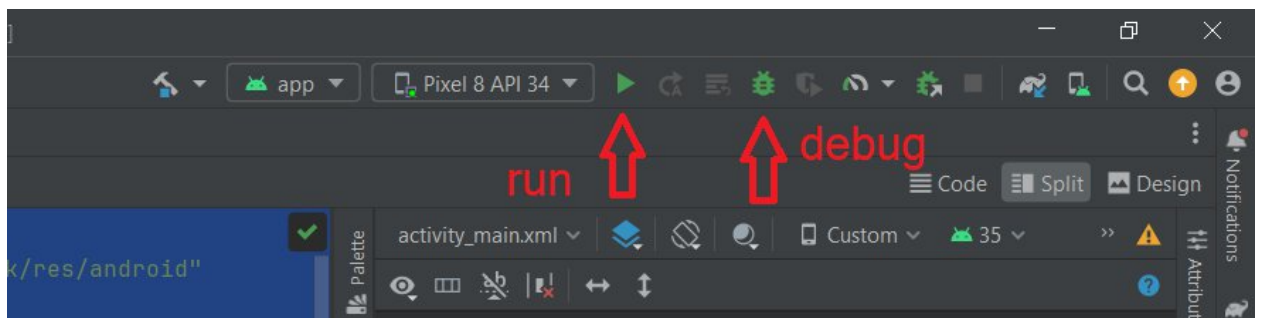


Figure 3.8. Run or Debug the Application

The final view of the emulator screen is shown on the next picture (Figure 3.9).

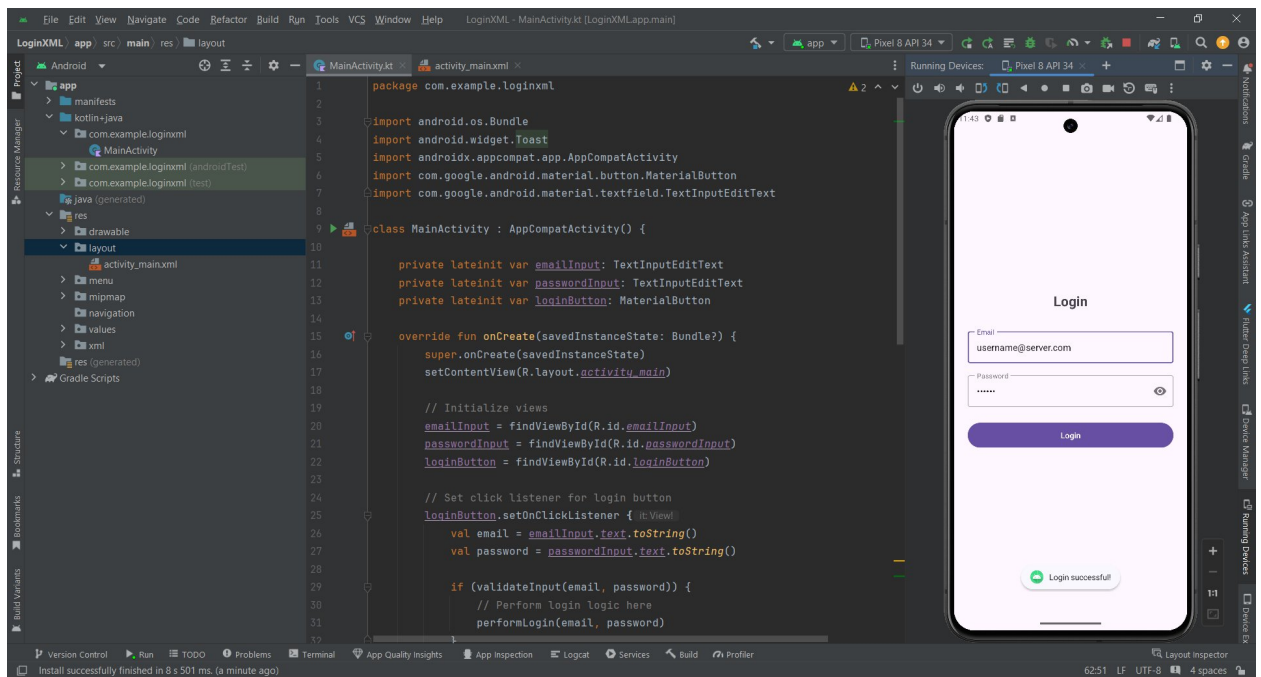


Figure 3.9. The Final Emulator Screen

The final project can be downloaded and run from the [Github repository](#).

3.4 Android Compose UI Layout

Jetpack Compose represents a fundamental change in how Android applications are developed. It uses a **declarative programming model**, meaning that developers describe what the UI should look like for a particular state of the application, rather than how to construct or manipulate it step by step. Whenever the app's state changes, Compose automatically updates the UI to reflect those changes.

This declarative approach simplifies UI development in several ways. For instance, it eliminates the need for tedious calls to `findViewById`, a common complicated point in traditional Android development. Instead, developers write modular UI components called **Composables**. These are reusable, self-contained functions that can be combined to build complex user interfaces.

Composable functions, identified by the `@Composable` annotation, are unique functions that describe the layout and behavior of UI elements. These functions don't return traditional values like integers or strings; instead, they directly define what the user interface should look like. For instance, a simple composable function might define a greeting message:

```

@Composable
fun Greeting() {
    Box(
        modifier = Modifier.fillMaxSize(),
        contentAlignment = Alignment.Center
    ){
        Text(text = "Hello Compose!")
    }
}

```

Hello Compose!

