Web Development with HTML, CSS, and JavaScript

Create dynamic, responsive websites with ease using HTML, CSS, and JavaScript



Kevin Wilson

bpb

Web Development with HTML, CSS, and JavaScript

Create dynamic, responsive websites with ease using HTML, CSS, and JavaScript



Kevin Wilson

bpb

Web Development with HTML, CSS, and JavaScript

Create dynamic, responsive websites with ease using HTML, CSS, and JavaScript

Kevin Wilson



www.bpbonline.com

First Edition 2026

Copyright © BPB Publications, India

ISBN: 978-93-65896-909

All Rights Reserved. No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true and correct to the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but the publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

To View Complete
BPB Publications Catalogue
Scan the QR Code:

www.bpbonline.com

About the Author

With an impressive tenure exceeding two decades in the computer industry, **Kevin Wilson** is a proven authority on computers and technology highlighted by extensive experience, and a master's degree in computer science software engineering.

His expertise seamlessly extends across a vast array of IT sectors from the artistic arenas of graphic and web design, digital film and photography to the intricate spheres of computer programming and software engineering. His skills also extend to the strategic domains of corporate network development, as well as building computer systems, and IT support.

In his capacity as the senior writer and director at Elluminet Press Ltd, Kevin has penned multiple tech-related books that have found favor among home computer users, students and professionals around the world. His contributions in the classroom as a computer science lecturer and IT trainer further demonstrate his knowledge and continue to help people understand computer technology.

Kevin's work is well respected among students in England, South Africa, Canada, and the United States. Beyond writing, he lends his technical expertise as a reviewer and consultant to many other computer publications and textbooks.

His philosophy, *If you can't explain something simply, then you haven't understood it well enough*, is the driving force behind his acclaimed exploring tech series, in which complex technical topics are broken into smaller, comprehensive steps that are easy to understand for both students, professionals, and the average computer user.

In addition to this, along with his team, Kevin has written and produced numerous video training courses using visual effects, computer graphics

and clear presentation covering a variety of subjects including computer hardware, Microsoft Office, and Python computer programming.

Preface

This comprehensive guide provides a complete introduction to web development, starting with the core technologies that power the internet. The book begins by defining web development and introducing the essential trio: HTML for structure, CSS for style, and JavaScript for interactivity. You'll set up a professional development environment using VS Code and explore key browser developer tools before building your very first web page.

The course then thoroughly explores HTML, covering elements, attributes, forms for user input, and incorporating multimedia like images and videos. The next major section focuses on CSS, detailing its syntax, the critical box model , and advanced layout techniques using flexbox and CSS grid. A dedicated chapter introduces the principles of responsive design using media queries to ensure your sites look great on any device. Finally, the book moves into client-side scripting with JavaScript, explaining variables, control flow, functions, and how to manipulate the **Document Object Model** (**DOM**) to create dynamic web experiences. The book concludes with a practical, hands-on chapter where you build a fully functional, responsive website from scratch, applying all the skills learned.

The book is divided into 13 chapters, providing a clear path from beginner concepts to practical website construction:

Chapter 1: Introduction to Web Development- This chapter introduces the core concepts of web development, starting with a definition and the essential technologies: HTML, CSS, and JavaScript.

Chapter 2: Setting up the Development Environment- In this chapter, you'll learn how to prepare your development environment for web projects. We'll start by installing **Visual Studio Code** (**VSCode**) as the primary code editor and exploring essential developer tools in modern web browsers.

Chapter 3: Creating Your First Web Page- In this chapter, you will learn the foundational steps to create a simple web page. Starting with how to set up an HTML file, we'll explore the basic structure of HTML, including essential elements like the doctype, head, and body tags.

Chapter 4: HTML Elements and Attributes- This chapter introduces the fundamental building blocks of HTML: tags and elements. Readers will learn how to use attributes to add additional information to elements, enhancing web pages with metadata and functionality.

Chapter 5: Images, Video, and Audio- This chapter covers how to enhance your website by incorporating various multimedia elements. You'll learn how to add images, embed video and audio files seamlessly into web pages, and create both ordered and unordered lists for structured content.

Chapter 6: HTML Forms: Collecting User Input- This chapter covers the essentials of building and structuring forms in HTML for effective user input collection.

Chapter 7: Introduction to CSS- This chapter provides a comprehensive overview of CSS, covering its fundamental syntax including selectors, properties, and values. It explores different methods for linking CSS to HTML—external, internal, and inline styles.

Chapter 8: CSS for Layouts- This chapter begins with an exploration of CSS positioning techniques, including static, relative, absolute, and fixed positioning.

Chapter 9: Responsive Design with CSS- This chapter introduces the principles of responsive design, starting with media queries and their role in tailoring web experiences to various devices.

Chapter 10: Styling Forms and Buttons with CSS- This chapter explores advanced CSS techniques for enhancing the appearance of web forms and buttons. It begins with customizing form inputs and text areas for a cohesive design.

Chapter 11: Introduction to JavaScript- This chapter provides a comprehensive introduction to JavaScript, covering fundamental concepts essential for beginners. It starts with JavaScript basics, including syntax and data types, then progresses to variables and constants.

Chapter 12: Working with the DOM- In this chapter you will learn **Document Object Model** (**DOM**) and its pivotal role in web development. This chapter covers selecting and manipulating HTML elements to dynamically update content and styles.

Chapter 13: Building a Responsive Website from Scratch- In this chapter, you'll learn how to build a fully responsive website from the ground up. It starts with structuring the site using semantic HTML, followed by applying styles with CSS to create a visually appealing layout.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/7811f8

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Web-Development-with-HTML-CSS-and-JavaScript**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

At **www.bpbonline.com**, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks. You can check our social media handles below:


Instagram


Facebook


Linkedin


YouTube

Get in touch with us at: **business@bpbonline.com** for more details.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com



Table of Contents

1. **Introduction to Web Development**
    - Introduction
    - Structure
    - Objectives
    - Understanding web development
    - Overview of web browsers and their working
    - Browser engines
    - Browser rendering process
    - Overview of HTML, CSS, and JavaScript
        - *HTML*
        - *CSS*
        - *JavaScript*
    - Conclusion
    - Points to remember
    - Exercises
2. **Setting up the Development Environment**
    - Introduction
    - Structure
    - Objectives
    - Code editors and IDEs
    - Installing VS Code
        - *Windows*

macOS
File structure and organization for web projects
Test server
Conclusion
Points to remember
Exercises

3. Creating Your First Web Page
Introduction
Structure
Objectives
Basic HTML structure
Creating an HTML file
Displaying text and headings
Using browser developer tools
Conclusion
Points to remember
Exercises

4. HTML Elements and Attributes
Introduction
Structure
Objectives
Understanding HTML tags and elements
Working with text
Headings
Paragraphs
Bold text
Italic text
Text color

Fonts
Page background color
Adding tables
Adding hyperlinks
Ordered lists
Unordered lists
Nesting lists
Conclusion
Points to remember
Exercises

5. Images, Video, and Audio
Introduction
Structure
Objectives
Adding images
Embedding videos
Embedding audio files
Conclusion
Points to remember
Exercises

6. HTML Forms: Collecting User Input
Introduction
Structure
Objectives
Introduction to HTML forms
Creating a form
Input types

Adding labels
Grouping form elements
Form processing
HTML form validation
Conclusion
Points to remember
Exercises

7. Introduction to CSS

Introduction
Structure
Objectives
Understanding CSS
Selectors, properties, and values
Selector types
Universal selector
Type selector
Class selector
Linking CSS to HTML
Inline styles
Internal styles
External styles
Working with colors and fonts
Named colors
Hexadecimal colors
Fonts
Adding special effects
Text effects
Pseudo-elements
Shadows

Rounded corners
Gradients
Pseudo-classes
Transitions
Animations
Using the box model
Understanding display properties
Introduction to Sass
Conclusion
Points to remember
Exercises

8. CSS for Layouts
Introduction
Structure
Objectives
CSS positioning
Flexbox
CSS grid layout
Float and clear
CSS measurement units
Conclusion
Points to remember
Exercises

9. Responsive Design with CSS
Introduction
Structure
Objectives

Adapting layouts for different screen sizes
Responsive layouts
Introduction to media queries
Creating mobile-first designs
Testing responsive websites
Calculating with CSS using calc(), min(), max()
Calc
Min
Max
Conclusion
Points to remember
Exercises

10. Styling Forms and Buttons with CSS

Introduction
Structure
Objectives
Customizing an HTML form
Styling buttons
Conclusion
Points to remember
Exercises

11. Introduction to JavaScript

Introduction
Structure
Objectives
JavaScript basics
Basic syntax

Reserved words
Variables and constants
Working with numbers, strings, and Booleans
Numbers
Basic operations
Mathematical functions
Strings
String operations
Useful string methods
Booleans
Output
Integration with HTML
Flow control
Conditional statements
If statement
If-else statement
Switch statement
Loops
For loop
While loop
do-while loop
Functions, parameters, and arguments
Defining a function
Types of function parameters
Return statement
Function scope
Hoisting
Introduction to arrays
Defining arrays
Accessing array elements
Modifying array elements

