React Fundamentals:

A Beginner's Guide to JavaScript, TypeScript, HTML, and CSS

Johan Corrales

React Fundamentals: A Beginner's Guide to JavaScript, TypeScript, HTML, and CSS

Johan Yesid Corrales López

2024

React Fundamentals: A Beginner's Guide to JavaScript, TypeScript, HTML, and CSS

Index

- Introduction to JavaScript
- Variables, Data Types, and Operators
- Functions and Control Flow (if, else, loops)
- DOM Manipulation: Making Your Webpage Interactive
- Events and User Interaction
- Practical Exercises: Adding Interactivity to Your Webpage

6. **Chapter 5: Getting Started with React**
   - Introduction to React and Component-Based Architecture
   - Setting Up Your First React App (with Vite/Next.js)
   - Understanding JSX: Writing HTML in JavaScript
   - React Components: Creating and Reusing Components
   - Exercise: Your First React App
   - Versions: Setting Up with Vite, Next.js (Pages Router), Next.js (App Router)
7. **Chapter 6: Working with React Components**
   - Function Components and JSX
   - Props: Passing Data to Components
   - State: Managing Component Data
   - Event Handling in React
   - Exercise: Building Reusable Components
8. **Chapter 7: Building Interactive UIs with React**
   - Conditional Rendering in React
   - Lists and Keys
   - Handling Forms in React
   - Controlled vs Uncontrolled Components
   - Exercise: Creating a Dynamic To-Do List
9. **Chapter 8: React Router (Optional)**
   - Introduction to Client-Side Routing
   - Setting Up React Router (for Vite)
   - Nested Routes and Route Parameters
   - Exercise: Adding Navigation to Your React App
10. **Chapter 9: Introduction to State Management**
    - React's useState Hook
    - Lifting State Up: Sharing Data Between Components
    - Context API: Managing Global State in React
    - Exercise: Building a Simple Shopping Cart
11. **Chapter 10: Working with TypeScript**
    - Introduction to TypeScript
    - Adding TypeScript to a JavaScript Project
    - Type Annotations and Interfaces
    - Type Inference and Generics
    - Using TypeScript with React
    - Practical Exercises: Adding Type Safety to React Components
12. **Chapter 11: Styling in React**
    - Styling with CSS Modules

Introduction

Why Learn React?

In today's fast-paced world of web development, React has become one of the most popular JavaScript libraries for building dynamic and interactive user interfaces. Whether you're browsing your favorite social media app, shopping online, or using productivity tools, chances are high that many of the websites you interact with are powered by React.

React simplifies the process of building complex UIs by allowing you to break them down into small, reusable components. Its declarative approach makes it easier to think about how your UI should look and behave at any given point in time, while its vast ecosystem provides countless tools and libraries to enhance your development experience. Most importantly, learning React opens the door to numerous career opportunities in front-end development.

What You Will Learn

This book is designed to take you on a journey from the basics of web development to building modern web applications with React and TypeScript. Whether you have little to no coding experience or are familiar with some web technologies but want to dive into React, this book is your starting point.

By the end of this book, you'll have a solid foundation in:

- **HTML:** Building the basic structure of a web page.
- **CSS:** Styling your web page to make it visually appealing and responsive.
- **JavaScript:** Adding interactivity to your site, making it more dynamic.
- **TypeScript:** Introducing you to type-safe coding, helping prevent common JavaScript errors.
- **React:** Building modern, component-based applications, managing state, handling user input, and fetching data from external sources.

Our focus is on providing practical examples that you can follow along with, allowing you to build confidence as you apply what you've learned step by step.

Why We Include TypeScript

TypeScript has become an essential tool in modern JavaScript development, especially in React applications. While JavaScript is flexible, it can sometimes lead to unexpected errors, which can be difficult to debug. TypeScript addresses this by adding static types, helping you catch bugs before they even make it into your codebase. It also improves the developer experience with powerful autocompletion and code hints, which make it easier to navigate larger applications.

By learning TypeScript alongside React, you'll gain an extra layer of confidence, knowing that your code is more robust and maintainable. Even if you're new to programming, TypeScript is a skill worth investing in early, as it will save you time and headaches as your projects grow in complexity.

How This Book is Structured

The book is divided into three main sections, each building on the previous one:

1. **Foundations** – We'll start with the fundamentals of HTML, CSS, and JavaScript. These are the building blocks of every web page. If you're new to these concepts, don't worry! We'll break them down into easy-to-understand lessons with hands-on exercises.

2. **React Basics** – Once you're comfortable with the web basics, we'll dive into React, exploring its component-based architecture, JSX, and how to manage state and props. You'll also learn how to set up your first React application using Vite or Next.js, depending on your preference.

3. **React with TypeScript** – Finally, we'll combine the power of React with TypeScript. You'll learn how to build strongly-typed components, manage state with confidence, and use advanced TypeScript features to create scalable, maintainable applications.

Each chapter includes small, practical examples, along with exercises that will guide you through creating your own mini-projects.

Who This Book is For

This book is for **absolute beginners** who want to take their first steps into web development, as well as those who have some experience but are new to React and TypeScript. If you're curious about how modern websites are built or want to start your career as a front-end developer, this guide will help you build a strong foundation and get hands-on experience.

By the end of this book, you'll not only understand how web pages are created but also how modern, professional-grade applications are built using React and TypeScript.

Setting Up Your Development Environment

To get started with React development, you'll need to set up your environment properly. This includes installing the tools necessary to write, run, and debug your code effectively.

1. Install Node.js and npm Node.js is a JavaScript runtime that allows you to run JavaScript on your computer, while npm (Node Package Manager) is used to install the libraries and dependencies you'll need. You can download Node.js (which includes npm) from https://nodejs.org/.