Here, the `Text` function, another composable, displays the string "Hello, World!" on the screen. It's as straightforward as writing a sentence.

3.4.1 Common Layouts in Compose

Layouts in Compose are flexible and easy to use. Here are the most common ones.

Column

The `Column` layout arranges its child composables in a vertical sequence, one below the other. It's great for stacking UI elements like text, buttons, or images when you want them aligned in a single column. We can also customize spacing, alignment, and padding between items.

```

@Composable
fun ColumnExample() {
    Column(
        modifier = Modifier.padding(16.dp),
        verticalArrangement =
Arrangement.spacedBy(8.dp)
    ){
        Text(text = "Item 1")
        Text(text = "Item 2")
        Text(text = "Item 3")
    }
}

```

Item 1

Item 2

Item 3

Row

The `Row` layout arranges its child composables horizontally in a single line, from left to right by default. It is ideal for displaying content like navigation menus, icons, or

inline text. We can control spacing and alignment to fit a design requirements. The `Row` arranges its children horizontally.

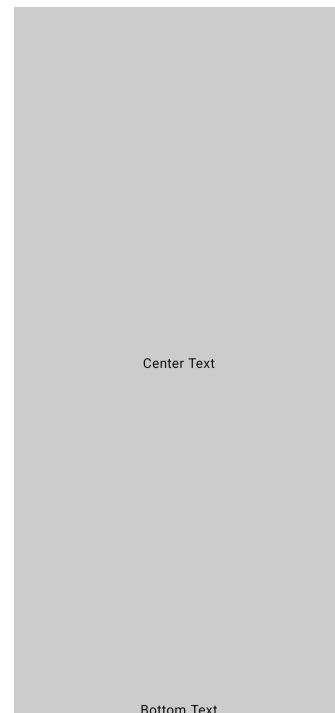
```
@Composable
fun RowExample() {
    Row(
        modifier = Modifier.padding(16.dp),
        horizontalArrangement =
Arrangement.spacedBy(8.dp)
    ){
        Text(text = "Left")
        Text(text = "Middle")
        Text(text = "Right")
    }
}
```

Left Middle Right

Box

The `Box` layout allows us to stack composables on top of each other. It is perfect for overlaying elements like a button on an image or centering one element within another. We can use alignment modifiers to position the child elements precisely.

```
@Composable
fun BoxExample() {
    Box(
        modifier = Modifier
            .fillMaxSize()
            .background(Color.LightGray)
    ){
        Text(
            text = "Bottom Text",
            modifier =
Modifier.align(Alignment.BottomCenter)
        )
        Text(
            text = "Center Text",
            modifier = Modifier.align(Alignment.Center)
        )
    }
}
```



Modifier for Customization

The `Modifier` is a powerful tool in Compose that allows us to modify the appearance and behavior of composables. We can apply padding, borders, size constraints, backgrounds, and more.

```
@Composable
fun StyledText() {
    Text(
        text = "Stylized Text", color = Color.White,
        modifier = Modifier
            .background(Color.DarkGray)
            .padding(16.dp)
    )
}
```

Stylized Text

Spacer

The `Spacer` is a simple but useful composable that adds empty space between other composables. It's particularly handy for fine-tuning the spacing in `Column`, `Row`, or other layouts.

```
@Composable
fun SpacerExample() {
    Column {
        Text(text = "Above")
        Spacer(modifier = Modifier.height(16.dp))
        Text(text = "Below")
    }
}
```

Above

Below

LazyColumn and LazyRow

These layouts are optimized for displaying scrollable lists. They only compose and draw the visible items, making them memory-efficient for rendering large data sets. Use `LazyColumn` for vertical scrolling and `LazyRow` for horizontal scrolling.

```
@Composable
fun LazyColumnExample() {
    LazyColumn(
        modifier = Modifier.padding(16.dp),
        verticalArrangement =
            Arrangement.spacedBy(8.dp)
    ) {
        items(3) { index ->
            Text(text = "Item $index", modifier =
                Modifier.padding(8.dp))
        }
    }
}
```

Item 0

Item 1

Item 2

3.5 State Management in Jetpack Compose

State management is a core concept in any UI framework, as it determines how data flows and updates the UI in response to changes. We are already familiar with state management in React (see section 2), so we can find some similarities in Jetpack Compose [9], but also key differences due to the nature of how Compose handles UI updates and recomposition.

3.5.1 React vs Jetpack Compose

Both React and Jetpack Compose follow declarative UI paradigms, meaning the UI is derived from the current state rather than being manually manipulated. However, there are differences in how state is managed and propagated:

State Storage and Ownership

React: State is typically managed using `useState`, `useReducer`, or external state management libraries like `Redux` or `Recoil`. The state is tied to functional components and follows the "unidirectional data flow" model.

Jetpack Compose: State is managed using `remember`, `mutableStateOf`, and `State<T>`. The Jetpack Compose model also encourages unidirectional data flow but is more tightly integrated with Android's architecture components like `ViewModel`.

Reactivity and Recomposition

React: When a state variable changes, React triggers a re-render of the affected component. Virtual DOM diffing ensures only necessary updates occur.