Adding and removing elements
Length of an array
Iterating over an array
Splice and slice
Objects and classes
Basic error handling with try...catch
Conclusion
Points to remember
Exercises

12. Working with the DOM
Introduction
Structure
Objectives
Understanding the Document Object Model
Selecting and manipulating HTML elements
Changing styles with JavaScript
Creating and removing elements dynamically
Event handling
Types of events
Event listeners
Conclusion
Points to remember
Exercises

13. Building a Responsive Website from Scratch
Introduction
Structure
Objectives
Planning

User interface
Creating the project
Adding files
HTML
CSS
JavaScript
Testing the website
Functional testing
User interface testing
Responsive design testing
Cross-browser compatibility
Introduction to debugging
Conclusion
Points to remember
Exercises

Index

CHAPTER 1

Introduction to Web Development

Introduction

This chapter introduces the core concepts of web development, starting with a definition and the essential technologies: **Hypertext Markup Language** (**HTML**), **Cascading Style Sheets** (**CSS**), and JavaScript. It explores the different roles and responsibilities of web developers and highlights how web browsers function to display web content. By the end of the chapter, readers will have a foundational understanding of how websites are built and rendered.

Structure

The chapter covers the following topics:

- Understanding web development
- Overview of web browsers and their working
- Browser engines
- Browser rendering process
- Overview of HTML, CSS, and JavaScript

Objectives

By the end of this chapter, you will have a clear understanding of what web development is, including its significance in building and maintaining websites and web applications for the World Wide Web. You will be able to differentiate among front end, back end, and full-stack development roles, and recognize the specific responsibilities associated with each. You will become familiar with the core front end technologies: HTML, CSS, and JavaScript, and understand how they work together to create structured, styled, and interactive web pages. Additionally, you will gain a basic insight into the purpose of back end development. You will understand the function of web browsers, their internal components such as browser engines, and the steps involved in rendering a web page from server response to a fully interactive user experience. Finally, you will be introduced to the foundational concepts of HTML document structure, CSS styling principles, and basic JavaScript functionality, setting the stage for deeper exploration in subsequent chapters.

Understanding web development

Web development is the overall process of building websites or web applications for the World Wide Web. This includes designing the site, building the layout, coding, and creating content.

HTML, CSS, and JavaScript are often used for the front end, or client side. The front end is what the users can see in a browser when they access the site. The back end of the website is often implemented using **Python Hypertext Preprocessor** (**PHP**), or Node.js, among others. The back end handles requests from clients, accesses databases, and contains the core functionality or logic of the website, as shown in *Figure 1.1*:



***Figure 1.1**: Front end vs. back end*

The three primary domains of web development are front end, back end, and full-stack. They are discussed as follows:

- Front end development uses HTML, CSS, and JavaScript to create the portions of a website that the user can view and interact with in a web browser.
- Back end development focuses on creating the website's core functionality, which is typically written in PHP, Python, Node.js, or another programming language, as well as managing database connections.
- Full-stack development is a combination of both front and back end.

A web developer is an IT specialist who builds websites and web apps. The type of developer corresponds to the domain of web development the developer is working in, like front end, back end, and full-stack. In addition to this, there may be web masters who are responsible for maintaining and managing the website. Their duties include overseeing site performance, ensuring content is up-to-date, troubleshooting technical issues, and implementing design changes.

Overview of web browsers and their working

A web browser is a software application used to access websites on the World Wide Web. Popular browsers include *Chrome*, *Firefox*, *Safari*, and

Edge. In the demo, in the following *Figure 1.2*, we have a web browser running on a user's laptop requesting a web page from a web server:



***Figure 1.2**: Browser requesting a web page from a web server*

Browser engines

At the heart of any web browser is a browser engine. This is a core component that is responsible for rendering the web page code it receives from the web server. The following figure shows the rendering process:

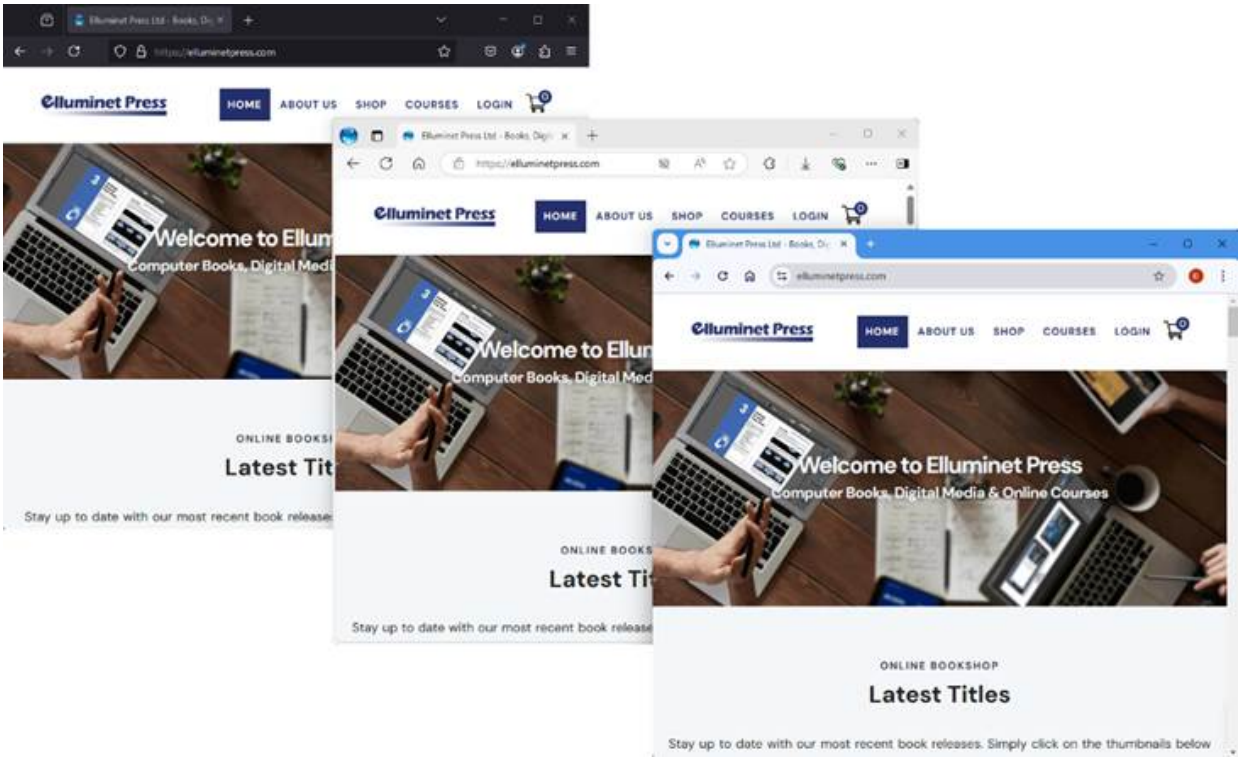


***Figure 1.3**: Firefox, Edge, and Chrome browsers*

There are various browser engines available, depending on which browser you are using, as seen in *Figure 1.3*. The following are the three most common browser engines:

- Blink, which is used by Chrome and Edge.
- Gecko, which is used by Firefox.
- WebKit, which is used by Safari.

Browser rendering process

The browser rendering process involves several steps that transform the raw web content received from a server into a fully rendered and interactive web page. The steps are as follows:

1. When you enter a URL into the browser, the browser sends an HTTP request to the web server, asking for the page.
2. The server responds with the HTML, CSS, and JavaScript files required to display this page. An example of a simple HTML document is shown in *Figure 1.4*:



***Figure 1.4**: Sample HTML document*

The browser begins parsing the HTML document from top to bottom

and builds a **Document Object Model** (**DOM**) tree that represents the structure and content of the page, as shown in *Figure 1.5*:

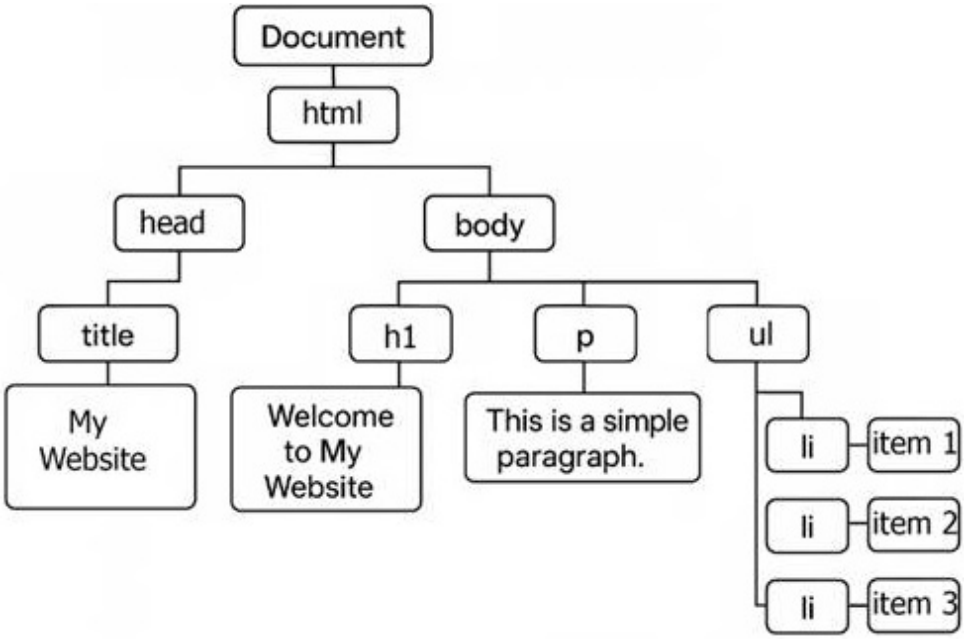


***Figure 1.5**: DOM tree of an HTML document*

As the browser parses the HTML, it simultaneously processes the CSS to determine the styles for each DOM element.

3. If JavaScript is included in the page, the browser executes this while parsing the HTML.
4. The browser then calculates the text style, size, content position, width, height, padding, and margins using the applied CSS styles to determine the page layout. This is used to represent the material graphically, placing each piece in the proper location based on the layout that was determined. This code is rendered in the web browser, as shown in the following *Figure 1.6*:

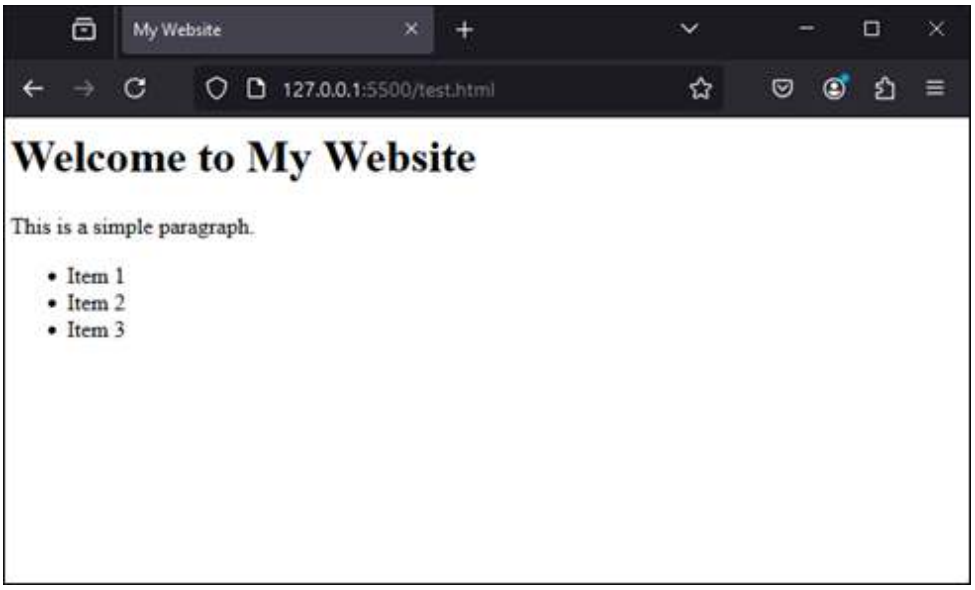


***Figure 1.6**: Rendered HTML page*

Overview of HTML, CSS, and JavaScript

When building websites and web applications, three core technologies form the foundation of the front end user experience, as follows:

- HTML defines the content and structure of a web page.
- CSS is responsible for visual styling and layout.
- JavaScript adds interactivity and dynamic behavior.

These three technologies work together to deliver interactive, styled, and structured content to users through a web browser.

HTML

HTML provides the basic structure and organization of the content of the website that is displayed in the web browser. It consists of various tags to define different types of content, such as headings, paragraphs, links, images, and lists.

Each HTML element starts with an opening tag. For example, the anchor element used to create a clickable link on a web page starts with the tag, as shown in *Figure 1.7*:

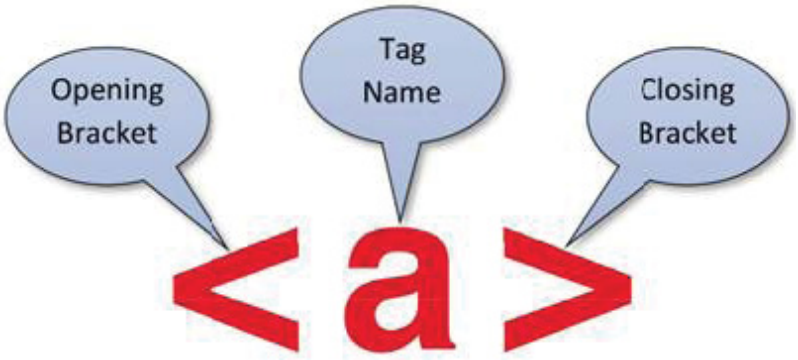


***Figure 1.7**: HTML opening tag*

Most elements also include a closing tag. The closing tag is similar to the opening tag but includes a forward slash before the tag name, as shown in *Figure 1.8*:

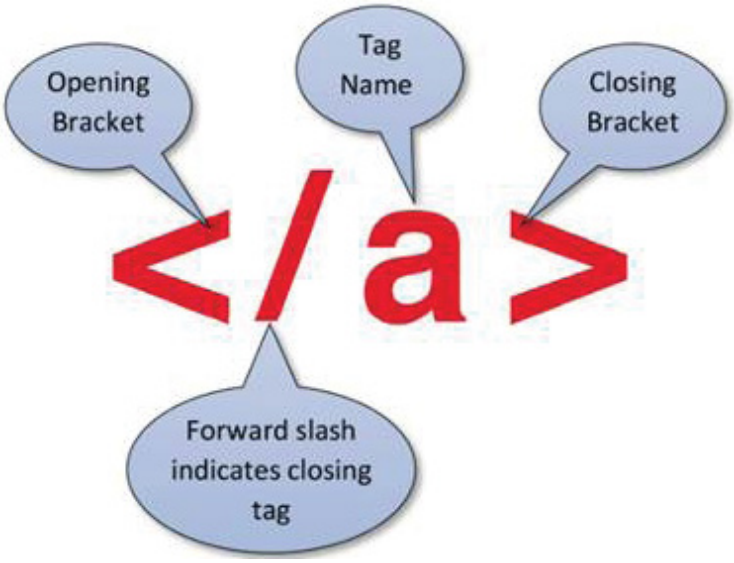


***Figure 1.8**: HTML closing tag*

Any content to be displayed using this tag is contained between the tags; in this example, it will be a clickable link, as follows:

```
<a href="link"> This is a link <a/>
```

Inside the opening tag, you can also include attributes. For example, in this particular element, there is an href attribute that specifies the URL for the link, as follows:

```
<a href="link"> This is a link <a/>
```

Or for text paragraphs, as follows:

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

CSS

CSS is responsible for the visual presentation and layout of a web page; these are known as **styles**. This allows developers to apply these styles to HTML elements to control the color, font size, spacing, and layout. These are known as **CSS selectors**, as shown in *Figure 1.9*:

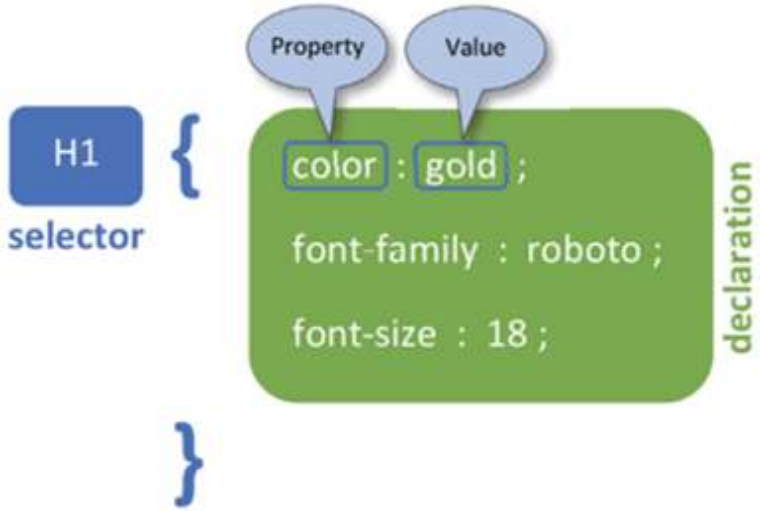


***Figure 1.9**: CSS selector*

With CSS, developers can create visually appealing layouts that adapt to different screen sizes, enhancing the user experience across devices.