2. Choose a Code Editor A good code editor will make your development process easier. We recommend using **Visual Studio Code (VS Code)**, as it provides excellent support for JavaScript, TypeScript, and React. It also offers features like syntax highlighting, debugging, and useful extensions.

Download VS Code from https://code.visualstudio.com/.

3. Install Browser Developer Tools Most modern web browsers, like **Google Chrome** and **Firefox**, have built-in developer tools that make debugging your code easier. You can inspect your HTML, modify styles, and see errors in your JavaScript. Make sure you are familiar with these tools as you'll be using them frequently.

4. Set Up Git (Optional) **Git** is a version control system that helps you keep track of changes in your code. It's especially helpful when working on larger projects or collaborating with others. You can download Git from https://git-scm.com/.

Once you've installed these tools, you'll be ready to start coding along with the book!

Getting Started

Let's embark on this learning journey together! Make sure to set up your development environment following the first chapter, as we'll be coding along the way.

Don't hesitate to experiment, play around with the code, and break things—learning is all about discovery.

Chapter 1: Getting Started with Web Development

The Basics of Web Technologies: HTML, CSS, and JavaScript

Before we dive into building dynamic web applications with React, it's essential to understand the foundation of the web: HTML, CSS, and JavaScript. These three technologies form the backbone of every website you visit.

- **HTML (Hypertext Markup Language)** defines the structure of a web page. Think of it as the skeleton that holds everything together. With HTML, you define headings, paragraphs, links, images, and more.
- **CSS (Cascading Style Sheets)** is what makes your web page look good. It's responsible for styling the content—choosing fonts, colors, layout, and how things are positioned on the screen. Without CSS, all web pages would look plain and unformatted.
- **JavaScript** adds interactivity to your web pages. If you want users to interact with buttons, forms, or menus, JavaScript is what makes that possible. It allows you to control what happens when a user clicks a button, submits a form, or scrolls through a page.

Together, these three technologies create a complete web experience—HTML provides the structure, CSS provides the styling, and JavaScript brings it to life with interactivity.

What is a Web Browser and How Does It Work?

To fully understand how web development works, it's helpful to understand what happens behind the scenes when you visit a website. A **web browser** (like Chrome, Firefox, or Safari) is an application that retrieves and displays web content. Here's how it works:

1. **You enter a URL** (such as `www.example.com`) into your browser.
2. **The browser sends a request** to a server, asking for the resources (HTML, CSS, JavaScript, images) needed to display the page.
3. **The server responds** with the resources, and the browser processes them.
4. **The browser renders the webpage**—HTML is used to structure the content, CSS styles the content, and JavaScript controls any interactivity.

Each time you refresh a page or visit a new one, the browser repeats this process. As web developers, our job is to write the HTML, CSS, and JavaScript that the browser uses to display the content and behavior we want.

Your First Webpage

Let's start by building a simple webpage to get hands-on experience with HTML, CSS, and JavaScript. You can follow along by using a text editor like **VS Code**, **Sublime Text**, or even a basic text editor like Notepad.

1. **Create an HTML File**

Open your text editor and create a new file called `index.html`. This file will contain the structure of your webpage.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>My First Webpage</title>
    <link rel="stylesheet" href="styles.css" />
  </head>
  <body>
    <h1>Welcome to My First Webpage!</h1>
    <p>This is a simple webpage created using HTML.</p>
    <button id="myButton">Click me!</button>

    <script src="script.js"></script>
  </body>
</html>
```

This simple file sets up the structure of your webpage:

- The `<head>` section includes meta information, like the character set and viewport settings, and links to external files like CSS.
- The `<body>` section is where your visible content goes. Here, we have a heading (`<h1>`), a paragraph (`<p>`), and a button (`<button>`).
- At the bottom, we link to a JavaScript file (`script.js`) that will handle the interactivity.

2. **Create a CSS File**

Next, create a file called `styles.css` in the same folder. This file will contain the styles for your webpage.

```
body {
  font-family: Arial, sans-serif;
  background-color: #f0f0f0;
  color: #333;
  text-align: center;
  padding: 50px;
}

```

```css
h1 {
  color: #007bff;
}

button {
  background-color: #007bff;
  color: white;
  border: none;
  padding: 10px 20px;
  font-size: 16px;
  cursor: pointer;
}

button:hover {
  background-color: #0056b3;
}
```

This CSS will:

- Style the body with a clean font and centered layout.
- Give the heading (`<h1>`) a blue color.
- Style the button with a blue background and change its color on hover.

3. **Create a JavaScript File**

Now, create a file called `script.js` to add interactivity to your page.

```javascript
document.getElementById("myButton").addEventListener("click", function () {
  alert("Button clicked!");
});
```

This JavaScript code listens for a click event on the button. When the user clicks the button, an alert message will pop up.

Understanding the Structure

Here's a breakdown of what we've created:

- **HTML (index.html):** This is the backbone of your webpage, organizing your content into elements like headings, paragraphs, and buttons.
- **CSS (styles.css):** This file controls how your webpage looks, setting the layout, colors, and fonts.
- **JavaScript (script.js):** This file adds interactivity, responding to user actions like button clicks.

You now have a simple, complete webpage! You can open `index.html` in your browser to see it in action.

What's Next?

In this chapter, we introduced the building blocks of web development—HTML, CSS, and JavaScript. You've also created your very first webpage! As you move forward, you'll expand on these skills, learning how to build more complex and interactive websites.

In the next chapter, we'll dive deeper into **HTML**, exploring its structure and elements in more detail. This will help you create more sophisticated layouts and better organize your content.

Chapter 2: HTML Fundamentals

In this chapter, we will take a deeper dive into **HTML**, the language used to structure every web page. HTML provides the foundation for all web content, defining the layout, text, images, and links that make up a website. Understanding how to use HTML effectively is crucial to creating web pages that are accessible and well-structured.

Basic Structure of an HTML Document

Every HTML document follows a basic structure that helps the browser understand how to render the content. Let's break down the structure:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Your Page Title</title>
  </head>
  <body>
    <h1>Hello, world!</h1>
    <p>Welcome to your first HTML page.</p>
  </body>
</html>
```