Jetpack Compose: When a state variable changes, Compose automatically recomposes only the affected UI elements. It does not rely on a Virtual DOM but instead directly manages recompositions efficiently at the function level.

Scoped vs. Global State

React: State is either local to a component (`useState`) or managed globally using `Redux`, `Context API`, or other state management solutions.

Jetpack Compose: State can be local (`remember` + `mutableStateOf`) or shared using `ViewModel` with `LiveData` or `StateFlow` for reactive updates.

3.5.2 State in Jetpack Compose

In Jetpack Compose [10], state represents a value that can change over time. The UI recomposes whenever the state updates. To manage state effectively, Compose provides tools like *remember*, *mutableStateOf*, and State Hoisting along with *ViewModel* pattern preventing losing of the composable variables in case of system wide changes of the current activity (screen rotation, 'dark mode' etc).

3.5.2.1 Remembering State

We use `remember` to retain state across recompositions within a composable function.

```
@Composable
fun Counter() {
    var count by remember { mutableStateOf(0) }

    Button(onClick = { count++ }) {
        Text("Count: $count")
    }
}
```

Here, `count` persists across recompositions but resets when the composable leaves the Composition (for example, when the parent activity becomes invisible). It works best for simple UI state within a single composable.

3.5.2.2 State Hoisting

State hoisting is the practice of moving state up the hierarchy to make it more reusable and manageable.

```
@Composable
fun CounterScreen() {
    var count by remember { mutableStateOf(0) }

    CounterDisplay(count)
    CounterControls { count++ }
}

@Composable
```

```

fun CounterDisplay(count: Int) {
    Text("Count: $count")
}

@Composable
fun CounterControls(onIncrement: () -> Unit) {
    Button(onClick = onIncrement) {
        Text("Increment")
    }
}

```

Here, `count` is lifted to `counterScreen`, making `CounterDisplay` and `CounterControls` stateless. Stateless composables are always a better choice in terms of reusability because they don't depend on any internal state, making them more predictable and easier to reuse across different parts of an application [11].

3.5.2.3 Using ViewModel for State Management

In Jetpack Compose, state stored within a composable is lost when the composable leaves the composition [12]. This becomes problematic during configuration changes such as screen rotations, which cause the entire activity to be recreated. The `viewModel` class helps persist state across such changes.

```

class CounterViewModel : ViewModel() {
    private val _count = mutableStateOf(0)
    val count: State<Int> = _count

    fun increment() {
        _count.value++
    }
}

@Composable
fun CounterWithViewModel(viewModel: CounterViewModel = viewModel()) {

```

```

val count by viewModel.count

Column {
    Text("Count: $count")
    Button(onClick = { viewModel.increment() }) {
        Text("Increment")
    }
}
}

```

Using `viewModel` ensures state persistence across screen rotations and similar system wide events.

3.5.2.4 State and Side Effects

State in Jetpack Compose doesn't exist in isolation — sometimes we need to react to state changes or trigger side effects like navigation, showing messages, or launching coroutines. For this, Compose provides *side-effect APIs* such as `LaunchedEffect`, `rememberUpdatedState`, and `SideEffect`.

LaunchedEffect is commonly used when we need to start a coroutine in response to a state or lifecycle event.

```

@Composable
fun Greeting(name: String) {
    LaunchedEffect(name) {
        // Called every time 'name' changes
        println("Hello $name")
    }

    Text("Hello, $name!")
}

```

If `name` changes, the coroutine inside `LaunchedEffect` is cancelled and restarted. It's ideal for tasks like animation, data loading, or responding to navigation events.

rememberUpdatedState allows us to safely capture a new value inside long-lived effects, such as coroutines or callbacks, without restarting them.

```

@Composable

```

```

fun Timer(onTimeout: () -> Unit) {
    val currentOnTimeout by rememberUpdatedState(onTimeout)

    LaunchedEffect(Unit) {
        delay(3000)
        currentOnTimeout()
    }

    Text("Waiting...")
}

```

This avoids stale state being used in long-lived operations.

SideEffect is useful for performing actions that need to run after every successful recomposition, such as updating external UI systems (e.g., status bar color).

```

@Composable
fun SetStatusBarColor(color: Color) {
    SideEffect {
        // Imagine a function that sets the system status bar color
        SystemStatusBar.setColor(color)
    }
}

```

We need to use side-effect APIs wisely avoiding put heavy operations or recomposition triggers inside them, as this can lead to unnecessary recomposition cycles or performance issues [13].

3.5.2.5 Derived State

Sometimes, we need to compute a value from a state in a way that avoids unnecessary recomposition. `derivedStateOf` helps optimize such derived computations by memoizing the result until dependencies change.

```

@Composable
fun FilteredList(searchQuery: String, items: List<String>) {
    val filteredItems by remember(searchQuery, items) {
        derivedStateOf {

```

```
        items.filter { it.contains(searchQuery, ignoreCase = true) }
    }
}

LazyColumn {
    items(filteredItems) {
        Text(it)
    }
}
}
```

By wrapping the computation in `derivedStateOf`, Compose only recomputes the result when `searchQuery` or `items` change. This is especially useful for performance when dealing with large datasets or expensive transformations [14].