JavaScript

JavaScript is a versatile programming language that allows you to manipulate elements on a web page in real-time, meaning you can process user actions, such as clicks and keyboard input, update content, and modify styles without requiring a page reload, as shown in the following code:

```
    <script>
        function addNumbers() {
            const num1 =
parseFloat(document.getElementById('num1').value);
            const num2 =
parseFloat(document.getElementById('num2').value);
```

```
            const sum = num1 + num2;
            document.getElementById('result').innerText
= 'Sum: ' + sum;
        }
    </script>
```

The language includes frameworks, such as React, Angular, and Vue.js, as well as libraries, such as jQuery, which make DOM manipulation easier. JavaScript can also be utilized on the server side thanks to Node.js, which enables developers to write full-stack apps using only one language.

Conclusion

In summary, web development is a multidisciplinary field that involves designing, building, and maintaining websites and web applications. It is broadly categorized into front end, back end, and full-stack development, each contributing distinct yet interrelated functionalities. Front end technologies like HTML, CSS, and JavaScript form the visual and interactive aspects users engage with, while back end technologies handle data processing, business logic, and server-side operations.

In the next chapter, we will take a look at how to set up a development environment, including how to install VS Code, how to structure the project, and how to set up a testing server.

Points to remember

- **Understanding web development**:
  - o Web development refers to building websites or web applications, covering design, layout, coding, and content creation.
  - o Involves two main areas: front end (client-side) and back end (server-side):
    - Front end development uses HTML, CSS, and JavaScript for content display and interactivity in browsers.
    - Back end development handles server-side logic, database

interactions, and core functionality using languages like Python, PHP, or Node.js.

- Full-stack development combines both front end and back end.

- **Web development roles**:
  - Front end developers focus on what users interact with in a browser.
  - Back end developers handle the behind-the-scenes functionality of the website.
  - Full-stack developers work on both the front and back end.
  - Webmasters manage site performance, content updates, troubleshooting, and design changes.
- **Web browsers overview**:
  - Web browsers (e.g., Chrome, Firefox, Safari, Edge) allow users to access websites on the web.
  - Browsers use browser engines to render web pages:
    - Blink for Chrome and Edge.
    - Gecko for Firefox.
    - WebKit for Safari.
- **Browser rendering process**:
  1. The browser sends an HTTP request to a web server for a web page.
  2. The server responds with HTML, CSS, and JavaScript files.
  3. The browser builds a DOM tree from the HTML, styles it with CSS, and executes any JavaScript.
  4. The browser calculates the layout and displays the page.
- **HTML, CSS, and JavaScript overview**:
  - HTML provides the structure of the web page using tags (e.g., headings, paragraphs, links).
  - CSS controls the visual presentation (colors, fonts, layout).
  - JavaScript is a programming language for real-time interaction, handling events like clicks, input, and dynamically updating page content.
  - JavaScript frameworks (React, Angular, Vue.js) and libraries (jQuery) simplify DOM manipulation.

- Node.js allows JavaScript to be used on the server-side for full-stack development.

Exercises

1. What are the main stages involved in the process of web development as described in the text?
2. Which technologies are commonly used for front end development, and what is their purpose?
3. Name three programming languages mentioned for back end development.
4. What is the role of a webmaster according to the content?
5. List four web browsers mentioned in the text.
6. What is a browser engine, and what is its primary function?
7. Which browser engines are used by Chrome, Firefox, and Safari, respectively?
8. Briefly describe the steps a browser takes when rendering a web page, starting from entering the URL.
9. How does HTML define the structure of a web page, and what is an example of an HTML element given in the content?
10. What is the function of the JavaScript code snippet provided in the content?

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com



CHAPTER 2

Setting up the Development Environment

Introduction

In this chapter, you will learn how to prepare your development environment for web projects. You will start by installing **Visual Studio Code** (**VS Code**) as the primary code editor and exploring essential developer tools in modern web browsers. Next, you will install Node.js to enable JavaScript tools and package management. Finally, you will learn how to organize your project files and directories, ensuring a clean and efficient structure for web development. By the end, you will have a fully functional setup ready for coding.

Structure

The chapter covers the following topics:

- Code editors and IDEs
- Installing VS Code
- File structure and organization for web projects
- Test server

Objectives

In this chapter, you will learn the essential skills required to set up a modern web development environment, forming a solid foundation for your future projects. You will begin by exploring the differences between code editors and **integrated development environments** (**IDEs**), helping you choose the right tools for your workflow. You will then install and configure VS Code, a powerful and widely used code editor. This includes choosing a theme, enabling useful options such as *Open with Code* in the context menu, and registering VS Code as the default editor. You will also learn how to organize your project files by creating a dedicated folder structure—an important habit for maintaining clean, scalable codebases. Finally, you will install the live server extension to instantly preview your web pages in a browser, allowing you to test and refine your code in real time. These skills can be directly applied to real-world development scenarios, such as building personal websites, developing client projects, or contributing to open-source applications. A well-configured development environment will boost your efficiency, reduce errors, and make coding more enjoyable and productive. By the end of the chapter, you will have a complete setup in place and be ready to start coding with confidence.

Code editors and IDEs

A code editor and an IDE are both software tools used in web development for writing and managing code. Although they serve similar purposes, they differ in scope and functionality.

An IDE is a software application that consists of a source code editor with syntax highlighting, along with built-in tools to help you develop and debug your code, as well as project management features. These are all integrated into one application, hence the name IDE. For example, *Adobe Dreamweaver* and *Microsoft Visual Studio* (not to be confused with VS Code). You can see Adobe Dreamweaver in *Figure 2.1*:

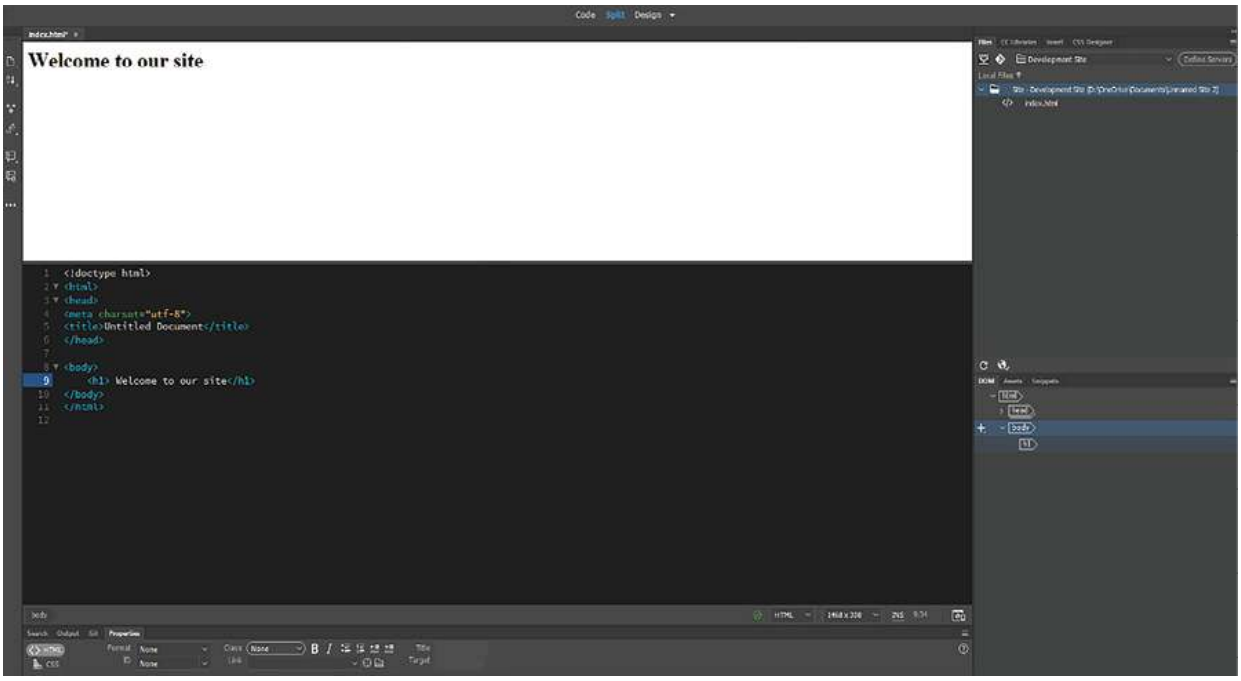


***Figure 2.1**: Adobe Dreamweaver*

A code editor is primarily designed for writing and editing code with a focus on simplicity and speed. It also offers syntax highlighting and basic debugging tools. Some examples include *VS Code*, *Brackets*, *Sublime Text*, and *Notepad++*.