- **`<!DOCTYPE html>`**: Declares the document type and version of HTML. This helps the browser render the page correctly.
- **`<html>`**: The root element that contains all other HTML elements.
- **`<head>`**: Contains meta-information about the document, such as the character set and title.
- **`<title>`**: Sets the page's title, which is displayed in the browser's title bar or tab.
- **`<body>`**: Everything inside the `<body>` tag is visible on the webpage. This is where you put your content, like text, images, and buttons.

Common HTML Tags

HTML offers a variety of tags (also known as elements) that you can use to build a web page. Here are some of the most commonly used tags:

1. **Headings**:
   - HTML provides six levels of headings, from `<h1>` to `<h6>`, with `<h1>` being the most important and `<h6>` the least.
   - Example:

```
<h1>Main Heading</h1>
<h2>Subheading</h2>
<h3>Smaller Subheading</h3>
```

2. **Paragraphs**:
   - Use the `<p>` tag for text content.
   - Example:

```
<p>This is a paragraph of text.</p>
```

3. **Links**:
   - Links are created using the `<a>` tag. The `href` attribute specifies the URL.
   - Example:

```
<a href="https://www.example.com">Visit Example.com</a>
```

4. **Images**:
   - Images are added using the `<img>` tag. The `src` attribute specifies the image source, and `alt` provides alternative text.
   - Example:

```
<img src="image.jpg" alt="A description of the image" />
```

5. **Lists**:
   - HTML supports both ordered (`<ol>`) and unordered (`<ul>`) lists.
   - **Unordered List** (bulleted):

```
<ul>
  <li>Item 1</li>
  <li>Item 2</li>
</ul>
```

   - **Ordered List** (numbered):

```
<ol>
  <li>Item 1</li>
  <li>Item 2</li>
</ol>
```

Organizing Content with Lists and Tables

HTML provides tools like **lists** and **tables** to help you organize content effectively.

Lists Lists are great for displaying information in a clear and organized way. You can use ordered lists (`<ol>`) for items that have a specific sequence, like steps in a recipe, and unordered lists (`<ul>`) for items without a specific order, like a list of ingredients.

Tables Tables allow you to display data in rows and columns, making it easy to present information in an organized format.

- Example of a Simple Table:

```
<table>
  <tr>
    <th>Name</th>
    <th>Age</th>
  </tr>
  <tr>
    <td>John Doe</td>
    <td>30</td>
  </tr>
  <tr>
    <td>Jane Smith</td>
    <td>25</td>
  </tr>
</table>
```

- **`<table>`**: Defines the table.
- **`<tr>`**: Defines a table row.
- **`<th>`**: Defines a header cell (usually bold and centered).
- **`<td>`**: Defines a standard cell.

HTML Forms and Input Elements

Forms are used to collect user input, and they are a key part of many websites. Forms contain various **input elements** like text fields, radio buttons, checkboxes, and submit buttons.

- Example of a Simple Form:

```
<form action="/submit" method="POST">
  <label for="name">Name:</label>
  <input type="text" id="name" name="name" />

  <label for="email">Email:</label>
  <input type="email" id="email" name="email" />

  <input type="submit" value="Submit" />
</form>
```

- `<form>`: Wraps the form elements and specifies the action (where the data should be sent) and the method (how it should be sent—`POST` or `GET`).
- `<input>`: Represents various form fields like text input, email input, and buttons.
- `<label>`: Defines labels for form fields, improving accessibility.

Practical Exercises: Building Simple Webpages

Now that you know the basic HTML tags, let's practice! In this exercise, you'll create a basic profile page with a heading, paragraph, an image, and a list of hobbies.

1. **Create a New HTML File**

   Create a new file called `profile.html` and add the following content:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.
    <title>My Profile</title>
  </head>
  <body>
    <h1>My Profile</h1>
    <p>Hello! My name is [Your Name]. I am learning web development, an

    <img src="profile-picture.jpg" alt="My profile picture" width="200"

    <h2>My Hobbies</h2>
    <ul>
      <li>Reading</li>
      <li>Cooking</li>
      <li>Traveling</li>
      <li>Coding</li>
    </ul>
  </body>
</html>
```

2. **Add Your Own Content**

   - Replace `[Your Name]` with your actual name.
   - Replace `"profile-picture.jpg"` with the path to an image of yourself (or any placeholder image).

3. **Test Your Page**

   Save the file and open it in your web browser to see your profile page in action.

What's Next?

In this chapter, you learned about the fundamentals of HTML, including the basic structure of an HTML document, common HTML tags, and how to organize content using lists, tables, and forms. You also built a simple profile page to practice using these elements.

In the next chapter, we will learn about **CSS** and how to style the webpage you created, making it visually appealing and well-designed.

Chapter 3: CSS Fundamentals

In this chapter, we will learn about **CSS (Cascading Style Sheets)**, the language used to style and lay out your web pages. CSS is what makes a website look visually appealing, and it's an essential skill for every web developer. You will learn how to add colors, style text, position elements, and make your pages responsive.

How CSS Works: Selectors, Properties, and Values

CSS works by selecting elements on the web page and applying specific styles to them. It follows a basic pattern of **selectors**, **properties**, and **values**:

- **Selectors**: Specify which HTML elements you want to style.
- **Properties**: Define the type of style you want to apply (e.g., color, font size).
- **Values**: Provide the specific value for the property.

Example:

```
h1 {
  color: blue;
  font-size: 24px;
}
```

- `h1`: The selector, targeting all `<h1>` elements.
- `color`: The property, specifying the color of the text.
- `blue`: The value, defining the color to be applied.

Styling Text, Colors, and Backgrounds

CSS gives you the ability to style text, add colors, and set backgrounds to make your webpage more attractive. Here are some common properties you can use:

- **Text Color**: You can set the color of text using the `color` property.