4 Cross-platform UI Development using Flutter

Cross-platform development refers to the practice of writing a single codebase that can run on multiple platforms — such as Android, iOS, web, and desktop — with minimal platform-specific changes. Instead of building separate native applications for each target, developers use tools and frameworks that abstract platform differences and provide a unified development experience.

Cross-platform development offers several compelling advantages, among them are:

- **Code Reuse:** Share up to 90% of the code across platforms.
- **Faster Development:** Build once, deploy everywhere.
- **Consistent UI/UX:** Unified look and feel across devices.
- **Simpler Maintenance:** Fix bugs or add features in one place.
- **Lower Costs:** Fewer resources and less dev time needed.

While the approach offers many efficiencies, it also introduces certain trade-offs:

- **Slower Performance:** Not as fast as fully native apps.
- **Feature Gaps:** Some native functions need extra work.
- **UI Glitches:** Can be hard to match native design exactly.
- **Learning Curve:** Need to learn how the framework talks to native code.

Table 4.1 presents some of the most popular cross-platform frameworks.

Table 4.1. Major Frameworks

Framework	Language	Platforms Supported	Key Features
Flutter	Dart	Android, iOS, Web, Desktop	Native-like UI, Hot Reload, High Performance
React Native	JavaScript	Android, iOS	JS Ecosystem, Native Modules
Xamarin	C#/.NET	Android, iOS, Windows	.NET integration, Native Bindings
Ionic	JavaScript/HTML	Android, iOS, Web	WebView-based, PWA support

While each has its strengths, Flutter stands out for its high performance, expressive UI, and robust developer tooling.

Flutter is an open-source UI toolkit developed by Google for building natively compiled applications for mobile (Android, iOS), web, and desktop from a single codebase [15]. It uses the Dart language and provides a rich set of widgets and tools.

Key Features:

- **Cross-platform:** You can write your code once and run it on Android, iOS, web, desktop, and embedded devices, saving development time and resources.
- **Hot Reload:** This feature allows developers to quickly view the results of changes without restarting the application, greatly speeding up the development process.
- **Rich UI Library:** Flutter provides numerous built-in widgets for UI design, and you can also create custom widgets for more flexibility.
- **Strong Community Support:** With contributions from developers around the world and active support from Google, Flutter is evolving rapidly.

Here is the basic Flutter app demonstrating that displays a greeting message.

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text('Hello Flutter')),
        body: Center(child: Text('Welcome!')),
      ),
    );
  }
}
```

This app creates a `MaterialApp` with a `Scaffold`, an `AppBar` with a title, and a `Center` widget that contains a `Text` widget.

4.1 Architectural layers

Flutter is designed as an extensible, layered system. It exists as a series of independent libraries that each depend on the underlying layer. No layer has

privileged access to the layer below, and every part of the framework level is designed to be optional and replaceable.

At the core of Flutter is the **Flutter engine**, which is mostly written in C++ and supports the primitives necessary to support all Flutter applications. The engine is responsible for rasterizing composited scenes whenever a new frame needs to be painted. It provides the low-level implementation of Flutter's core API, including graphics (through [Impeller](#) on iOS, Android, and desktop (behind a flag), and [Skia](#) on other platforms), text layout, file and network I/O, accessibility support, plugin architecture, and a Dart runtime and compile toolchain (see Figure 4.1).

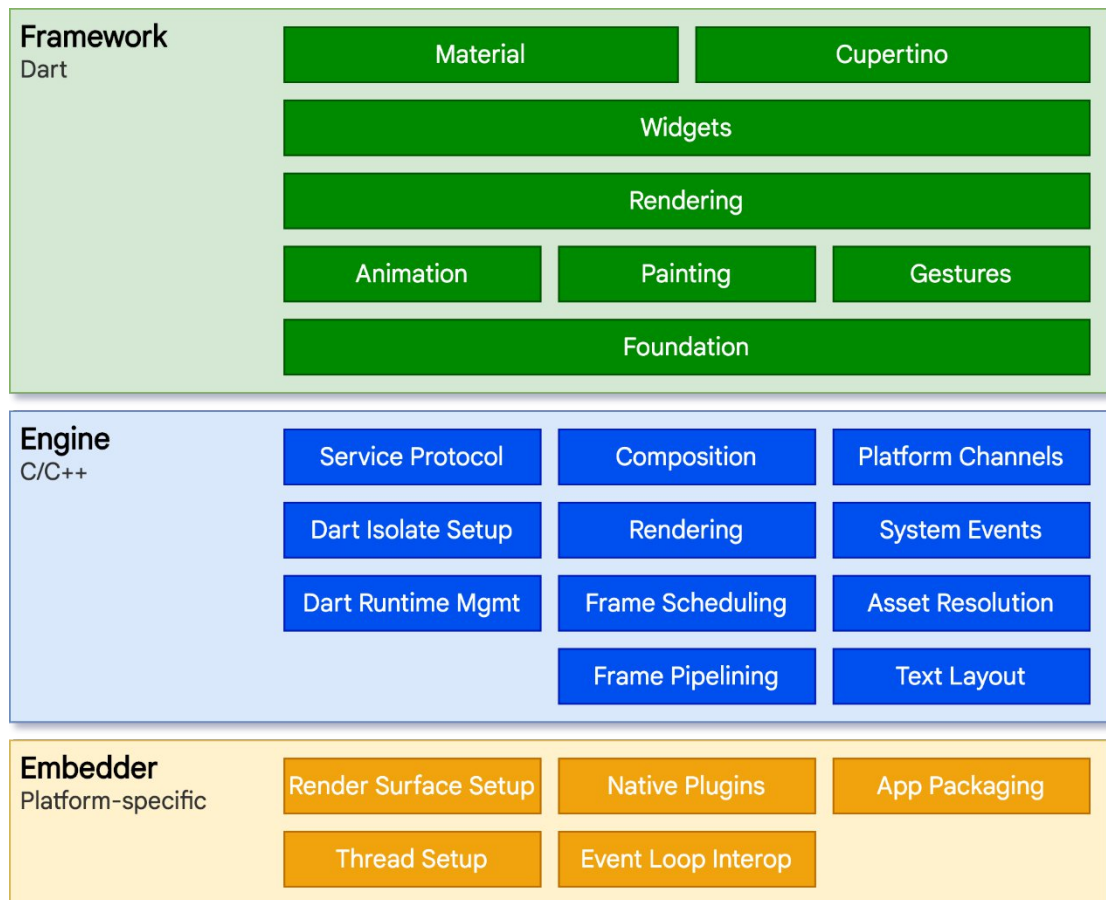


Figure 4.1. Flutter architecture

4.2 Setting Up the Environment

Follow these steps to get your environment ready for Flutter development.

Installation Steps

1. Install the Flutter SDK

Download and install the Flutter SDK from flutter.dev.

2. Update System Path

Add the flutter/bin directory to your system's environment variables to access Flutter commands globally.

3. Verify Setup

Run **flutter doctor** in the terminal to check for any missing dependencies or configurations.

IDE Setup

Android Studio

Install the Flutter and Dart plugins via the plugin marketplace.

Set up an Android emulator or connect a physical device for testing.

Visual Studio Code

Install the Flutter and Dart extensions from the Extensions Marketplace.

Use the integrated terminal to run Flutter commands and manage your project.

Device Setup

Enable Developer Mode

On Android: Enable Developer Options and USB Debugging in device settings.

On iOS: Enable Developer Mode via Xcode and authorize the device.

List Devices

Use flutter devices to list all connected emulators and physical devices.

Run the App

Launch the app on a connected device with flutter run.

4.3 Dart language

Dart is an object-oriented, class-based language optimized for UI development [16]. It powers Flutter and supports features such as null safety, asynchronous programming, and hot reload.

Setting Up Dart

Option 1: Install Dart SDK

Download Dart SDK from <https://dart.dev/get-dart>.

Add to PATH (so you can run dart in terminal).

Verify installation:

```
dart --version
```

Should print something like:

```
Dart SDK version: 3.4.0 (stable)
```

Option 2: Use DartPad (No Installation Needed)

<https://dartpad.dev/> – Write and run Dart code in the browser.

Writing The First Dart Program

Create a file `hello.dart` and write:

```
// This is a comment (ignored by Dart)
void main() { // Entry point of every Dart program
  print("Hello, Dart!"); // Prints to console
}
```

Run it:

```
dart hello.dart
```

Output:

```
Hello, Dart!
```

Explanation:

`void main()` → The starting point of execution.

`print()` → Outputs text to the console.

Semicolons (`;`) → Required at the end of statements.

Variables & Data Types

Dart supports type inference (`var`) and explicit typing.

```
void main() {
  var name = "John"; // Dart infers `name` as String
  int age = 25;      // Explicitly declared as integer
  double height = 5.9; // Decimal number
  bool isStudent = true; // true or false
}
```

```
print("$name is $age years old."); // String interpolation
}
```

Constants

final → Assigned once at runtime.

const → Compile-time constant (must be known before execution).

```
final city = "New York"; // Can be set once
```

```
const pi = 3.14; // Must be known at compile time
```

Common Data Types

In Dart, several core data types are commonly used:

- **int** is used for whole numbers, such as 42.
- **double** represents decimal numbers, for example, 14.3.
- **String** holds text, using UTF-16 encoding, like "Hello".
- **bool** is a Boolean type that can be either true or false.
- **List** is an ordered collection of values (similar to an array), e.g., [1, 2, 3].
- **Map** stores key-value pairs, much like a dictionary, such as {"key": "value"}.

Example:

```
List<int> numbers = [1, 2, 3]; // List of integers
Map<String, dynamic> person = { // Map with String keys
  "name": "Bob",
  "age": 30,
  "isStudent": false
};
```

Control Flow (Conditionals & Loops)

If-Else Statements

```
int age = 18;
if (age >= 18) {
```

```
    print("You are an adult.");
} else if (age >= 13) {
    print("You are a teenager.");
} else {
    print("You are a child.");
}
```

For Loop

```
for (int i = 0; i < 5; i++) {
    print(i); // Prints 0, 1, 2, 3, 4
}
```

While Loop

```
int count = 0;
while (count < 3) {
    print(count);
    count++;
}
```

For-Each Loop (Iterating Lists)

```
var fruits = ["Apple", "Banana", "Cherry"];
for (var fruit in fruits) {
    print(fruit);
}
```

Switch-Case

```
String grade = 'B';
switch (grade) {
    case 'A':
        print("Excellent!");
        break;
    case 'B':
        print("Good!");
}
```