Brackets is a free editor that can be downloaded from the developer's website:

www.brackets.io

In the following *Figure 2.2,* the Brackets window is open and is a sound editor for coding:

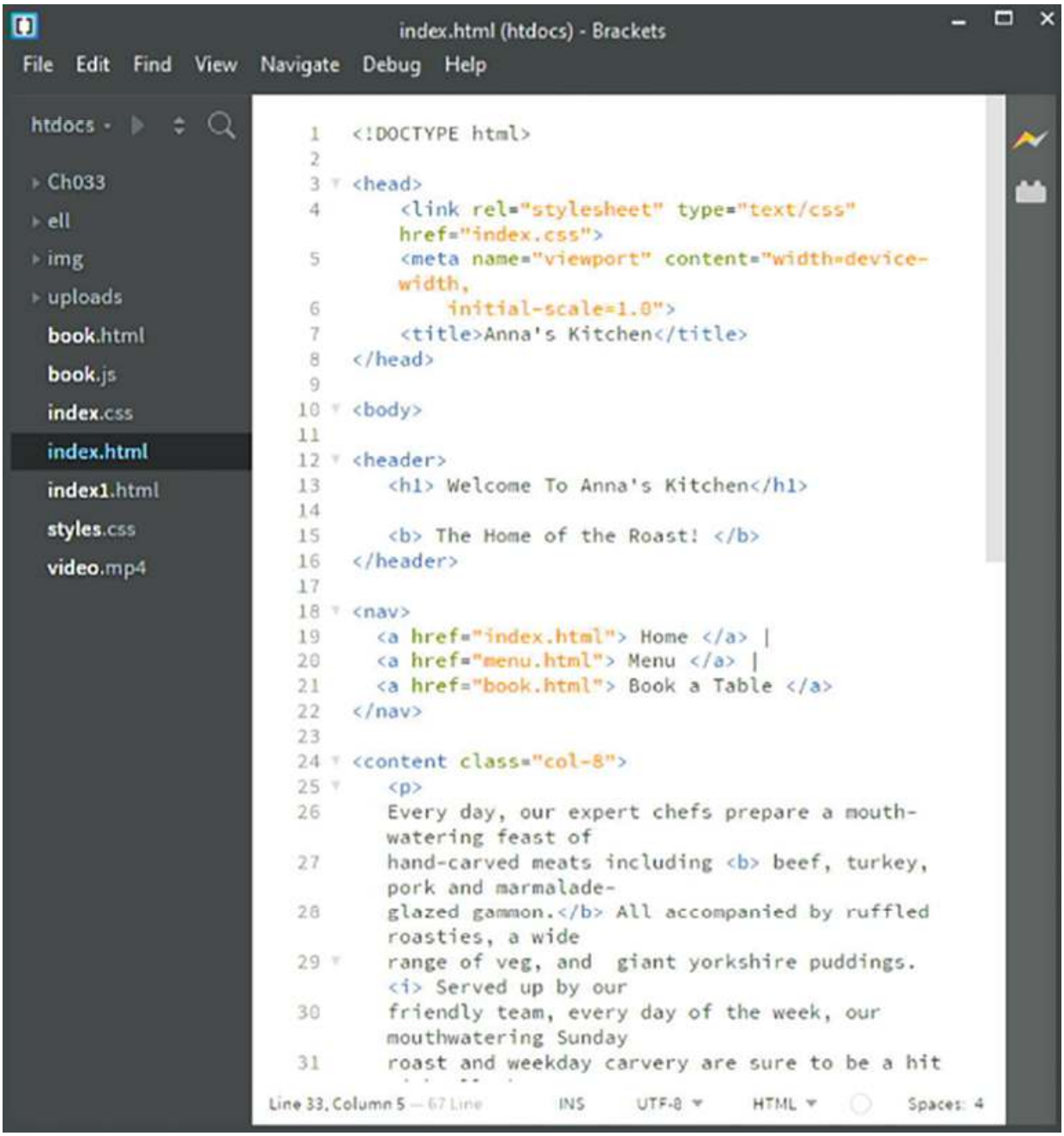


***Figure 2.2**: Brackets*

You can open up your live preview to see what your page looks like as you are writing your code, as shown in the following figure:

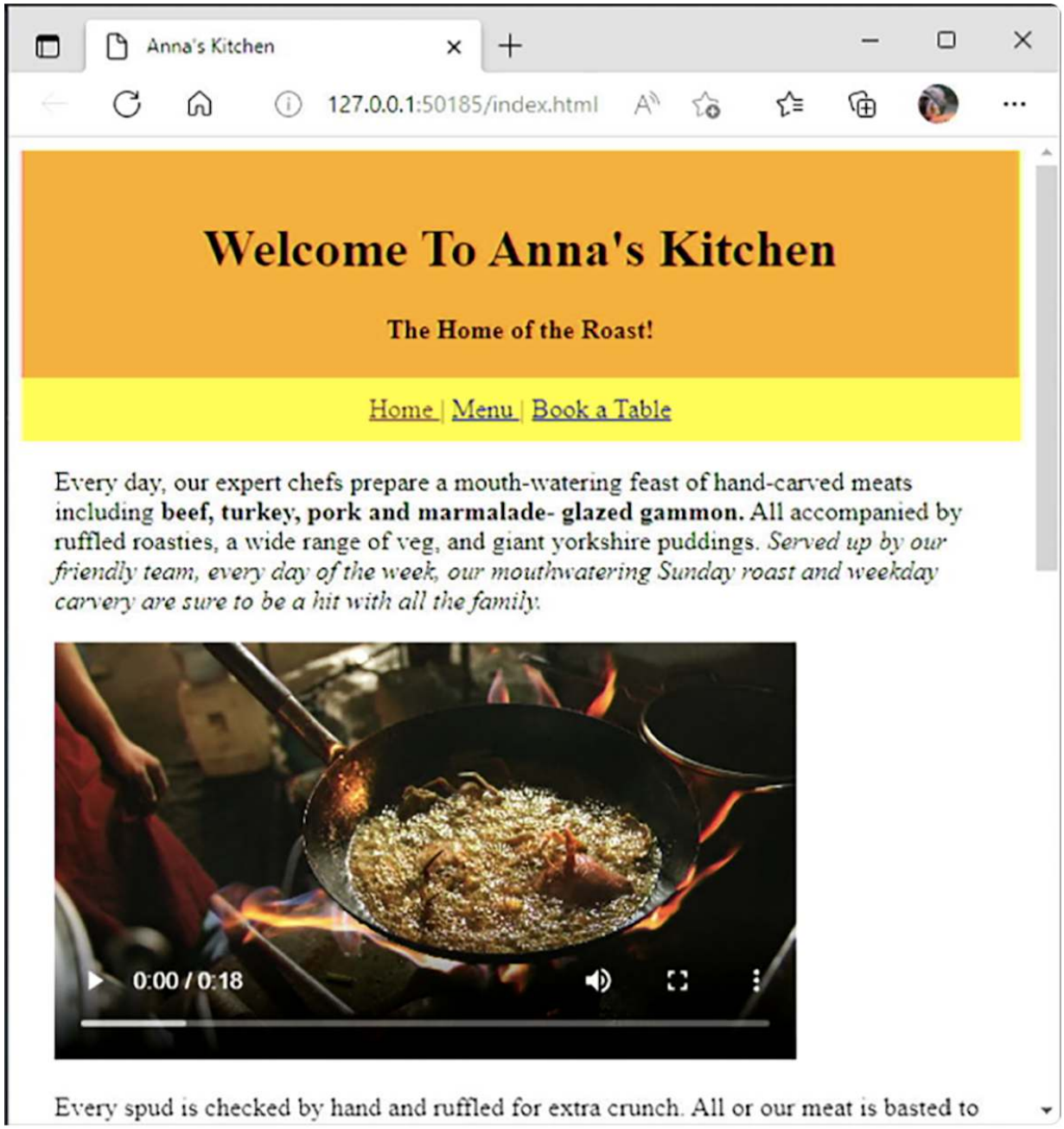


***Figure 2.3**: Live preview*

One of the most popular code editors is VS Code, and it is what we will be using throughout this book. In *Figure 2.4,* we can see VS Code with a simple HTML project open.