```
p {
  color: #333333;
}
```

- **Font Family**: You can change the font of your text using the `font-family` property.

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

- **Font Size**: You can adjust the size of the text using the `font-size` property.

```
h2 {
  font-size: 28px;
}
```

- **Text Alignment**: You can align text using the `text-align` property.

```
p {
  text-align: center;
}
```

- **Background Color**: You can set the background color of elements using the `background-color` property.

```
div {
  background-color: #f0f0f0;
}
```

Box Model: Margins, Padding, and Borders

The **box model** is a fundamental concept in CSS that determines how elements are displayed on a page. It consists of four parts:

1. **Content**: The inner part, where your text or image is displayed.
2. **Padding**: The space between the content and the border.
3. **Border**: The edge around the padding (or content if padding is not specified).
4. **Margin**: The space outside the border, separating the element from other elements.

Example:

```
div {
  padding: 20px;
  border: 2px solid #000;
  margin: 10px;
}
```

- `padding`: Adds space inside the element, between the content and the border.
- `border`: Adds a border around the element.
- `margin`: Adds space outside the element, pushing it away from surrounding elements.

Layout Basics: Flexbox and Grid

CSS provides powerful layout techniques to help you create complex and responsive designs. Two of the most popular layout methods are **Flexbox** and **Grid**.

Flexbox **Flexbox** is a one-dimensional layout model that helps you arrange items in a row or column. It's perfect for creating flexible and responsive layouts.

Example:

```
.container {
  display: flex;
  justify-content: space-between;
}
```

- `display: flex`: Turns the container into a flex container.
- `justify-content`: Controls the alignment of items along the main axis (e.g., space between).

Grid **CSS Grid** is a two-dimensional layout model that allows you to create complex layouts with rows and columns. It provides more control over both dimensions compared to Flexbox.

Example:

```
.grid-container {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
  gap: 20px;
}
```

- `display: grid`: Turns the container into a grid container.
- `grid-template-columns`: Defines the number of columns and their width (e.g., three equal columns).
- `gap`: Sets the spacing between grid items.

Responsive Design: Media Queries

Responsive design ensures that your website looks good on all devices, from desktop computers to mobile phones. You can use **media queries** to apply different styles based on the screen size.

Example:

```
@media (max-width: 768px) {
  body {
    font-size: 16px;
  }
}
```

- `@media`: Defines a media query.
- `max-width: 768px`: Applies the styles only if the screen width is 768 pixels or smaller.

Practical Exercises: Styling Your Webpage

Now that you know the basics of CSS, let's practice by adding styles to the profile page you created in Chapter 2.

1. **Create a New CSS File**

   Create a new file called `profile-styles.css` and link it to your `profile.html` file by adding the following line inside the `<head>` tag:

```
<link rel="stylesheet" href="profile-styles.css" />
```

2. **Add Styles**

   Add the following styles to `profile-styles.css` to make your profile page more visually appealing:

```
body {
  font-family: Arial, sans-serif;
  background-color: #f0f8ff;
  color: #333;
  text-align: center;
  padding: 20px;
}

h1 {
  color: #007bff;
}

img {
  border-radius: 50%;
  width: 150px;
  height: 150px;
}

ul {
  list-style-type: none;
  padding: 0;
}

li {
  background-color: #e0e0e0;
  margin: 5px;
  padding: 10px;
```

```
  border-radius: 5px;
}
```

3. **Test Your Page**

   Save the file and open `profile.html` in your browser to see the updated styles applied to your profile page.

What's Next?

In this chapter, you learned about the fundamentals of CSS, including how to style text, work with the box model, and create layouts using Flexbox and Grid. You also practiced adding styles to a basic profile page.

In the next chapter, we will dive into **JavaScript**, where you will learn how to make your web pages interactive by adding dynamic behavior.

Chapter 4: JavaScript Basics

JavaScript is a versatile programming language that adds interactivity to your web pages. It allows you to create dynamic content, respond to user events, and make your websites more engaging. In this chapter, we will cover the fundamentals of JavaScript, including variables, data types, functions, and how to manipulate the Document Object Model (DOM).

Introduction to JavaScript

JavaScript is a scripting language that runs directly in the web browser, making it a powerful tool for creating dynamic and interactive web applications. With JavaScript, you can change content, validate forms, create animations, and much more.

JavaScript code can be added to your HTML file in several ways:

- **Inline**: Using the `onclick` attribute within an HTML element.
- **Internal**: Placing JavaScript within a `<script>` tag inside your HTML file.
- **External**: Linking to an external JavaScript file.

Example of adding JavaScript internally:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>JavaScript Example</title>
  </head>
  <body>
```

```
    <h1>Welcome to JavaScript Basics!</h1>
    <button onclick="sayHello()">Click Me</button>
    <script>
      function sayHello() {
        alert("Hello, world!");
      }
    </script>
  </body>
</html>
```

Variables, Data Types, and Operators

Variables Variables are used to store information in JavaScript. You can create variables using the `let`, `const`, or `var` keywords.