```
    break;
  default:
    print("Unknown grade");
}
```

Functions

Basic Function

```
int add(int a, int b) {
  return a + b;
}

void main() {
  print(add(5, 3)); // Output: 8
}
```

Arrow Syntax (Short Functions)

```
int multiply(int a, int b) => a * b;
```

Optional & Named Parameters

```
void greet(String name, {String? title}) {
  print("Hello, ${title ?? ""} $name");
}

void main() {
  greet("Alice");           // "Hello, Alice"
  greet("Bob", title: "Mr."); // "Hello, Mr. Bob"
}
```

Classes & Objects (OOP in Dart)

Defining a Class

```
class Person {
  String name;
```

```

int age;

// Constructor (short syntax)
Person(this.name, this.age);

// Method
void introduce() {
    print("Hi, I'm $name and I'm $age years old.");
}
}

void main() {
    var person = Person("Alice", 25);
    person.introduce();
}

```

Inheritance (Extending a Class)

```

class Student extends Person {
    String university;

    Student(String name, int age, this.university) : super(name, age);

    @override
    void introduce() {
        print("I'm $name, a student at $university.");
    }
}

```

Null Safety (Avoiding Null Errors)

Dart enforces null safety to prevent crashes.

```
String? nullableName = null; // Can be null
```

```
String name = "Alice";    // Must NOT be null
```

```
void main() {  
  print(nullableName?.length); // Safe access (prints null)  
  print(name.length);         // Direct access (prints 5)  
}
```

Asynchronous Programming (Future & Async/Await)

Used for network calls, file I/O, delays.

```
Future<String> fetchData() async {  
  await Future.delayed(Duration(seconds: 2)); // Simulate delay  
  return "Data loaded!";  
}  
  
void main() async {  
  print("Loading...");  
  var data = await fetchData();  
  print(data); // "Data loaded!" after 2 seconds  
}
```

Isolates in Dart

Dart uses a single-threaded event loop model. To prevent UI jank we need to offload expensive computations using isolates.

```
import 'dart:isolate';  
  
void main() async {  
  final receivePort = ReceivePort();  
  await Isolate.spawn(isolateEntry, receivePort.sendPort);  
  final sendPort = await receivePort.first as SendPort;  
  
  final responsePort = ReceivePort();
```

```

sendPort.send([10, responsePort.sendPort]);

print(await responsePort.first);
}

void isolateEntry(SendPort sendPort) {
  final port = ReceivePort();
  sendPort.send(port.sendPort);

  port.listen((msg) {
    int value = msg[0];
    SendPort reply = msg[1];
    reply.send(value * 2);
  });
}

```

This setup allows two-way communication between the main isolate and the worker isolate.

4.4 Widgets

Widgets are the building blocks of every Flutter application [17]. Everything in Flutter is a widget — from layout models like rows and columns to UI controls like text and buttons.

Key characteristics:

- Widgets are **immutable**: When UI changes, Flutter creates a new widget tree.
- Widgets describe **what to show**, not **how to show it** — rendering is handled by the Flutter engine.

There are two main types:

- **StatelessWidget**: Doesn't maintain state.
- **StatefulWidget**: Maintains internal state that may change over time.

Here are the examples of build-in widgets:

Text

```
Text('Hello, Flutter!', style: TextStyle(fontSize: 24))
```

Image

```
Image.asset('assets/logo.png')
```

Icon

```
Icon(Icons.favorite, color: Colors.red)
```

Button

```
ElevatedButton(  
  onPressed: () => print('Pressed'),  
  child: Text('Click me'),  
)
```

Flutter has a rich set of layout widgets for positioning children like:

Row & Column

```
Row(  
  children: [  
    Icon(Icons.star),  
    Text('Rating: 4.5'),  
  ],  
)
```

```
Column(  
  children: [  
    Text('Header'),  
    Text('Subheader'),  
  ],  
)
```

Container

```
Container(  
  padding: EdgeInsets.all(16),
```

```
margin: EdgeInsets.all(8),  
color: Colors.blueAccent,  
child: Text('Boxed Text'),  
)
```

Stack

```
Stack(  
  children: [  
    Image.asset('background.png'),  
    Positioned(  
      bottom: 10,  
      right: 10,  
      child: Text('Overlay Text'),  
    ),  
  ],  
)
```

Input & Form Widgets

```
TextField(  
  decoration: InputDecoration(labelText: 'Enter name'),  
)
```

```
Checkbox(  
  value: true,  
  onChanged: (bool? newValue) {},  
)
```

4.5 Stateful and Stateless Widgets

Flutter apps are made of widgets, but not all widgets behave the same way [18]. Some remain constant once built, and others can change in response to user actions or data updates.

A `StatelessWidget` is a widget that **never changes** once it's built. It's **static** and ideal for UI parts that don't need to update — like a label, logo, or static layout:

```
class Greeting extends StatelessWidget {  
  final String name;  
  
  Greeting({required this.name});  
  
  @override  
  Widget build(BuildContext context) {  
    return Text('Hello, $name!');  
  }  
}
```

The widget shows a greeting using the `name`. It will only change if the parent widget rebuilds and gives it a different name.

A `StatefulWidget` is a widget that **can change** over time — when the user interacts with it or when data updates.

A `StatefulWidget` is split into two parts:

- The **StatefulWidget class**: This is the immutable configuration.
- The **State class**: This is mutable and holds the internal logic and state.