- `let`: Declares a variable that can be reassigned.
- `const`: Declares a constant variable that cannot be reassigned.
- `var`: Declares a variable (older syntax, typically replaced by `let` or `const`).

Example:

```
let name = "Alice";
const age = 30;
var isStudent = true;
```

Data Types JavaScript has several data types, including:

- **String**: Textual data (e.g., `"Hello"`).
- **Number**: Numeric values (e.g., `42`).
- **Boolean**: Represents `true` or `false`.
- **Array**: A list of values (e.g., `[1, 2, 3]`).
- **Object**: A collection of key-value pairs (e.g., `{ name: "Alice", age: 30 }`).

Example:

```
let message = "Hello, world!";
let score = 100;
let isActive = true;
let colors = ["red", "green", "blue"];
let person = { name: "Alice", age: 30 };
```

Operators JavaScript provides various operators for performing operations:

- **Arithmetic Operators**: `+`, `-`, `*`, `/`, `%` (e.g., `let sum = 10 + 5;`).
- **Comparison Operators**: `==`, `===`, `!=`, `<`, `>`, `<=`, `>=` (e.g., `10 > 5` returns `true`).

- **Logical Operators**: `&&` (AND), `||` (OR), `!` (NOT) (e.g., `true && false` returns `false`).

Functions and Control Flow

Functions Functions are blocks of reusable code that can be called to perform a specific task. You can define a function using the `function` keyword.

Example:

```
function greet(name) {
  console.log("Hello, " + name + "!");
}

greet("Alice"); // Output: Hello, Alice!
```

Functions help you organize your code, reduce repetition, and improve readability.

Control Flow **Control flow** refers to the order in which code is executed. JavaScript has several control flow statements, including:

- **`if` Statements**: Used to execute code based on a condition.

```
let age = 18;
if (age >= 18) {
  console.log("You are an adult.");
} else {
  console.log("You are a minor.");
}
```

- **`for` Loops**: Used to repeat a block of code a specific number of times.

```
for (let i = 0; i < 5; i++) {
  console.log("Iteration: " + i);
}
```

- **`while` Loops**: Used to repeat a block of code while a condition is true.

```
let count = 0;
while (count < 3) {
  console.log("Count: " + count);
  count++;
}
```

DOM Manipulation: Making Your Webpage Interactive

The **Document Object Model (DOM)** represents the structure of a web page, allowing JavaScript to interact with and manipulate the content. You can use JavaScript to select, modify, and create HTML elements dynamically.

Selecting Elements You can select HTML elements using methods like `getElementById()`, `querySelector()`, and `getElementsByClassName()`.

Example:

```
let heading = document.getElementById("mainHeading");
let button = document.querySelector("button");
```

Modifying Elements Once you have selected an element, you can modify its content, style, or attributes.

Example:

```
heading.textContent = "Updated Heading!";
button.style.backgroundColor = "blue";
```

Creating New Elements You can also create new HTML elements and add them to the page.

Example:

```
let newParagraph = document.createElement("p");
newParagraph.textContent = "This is a new paragraph.";
document.body.appendChild(newParagraph);
```

Events and User Interaction

JavaScript allows you to respond to user actions, such as clicking a button, by using **event listeners**. An event listener listens for specific user actions and triggers a function when the event occurs.

Example:

```
let myButton = document.getElementById("myButton");
myButton.addEventListener("click", function () {
  alert("Button clicked!");
});
```

In this example, when the user clicks the button, an alert message will pop up.

Practical Exercises: Adding Interactivity to Your Webpage

Let's practice adding some JavaScript to make your profile page interactive. Follow these steps:

1. **Update Your HTML File**

   Add a button to your `profile.html` file:

```
<button id="greetButton">Greet Me</button>
<p id="greetMessage"></p>
```

2. **Create a JavaScript File**

   Create a new file called `profile-script.js` and link it to your `profile.html` file by adding the following line inside the `<head>` tag:

```
<script src="profile-script.js" defer></script>
```

3. **Add JavaScript**

   In `profile-script.js`, add the following code to make the button display a greeting message when clicked:

```
let greetButton = document.getElementById("greetButton");
let greetMessage = document.getElementById("greetMessage");

greetButton.addEventListener("click", function () {
  greetMessage.textContent = "Hello, welcome to my profile page!";
});
```

4. **Test Your Page**

   Save the file and open `profile.html` in your browser. Click the button to see the greeting message appear.

What's Next?

In this chapter, you learned about JavaScript fundamentals, including variables, data types, functions, and how to manipulate the DOM to add interactivity to your web pages. You also practiced adding JavaScript to your profile page to make it more interactive.

In the next chapter, we will begin working with **React**, where you will learn how to create reusable components and manage state in your applications.

Chapter 5: Getting Started with React

React is a powerful JavaScript library used to build dynamic and interactive user interfaces. Developed by Facebook, React allows developers to create reusable components, manage state efficiently, and build modern, scalable applications. In this chapter, we will cover the basics of React, including setting up your first React app, understanding JSX, and creating components.

Introduction to React and Component-Based Architecture

React is all about **components**. Components are independent, reusable pieces of code that represent parts of the user interface. You can think of components as building blocks that you use to build complex UIs.

For example, a webpage may be composed of a navigation bar, a header, a content section, and a footer. Each of these can be represented as a separate component, making it easier to manage and reuse code.

React's component-based architecture has several advantages:

- **Reusability**: Components can be reused throughout the application, reducing redundancy.
- **Modular Development**: Each component is self-contained, making it easier to understand and maintain.
- **Declarative UI**: React's declarative approach allows you to describe what the UI should look like based on the current state.

Setting Up Your First React App (with Vite/Next.js)

To start working with React, you first need to set up a React project. There are several ways to do this, and two popular tools are **Vite** and **Next.js**.

Option 1: Setting Up with Vite **Vite** is a modern build tool that offers fast and easy setup for React applications.

1. **Install Node.js**: First, ensure that you have Node.js installed on your computer. You can download it from https://nodejs.org/.
2. **Create a Vite Project**: Run the following command in your terminal to create a new React project with Vite:

   ```
   npm create vite@latest my-react-app --template react
   ```

   Replace `my-react-app` with your desired project name.
3. **Install Dependencies**: Navigate to your project directory and install the necessary dependencies:

   ```
   cd my-react-app
   npm install
   ```

4. **Start the Development Server**: Run the following command to start the development server:

   ```
   npm run dev
   ```

   Your React app will be available at `http://localhost:3000`.
5. **Folder Structure Overview**:
   - `public/`: Contains static assets like images.
   - `src/`: Contains your application code, including components, styles, and main app logic.
   - `index.html`: The main HTML file that serves your app.

Option 2: Setting Up with Next.js **Next.js** is a powerful React framework that provides server-side rendering, routing, and other features out of the box.

1. **Create a Next.js Project**: Run the following command in your terminal to create a new Next.js project:

   ```
   npx create-next-app@latest my-next-app
   ```

   Replace `my-next-app` with your desired project name.

2. **Navigate to the Project Directory**: Use the following command to navigate to your project folder:

   ```
   cd my-next-app
   ```

3. **Start the Development Server**: Run the following command to start the development server:

   ```
   npm run dev
   ```

   Your Next.js app will be available at `http://localhost:3000`.

4. **Folder Structure Overview**:

   - `pages/`: Contains your application's routes. Each file in this directory corresponds to a route (e.g., `index.js` is the home page).
   - `public/`: Contains static assets.
   - `components/`: A directory you can create to store reusable components.

Option 3: Setting Up with Next.js (App Router) The **App Router** is a new feature introduced in the latest versions of Next.js, providing more flexibility and improved routing capabilities for complex applications.

1. **Create a Next.js Project with App Router**: Use the same command as above to create a Next.js project:

   ```
   npx create-next-app@latest my-next-app
   ```

2. **Navigate to the Project Directory**:

   ```
   cd my-next-app
   ```

3. **Start the Development Server**:

   ```
   npm run dev
   ```

4. **Folder Structure Overview (App Router)**:

   - `app/`: Instead of the `pages/` directory, the new app router uses the `app/` directory. This allows for nested routing and layouts.
   - `public/`: Contains static assets.
   - `components/`: Stores reusable components that can be used across your application.

The **App Router** provides enhanced features like nested layouts, colocation of server and client components, and improved data fetching, making it ideal for large and scalable projects.

Understanding JSX: Writing HTML in JavaScript

JSX (JavaScript XML) is a syntax extension for JavaScript that allows you to write HTML-like code within your JavaScript. JSX is used in React to describe what the UI should look like.

Example of JSX:

```
function App() {
  return (
    <div>
      <h1>Welcome to My React App!</h1>
      <p>This is a simple React component.</p>
    </div>
  );
}
```

In the example above, the `App` function returns JSX that represents a `<div>` containing an `<h1>` and a `<p>`. JSX makes it easier to visualize the structure of your components.

React Components: Creating and Reusing Components

A **component** in React is simply a JavaScript function that returns JSX. Components can be defined in two ways:

1. **Function Components**: The simplest way to define a React component is as a function that returns JSX.

   Example:

   ```
   function Greeting() {
     return <h1>Hello, world!</h1>;
   }
   ```

2. **Class Components**: Components can also be defined using ES6 classes. While function components are more commonly used today, it's still helpful to understand class components.

   Example:

   ```
   class Greeting extends React.Component {
     render() {
       return <h1>Hello, world!</h1>;
     }
   }
   ```

Components can be reused throughout your application. For example, if you want to display the same greeting message in multiple places, you can reuse the `Greeting` component wherever it's needed.

Exercise: Your First React App

Let's create a simple React app with a reusable component:

1. **Create a New Component**

   - **Vite**: Inside your `src` directory, create a new file called `Greeting.jsx` and add the following code:

```jsx
function Greeting() {
  return <h1>Hello, welcome to my first React app!</h1>;
}

export default Greeting;
```

   - **Next.js (Pages Router)**: Inside your `components` directory, create a new file called `Greeting.jsx` and add the following code:

```jsx
function Greeting() {
  return <h1>Hello, welcome to my first React app!</h1>;
}

export default Greeting;
```

   - **Next.js (App Router)**: Inside your `components` directory, create a new file called `Greeting.jsx` and add the following code:

```jsx
function Greeting() {
  return <h1>Hello, welcome to my first React app!</h1>;
}

export default Greeting;
```

2. **Use the Component in Your App**

   - **Vite**: Open `App.jsx` in the `src` directory and import the `Greeting` component:

```jsx
import Greeting from "./Greeting";

function App() {
  return (
    <div>
      <Greeting />
    </div>
  );
```

```
}

export default App;
```

- **Next.js (Pages Router)**: Open `pages/index.js` and import the `Greeting` component:

```
import Greeting from "../components/Greeting";

export default function Home() {
  return (
    <div>
      <Greeting />
    </div>
  );
}
```

- **Next.js (App Router)**: Open `app/page.js` and import the `Greeting` component:

```
import Greeting from "../components/Greeting";

export default function Home() {
  return (
    <div>
      <Greeting />
    </div>
  );
}
```

3. **Run Your App**

   Save the changes, and check your browser. You should see the greeting message rendered on the screen.