```
class MyWidget extends StatefulWidget {  
  @override  
  _MyWidgetState createState() => _MyWidgetState();  
}  
  
class _MyWidgetState extends State<MyWidget> {  
  // This is where your state and logic lives  
}
```

As you can see in the example the `StatefulWidget` itself doesn't hold state. The state lives in the **State object**.

4.5.1 Widget Lifecycle

Here are the most important methods and when they're used:

- **createState()**. Called once when the widget is inserted into the widget tree. Returns a new State object.
- **initState()**. Called once when the state is initialized. We use it for initializing variables, starting timers and fetching data from a database or API. It's important not to use BuildContext here to interact with other widgets.
- **build()**. Called every time Flutter needs to redraw the widget. Should return the widget tree based on the current state.
- **setState()**. Tells Flutter that something changed. Triggers a rebuild of the widget by calling build() again:

```
setState() {  
  _counter++;  
};
```

- **dispose()**. Called when the widget is permanently removed from the tree. Used to clean up controllers or cancel timers or listeners.

Let's consider the Interactive Counter example:

```
class CounterApp extends StatefulWidget {  
  @override  
  _CounterAppState createState() => _CounterAppState();  
}  
  
class _CounterAppState extends State<CounterApp> {  
  int _counter = 0;  
  void _increment() {  
    setState() {  
      _counter++;  
    };  
  }  
  @override  
  void initState() {  
    super.initState();
```

```

    print('State created');
  }
  @override
  void dispose() {
    print('State destroyed');
    super.dispose();
  }
  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        Text('Counter: $_counter', style: TextStyle(fontSize: 24)),
        ElevatedButton(onPressed: _increment, child: Text('Increment')),
      ],
    );
  }
}

```

Let's walk through what happens behind the scenes when a StatefulWidget like our CounterApp is used in a Flutter app:

1. **createState() is called**

Flutter first calls the createState() method of the StatefulWidget. This creates an instance of the associated State class — in this case, _CounterAppState.

2. **initState() is executed**

Once the State object is created, Flutter immediately calls its initState() method. This is where you typically perform one-time setup, like initializing variables or starting a timer. In our example, it simply prints a message.

3. **build() is called for the first time**

Flutter then calls the build() method to construct the UI. At this point, _counter is 0, so the widget displays the message: Counter: 0.

4. **User interacts with the widget**

When the user taps the "Increment" button, the increment() method is triggered.

5. **setState()** is called

Inside `_increment()`, `setState()` is used to update the `_counter` variable by increasing its value. This tells Flutter that the state has changed and a rebuild is needed.

6. **build()** is called again

Flutter calls the `build()` method again — this time, it reflects the updated value of `_counter`. Now the UI shows: Counter: 1.

This flow illustrates how the **state drives the UI** in Flutter: whenever you call `setState()`, Flutter re-runs the `build()` method to reflect the new state.

4.6 Flutter State Management. Riverpod

In Flutter development, managing the application's state becomes more complex as the app grows. While `StatefulWidget` and `setState()` are great for simple, local updates, they quickly become limiting when multiple parts of your app need access to shared state, or when you want to separate logic from UI. This is where **Riverpod** becomes a powerful and modern solution [19]. It improves upon older patterns like `Provider` by offering more flexibility, better performance, and compile-time safety. With Riverpod, your state lives outside the UI, making it easier to test, reuse, and scale.

Riverpod provides a way to define and manage state in your application using **providers**. A provider is a unit that holds a piece of data or exposes logic. Widgets that need access to this data can subscribe to it and automatically rebuild when the data changes.

In Riverpod, widgets do not manage their own state directly. Instead, they use a `ref` object to **watch** or **read** providers. Watching a provider means the widget listens for changes and rebuilds when necessary. Reading a provider simply fetches the current value without subscribing to updates.

Riverpod supports different types of providers for different use cases. For simple values, `StateProvider` works well. For more complex logic, you can use `Notifier` or `AsyncNotifier`.

Let's look at a simple counter app written with Riverpod:

```
final counterProvider = StateProvider<int>((ref) => 0);
```

```
class CounterScreen extends ConsumerWidget {  
  @override
```

```

Widget build(BuildContext context, WidgetRef ref) {
  final counter = ref.watch(counterProvider);

  return Column(
    children: [
      Text('Counter: $counter'),
      ElevatedButton(
        onPressed: () {
          ref.read(counterProvider.notifier).state++;
        },
        child: Text('Increment'),
      ),
    ],
  );
}

```

Here's what happens behind the scenes:

1. A `StateProvider<int>` is declared, initialized with the value 0.
2. The `ConsumerWidget` uses `ref.watch(counterProvider)` to obtain the current counter value and listen for future changes.
3. The `build()` method runs and displays the current counter value.
4. When the user taps the button, `ref.read(counterProvider.notifier).state++` is executed to increment the counter.
5. Riverpod detects that the state has changed and automatically triggers a rebuild of the widget.
6. The updated counter value is displayed.

This approach cleanly separates state from the UI, making the code easier to manage and test.

4.7 Streams in Flutter

In many Flutter apps, we need to handle **asynchronous events** that occur over time — such as user input, data from an API, sensor updates, or real-time messages. While Future is useful for a **single** async result, **Streams** are used when you want to **listen to a continuous flow of data**.

Flutter supports Streams as a core part of the Dart language, and they are commonly used for handling real-time updates, event-driven data, or long-lived asynchronous sources.

A **Stream** is a sequence of asynchronous events. Unlike a Future that returns a single value, a Stream can emit multiple values over time. It might send values at any interval — immediately, after some delay, or periodically.

We can think of a Stream as a **pipeline** through which data flows from a source to one or more listeners. When something new happens — like a button press, data change, or incoming message — the stream emits an event, and the listener reacts to it.

Streams can emit:

- **Data events** – regular values
- **Error events** – if something goes wrong
- **Done event** – when the stream closes

To receive updates from a stream, the program **listens** to it. Dart provides several ways to do this, but in Flutter the most common approach in UI is using a **StreamBuilder** widget [20].

Here's a basic example that emits numbers every second:

```
Stream<int> counterStream() async* {  
  int i = 0;  
  while (true) {  
    await Future.delayed(Duration(seconds: 1));  
    yield i++;  
  }  
}
```

And we can use this stream in your widget like this:

```
class CounterStreamWidget extends StatelessWidget {
```

```

@override
Widget build(BuildContext context) {
  return StreamBuilder<int>(
    stream: counterStream(),
    builder: (context, snapshot) {
      if (snapshot.connectionState == ConnectionState.waiting) {
        return Text('Waiting...');
      } else if (snapshot.hasError) {
        return Text('Error: ${snapshot.error}');
      } else if (snapshot.hasData) {
        return Text('Counter: ${snapshot.data}');
      } else {
        return Text('No data');
      }
    },
  );
}

```

Let's break down what happens when this widget runs:

1. The `counterStream()` function starts running and returns a Stream that emits integers every second.
2. The `StreamBuilder` starts listening to the stream.
3. Initially, `snapshot.connectionState` is waiting, so the UI shows: Waiting....
4. After one second, the stream emits the first value: 0.
5. The `StreamBuilder` rebuilds and displays: Counter: 0.
6. Every second, a new value is emitted (1, 2, 3, ...).
7. The widget updates each time with the latest value.

Streams are particularly useful in Flutter applications that require handling real-time or continuous data [21]. Common use cases include real-time chat applications where new messages arrive dynamically, tracking audio or video playback positions

that update frequently, and integrating with Firebase Firestore to receive live updates from a database. Streams are also used to capture sensor input, such as data from an accelerometer, or to handle periodic data updates like stock prices or weather information.

References

1. Nielsen, Jakob. *Usability Engineering*. Morgan Kaufmann, 1994.
2. Norman, Don. *The Design of Everyday Things*. Revised and Expanded Edition, MIT Press, 2013.
3. Freeman, Eric, et al. *Head First Design Patterns*. 2nd ed., O'Reilly Media, 2021.
4. "React – A JavaScript Library for Building User Interfaces." ReactJS, <https://reactjs.org>. Accessed 28 May 2025.
5. "Hooks API Reference – React." ReactJS, <https://reactjs.org/docs/hooks-reference.html>. Accessed 28 May 2025.
6. "Redux – A Predictable State Container for JS Apps." Redux, <https://redux.js.org>. Accessed 28 May 2025.
7. "Redux Toolkit – The Official, Recommended Way to Write Redux Logic." Redux Toolkit, <https://redux-toolkit.js.org>. Accessed 28 May 2025.
8. "Android Developers Guide." Android Developers, <https://developer.android.com/guide>. Accessed 28 May 2025.
9. "Jetpack Compose – Modern UI Toolkit." Android Developers, <https://developer.android.com/jetpack/compose>. Accessed 28 May 2025.
10. "Compose Layouts." Android Developers, <https://developer.android.com/jetpack/compose/layouts>. Accessed 28 May 2025.
11. "Compose State and Side Effects." Android Developers, <https://developer.android.com/jetpack/compose/state>. Accessed 28 May 2025.
12. "Jetpack ViewModel." Android Developers, <https://developer.android.com/topic/libraries/architecture/viewmodel>. Accessed 28 May 2025.
13. Sullivan, Alex. *Pragmatic Jetpack Compose*. <https://www.jetpackcompose.app>. Accessed 28 May 2025.
14. Smyth, Neil. *Jetpack Compose 1.7 Essentials*. PragProg, 2024, 702pp.
15. "Flutter – Build Apps for Any Screen." Flutter, <https://flutter.dev>. Accessed 28 May 2025.

28 May 2025.

16.“Dart Language Tour.” Dart Programming Language,
<https://dart.dev/guides/language/language-tour>. Accessed 28 May 2025.

17.Windmill, Eric. *Flutter in Action*. Manning Publications, 2020.

18.“Managing State in Flutter Apps.” Google I/O 2019,
https://www.youtube.com/watch?v=d_m5csmrf7I. Accessed 28 May 2025.

19.“Riverpod – Simple, Robust State Management.” Riverpod,
<https://riverpod.dev>. Accessed 28 May 2025.

20.“StreamBuilder Class.” Flutter API Reference,
<https://api.flutter.dev/flutter/widgets/StreamBuilder-class.html>. Accessed 28 May 2025.

21.“Dart Asynchronous Programming: Streams.” Dart,
<https://dart.dev/tutorials/language/streams>. Accessed 28 May 2025.