Versions: Setting Up with Vite, Next.js (Pages Router), Next.js (App Router)

React applications can be set up in different environments depending on your use case. Below are three versions for setting up React:

1. **Vite**: Offers a simple and fast setup for React development, ideal for prototyping and small projects.
2. **Next.js (Pages Router)**: Provides server-side rendering and routing capabilities. The pages router structure is based on file-based routing where each file in the `pages` directory represents a route.
3. **Next.js (App Router)**: Introduced in newer versions of Next.js, the app

router offers more flexibility and improved routing capabilities for larger and more complex applications.

Each setup has its unique benefits, and you can choose based on the requirements of your project.

What's Next?

In this chapter, you learned how to set up a React app, understand JSX, and create reusable components. React's component-based architecture makes it easy to build complex UIs by breaking them down into manageable pieces.

In the next chapter, we will learn how to work with **React components** more deeply, including props, state, and event handling.

Chapter 6: Working with React Components

In the previous chapter, we learned how to set up a React app, understand JSX, and create basic React components. Now, we will dive deeper into working with React components, exploring how to pass data between components using props, manage component state, and handle events.

Function Components and JSX

React components are the building blocks of React applications, and they can be either **function components** or **class components**. Today, **function components** are the most commonly used approach because they are simpler, more concise, and have access to **React Hooks**.

Example of a simple function component:

```
function Welcome() {
  return <h1>Welcome to React Components!</h1>;
}
```

This function component takes no input and simply renders a heading element. Components can also accept input data, called **props**, which we will explore next.

Props: Passing Data to Components

Props (short for properties) are used to pass data from one component to another. Props make components more reusable, as you can customize their behavior by passing different values.

Example:

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

```
function App() {
  return (
    <div>
      <Greeting name="Alice" />
      <Greeting name="Bob" />
    </div>
  );
}
```

In the example above, the `Greeting` component accepts a `name` prop and displays a personalized greeting message. By passing different values (`name="Alice"` and `name="Bob"`), we can reuse the same component with different outputs.

Destructuring Props Instead of accessing props through `props.name`, you can **destructure** the props for cleaner code:

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

State: Managing Component Data

State is a way to manage data that changes over time. Unlike props, which are read-only, state is mutable, and a component can update its state to reflect changes in the user interface.

To manage state in function components, we use the **`useState`** hook.

Example:

```
import { useState } from "react";

function Counter() {
  const [count, setCount] = useState(0);

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

In this example:

- We import the `useState` hook from React.
- The `useState(0)` function returns an array with two values: the current state (`count`) and a function to update that state (`setCount`).

- The button's `onClick` event updates the count when clicked.

Event Handling in React

React allows you to handle user interactions, such as clicks, form submissions, and more, using **event handlers**. Event handlers are similar to those in plain JavaScript, but in React, they are written in **camelCase** and passed as props to elements.

Example:

```
function ButtonClick() {
  function handleClick() {
    alert("Button was clicked!");
  }

  return <button onClick={handleClick}>Click Me</button>;
}
```

In the example above, the `handleClick` function is triggered when the button is clicked, displaying an alert.

Passing Arguments to Event Handlers If you need to pass arguments to an event handler, you can use an inline arrow function:

```
function WelcomeButton() {
  function greetUser(name) {
    alert(`Hello, ${name}!`);
  }

  return <button onClick={() => greetUser("Alice")}>Greet</button>;
}
```

Exercise: Building Reusable Components

Let's create a simple app that includes multiple reusable components. We'll build a small **shopping list** that allows you to add and remove items.

1. **Create a ShoppingItem Component**

   Create a new file called `ShoppingItem.jsx` and add the following code:

```
function ShoppingItem({ item, onRemove }) {
  return (
    <div>
      <span>{item}</span>
      <button onClick={() => onRemove(item)}>Remove</button>
    </div>
  );
```

```
}

export default ShoppingItem;
```

The `ShoppingItem` component accepts `item` (the name of the item) and `onRemove` (a function to remove the item) as props.

2. **Use the ShoppingItem Component in Your App**

Open `App.jsx` and update it to use the `ShoppingItem` component:

```
import { useState } from "react";
import ShoppingItem from "./ShoppingItem";

function App() {
  const [items, setItems] = useState(["Milk", "Bread", "Eggs"]);
  const [newItem, setNewItem] = useState("");

  function addItem() {
    if (newItem) {
      setItems([...items, newItem]);
      setNewItem("");
    }
  }

  function removeItem(itemToRemove) {
    setItems(items.filter((item) => item !== itemToRemove));
  }

  return (
    <div>
      <h1>Shopping List</h1>
      <input
        type="text"
        value={newItem}
        onChange={(e) => setNewItem(e.target.value)}
        placeholder="Add a new item"
      />
      <button onClick={addItem}>Add Item</button>
      <div>
        {items.map((item) => (
          <ShoppingItem key={item} item={item} onRemove={removeItem} />
        ))}
      </div>
    </div>
  );
}
```

```
export default App;
```

3. **Test Your App**

   Save the changes and check your browser. You should be able to add new items to your shopping list and remove items using the `ShoppingItem` component.

What's Next?

In this chapter, you learned how to work with React components more deeply, including passing data using props, managing state with the `useState` hook, and handling events. You also built a simple shopping list app to practice using reusable components.

In the next chapter, we will explore how to build **interactive user interfaces** using conditional rendering, lists, and form handling in React.

Chapter 7: Building Interactive UIs with React

In the previous chapter, we explored how to pass data to components using props, manage state, and handle events. In this chapter, we will focus on building more interactive user interfaces using **conditional rendering**, **lists**, and **form handling** in React. These concepts will help you create dynamic and user-friendly web applications.

Conditional Rendering in React

Conditional rendering is used to display different content based on certain conditions. In React, you can use JavaScript's conditional statements, such as **if-else** statements or **ternary operators**, to control what is displayed.

Example of conditional rendering using an if-else statement:

```
function UserGreeting({ isLoggedIn }) {
  if (isLoggedIn) {
    return <h1>Welcome back!</h1>;
  } else {
    return <h1>Please sign in.</h1>;
  }
}
```

Example of conditional rendering using a ternary operator:

```
function UserGreeting({ isLoggedIn }) {
  return (
    <h1>{isLoggedIn ? "Welcome back!" : "Please sign in."}</h1>
```

```
  );
}
```

Conditional rendering can also be used to show or hide specific UI elements based on a condition. For example, you might show a loading spinner while data is being fetched:

```
function LoadingIndicator({ isLoading }) {
  return isLoading ? <p>Loading...</p> : <p>Data Loaded</p>;
}
```

Rendering Lists in React

In many applications, you'll need to display a list of items, such as a list of products or users. React makes it easy to render lists using JavaScript's **map()** method.

Example:

```
function ShoppingList({ items }) {
  return (
    <ul>
      {items.map((item, index) => (
        <li key={index}>{item}</li>
      ))}
    </ul>
  );
}
```

In the example above:

- `items.map()` is used to iterate over the list of items.
- The `key` prop is added to each list item to provide a unique identifier. This helps React keep track of which items have changed, been added, or removed.

Handling Forms in React

Forms are an essential part of many web applications, allowing users to submit data, such as signing up for an account or adding items to a list. In React, you can manage form data using **controlled components**.

A **controlled component** is a component where React controls the form data through state. This means that the form elements' values are stored in the component's state and updated via **event handlers**.

Example of a controlled form component:

```
import { useState } from "react";

function SignupForm() {
  const [username, setUsername] = useState("");
```

```
  const [email, setEmail] = useState("");

  function handleSubmit(event) {
    event.preventDefault();
    alert(`Username: ${username}, Email: ${email}`);
  }

  return (
    <form onSubmit={handleSubmit}>
      <div>
        <label>Username:</label>
        <input
          type="text"
          value={username}
          onChange={(e) => setUsername(e.target.value)}
        />
      </div>
      <div>
        <label>Email:</label>
        <input
          type="email"
          value={email}
          onChange={(e) => setEmail(e.target.value)}
        />
      </div>
      <button type="submit">Sign Up</button>
    </form>
  );
}
```

In the example above:

- **`useState`** is used to store the form data (e.g., **`username`** and **`email`**).
- **`onChange`** event handlers are used to update the state as the user types.
- **`handleSubmit`** prevents the default form submission and displays an alert with the form data.

Controlled vs Uncontrolled Components

- **Controlled Components**: Form elements whose values are controlled by React state. This approach makes it easy to read and modify form data and to enforce validation.
- **Uncontrolled Components**: Form elements where the data is handled by the DOM instead of React. You can use **refs** to access the values of uncontrolled components.

Example of an uncontrolled component using a ref:

```jsx
import { useRef } from "react";

function UncontrolledForm() {
  const inputRef = useRef(null);

  function handleSubmit(event) {
    event.preventDefault();
    alert(`Input Value: ${inputRef.current.value}`);
  }

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" ref={inputRef} />
      <button type="submit">Submit</button>
    </form>
  );
}
```

In this example, **useRef** is used to directly access the value of the input field when the form is submitted.

Exercise: Creating a Dynamic To-Do List

Let's put together what we've learned so far to create a simple, dynamic **To-Do List** application.

1. **Create a ToDoItem Component**

   Create a new file called `ToDoItem.jsx` and add the following code:

```jsx
function ToDoItem({ item, onRemove }) {
  return (
    <div>
      <span>{item}</span>
      <button onClick={() => onRemove(item)}>Remove</button>
    </div>
  );
}

export default ToDoItem;
```

   The `ToDoItem` component accepts `item` (the task) and `onRemove` (a function to remove the task) as props.

2. **Use the ToDoItem Component in Your App**

   Open `App.jsx` and update it to use the `ToDoItem` component:

```jsx
import { useState } from "react";
import ToDoItem from "./ToDoItem";

function App() {
  const [tasks, setTasks] = useState(["Learn React", "Build a project"]
  const [newTask, setNewTask] = useState("");

  function addTask() {
    if (newTask) {
      setTasks([...tasks, newTask]);
      setNewTask("");
    }
  }

  function removeTask(taskToRemove) {
    setTasks(tasks.filter((task) => task !== taskToRemove));
  }

  return (
    <div>
      <h1>To-Do List</h1>
      <input
        type="text"
        value={newTask}
        onChange={(e) => setNewTask(e.target.value)}
        placeholder="Add a new task"
      />
      <button onClick={addTask}>Add Task</button>
      <div>
        {tasks.map((task) => (
          <ToDoItem key={task} item={task} onRemove={removeTask} />
        ))}
      </div>
    </div>
  );
}

export default App;
```

3. **Test Your To-Do List**

   Save the changes and check your browser. You should be able to add new tasks to your to-do list and remove tasks using the `ToDoItem` component.

What's Next?

In this chapter, you learned how to build interactive UIs in React using conditional rendering, lists, and forms. You also built a dynamic To-Do List app to practice these concepts. These skills are essential for creating user-friendly applications that respond to user interactions.

In the next chapter, we will explore **React Router** and learn how to add navigation to your React applications.