Feel free to install any code editor or IDE you prefer, like the one shown in the following figure:

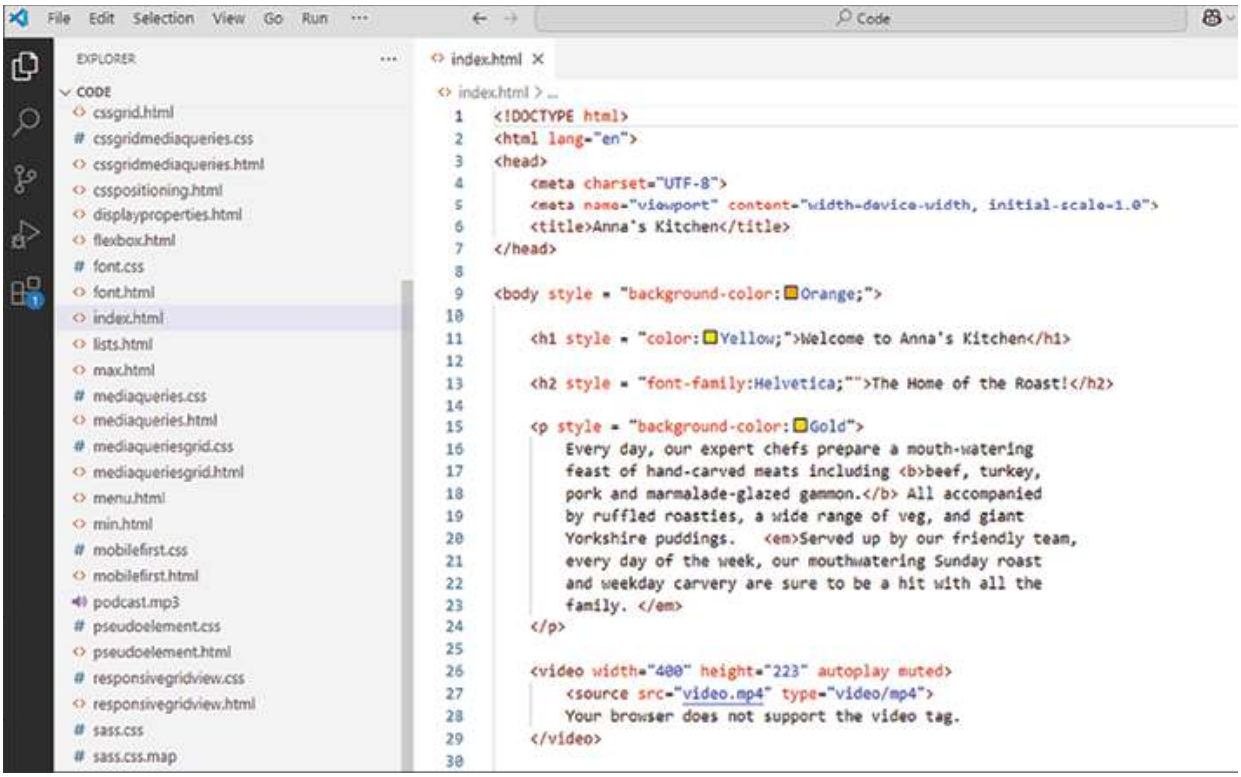


***Figure 2.4**: VS Code*

Installing VS Code

In this section, we will examine the process of installing VS Code on a computer. Installation steps for both Windows and macOS operating systems will be covered.

Windows

To begin using VS Code, it must first be installed on the local machine. This section provides a step-by-step overview of the installation process for Windows. The procedure involves downloading the official installer, configuring basic setup options, and launching the application upon completion. The steps are outlined as follows:

1. Open the web browser, then go to the official VS Code website:
   https://code.visualstudio.com/
2. On the website, click on **Download for Windows**, as shown in the following figure:



***Figure 2.5**: Download VS Code for Windows*

3. Once the download is complete, you will find the file in the downloads folder.
4. In **File Explorer**, navigate to the **Downloads** folder and double-click on the `.exe` to begin the installation, as shown in the following figure:

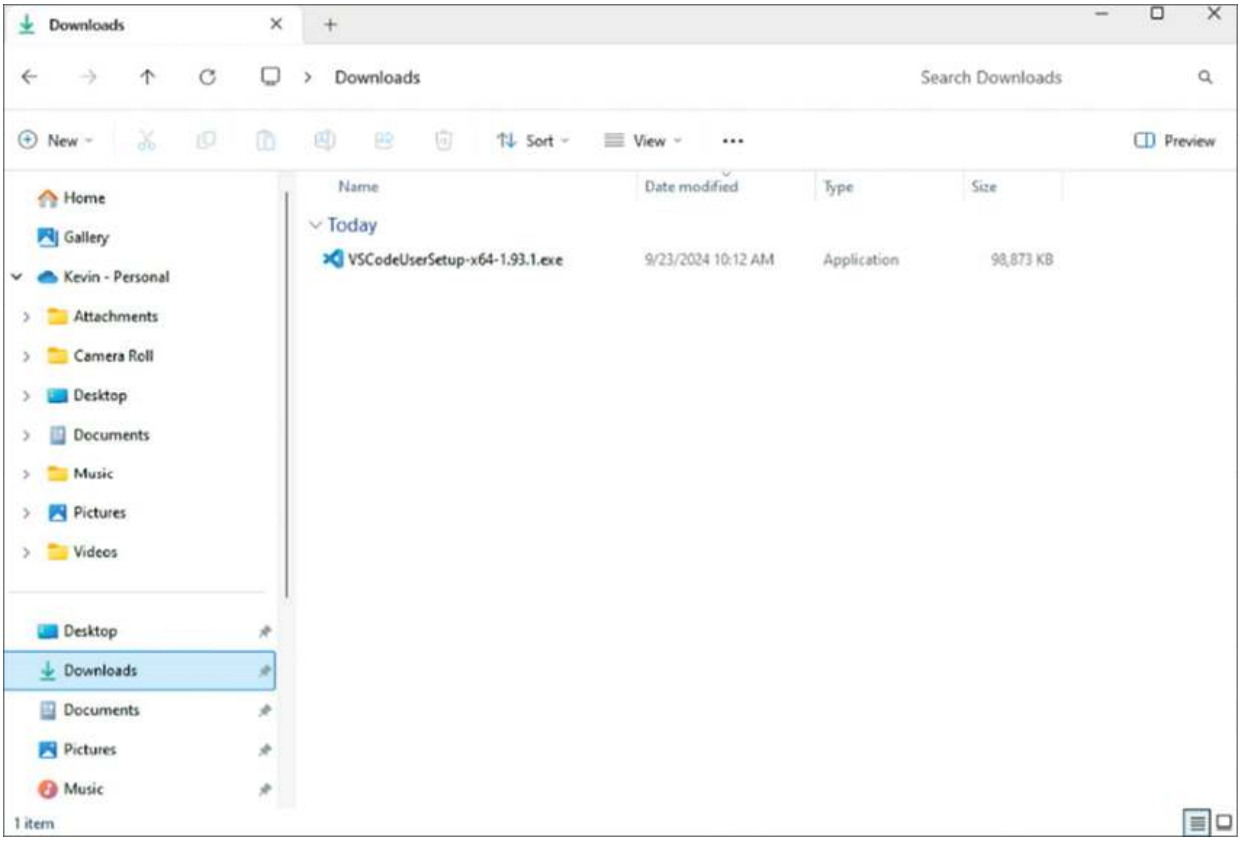


***Figure 2.6**: File Explorer downloads folder*

5. Run through the installation process, agree to the terms and conditions, and click **Next**, as shown in the following figure:

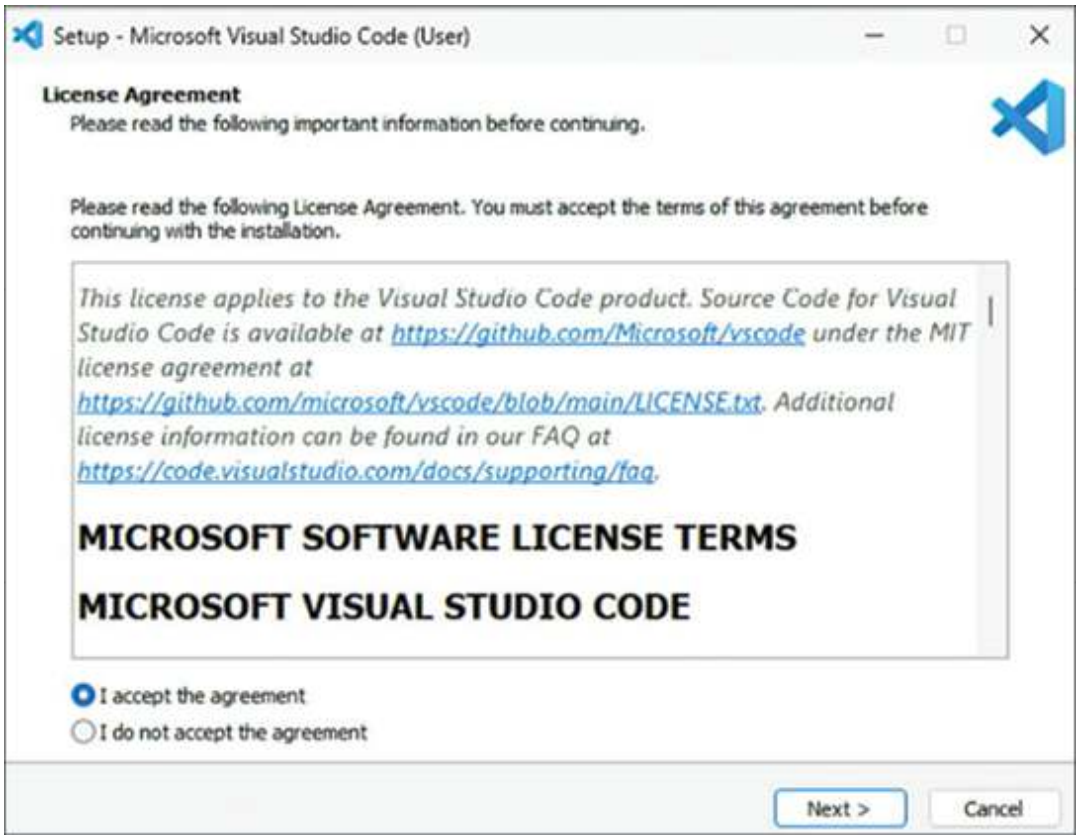


***Figure 2.7**: Terms and conditions*

6. Select the installation folder if necessary. If you need to change the directory, click **Browse**, then select the directory you want, and click **Next**, as shown in the following figure:

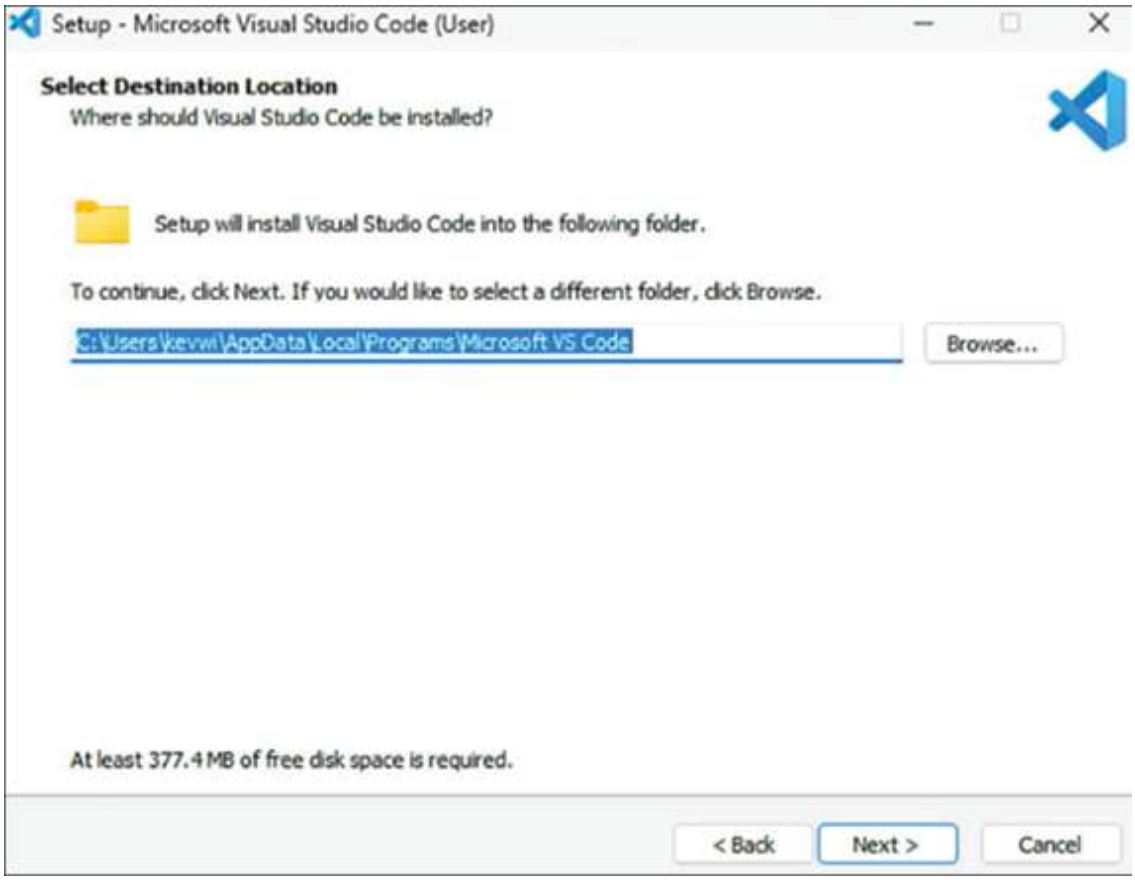


***Figure 2.8**: VS Code installation*

7. Choose your installation options, as shown in *Figure 2.9*. Add the Open with Code action to the context menu (useful for opening files/folders directly). Register as the default editor for supported file types. Select **Add to PATH** and click **Next**, as shown in the following figure:

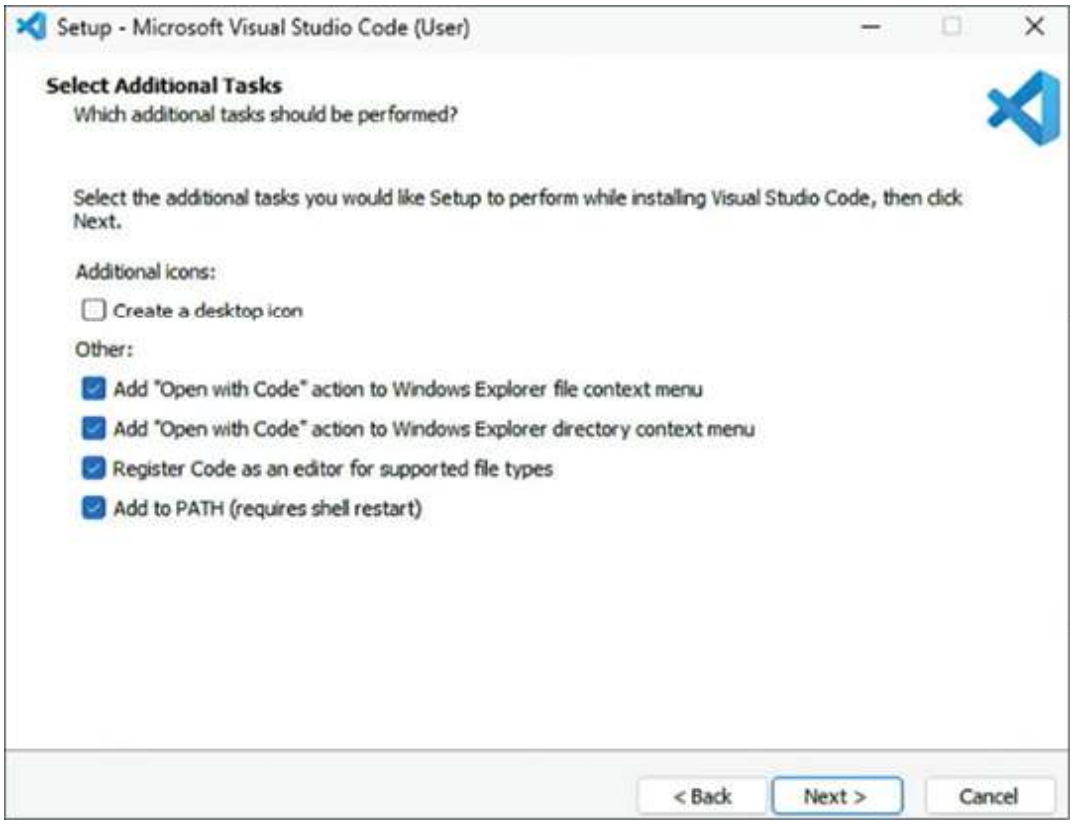


***Figure 2.9**: VS Code options*

8. Click the **Install** button to complete the installation. After installation, you can launch VS Code from the **Start** menu.
9. When you run VS Code for the first time, choose the theme. If you spend a lot of time coding, using the **Dark Modern** theme is recommended, as this shows your code on a dark background and will reduce eye strain:

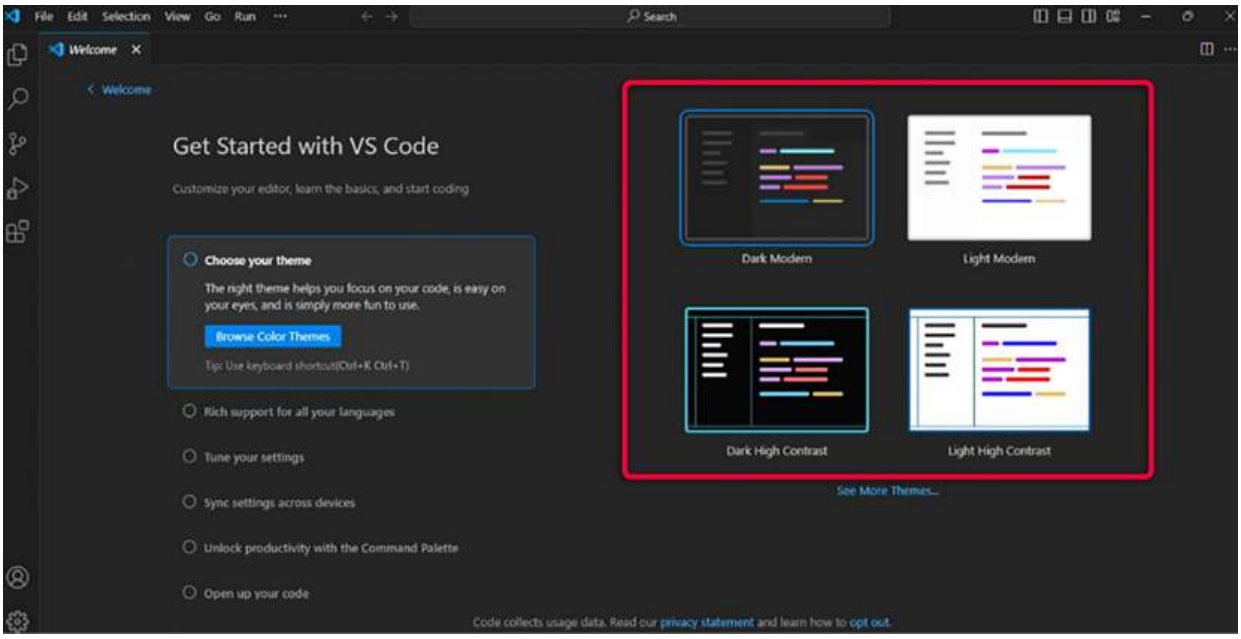


***Figure 2.10**: VS Code themes*

10. Once you have chosen the theme, you can close the **Welcome** tab. Click the **x** on the top right of the tab, as shown in the following figure:

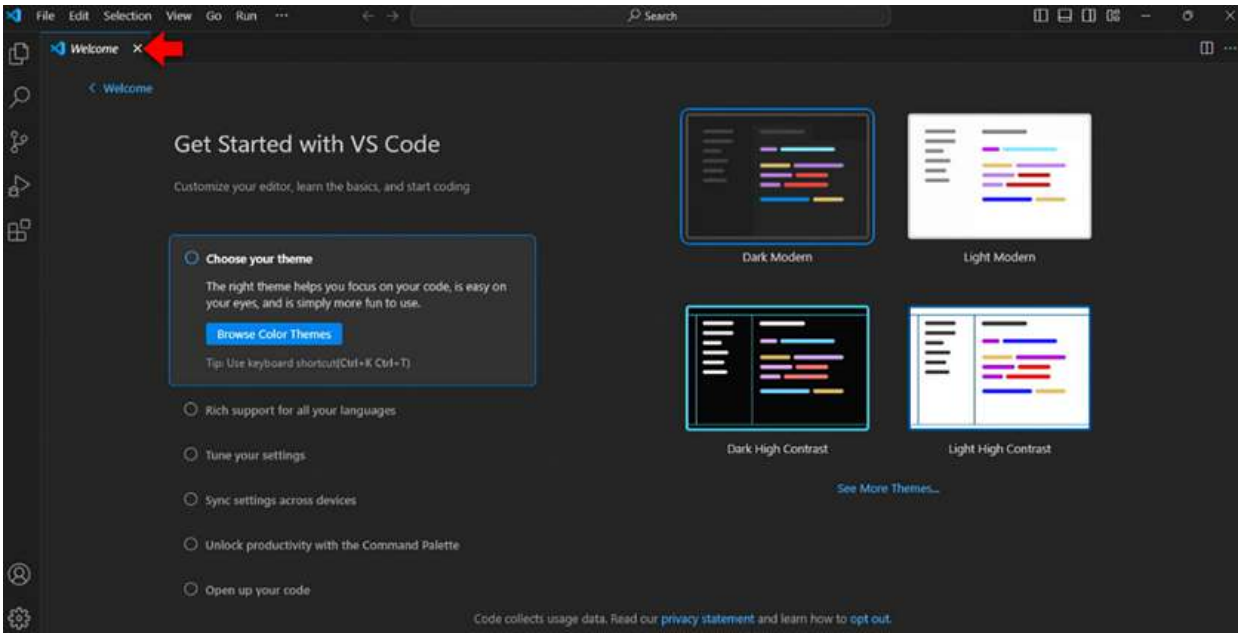


***Figure 2.11**: VS Code welcome*

macOS

To begin using VS Code on macOS, the application must first be installed on

the system. This section provides a step-by-step overview of the installation process for macOS. The procedure includes downloading the official installer, moving the application to the appropriate directory, and launching it for the first time. The detailed steps are outlined as follows:

1. Open the web browser, then go to the official VS Code website:
   https://code.visualstudio.com/
2. Click on the **Download for macOS** button, as shown in the following figure:

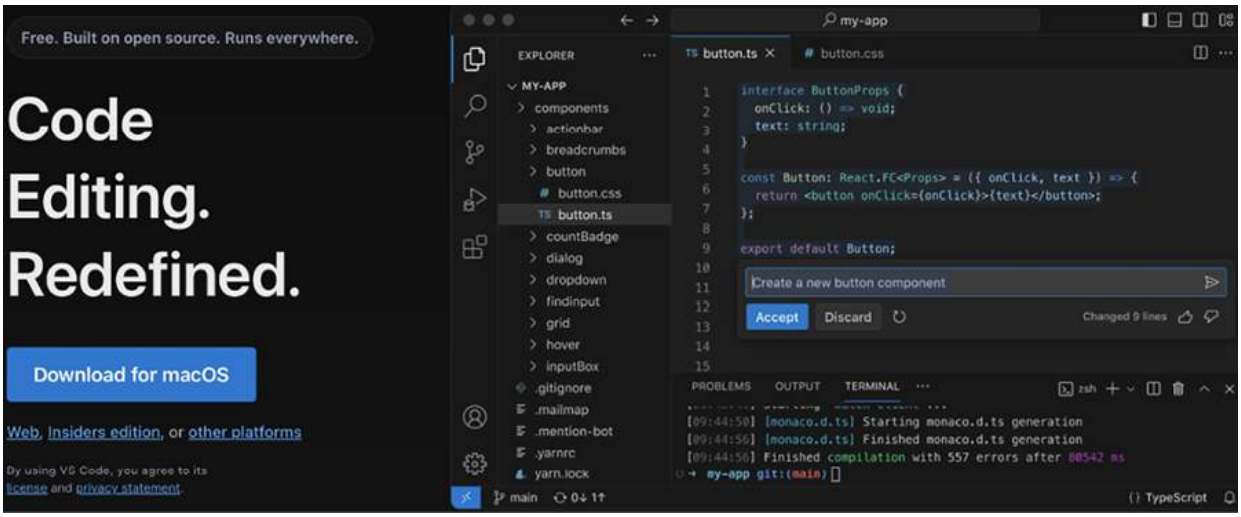


***Figure 2.12**: VS Code download page*

3. Open **Finder**, navigate to the **Downloads** folder as shown in *Figure 2.13*, then double-click on the **`.zip`** file you just downloaded to extract the contents:

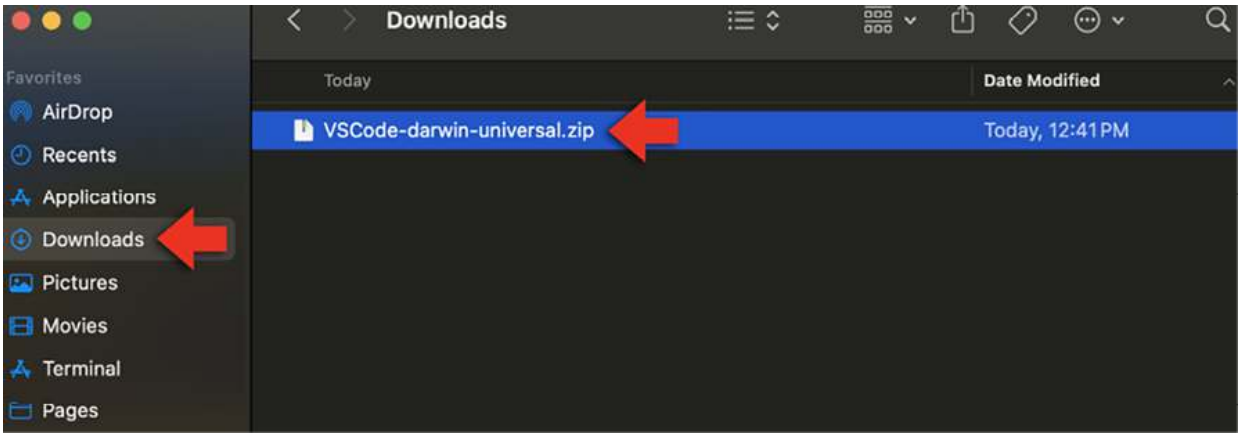


***Figure 2.13**: Finder downloads folder*

4. You will see an application appear: **Visual Studio Code**. Click and drag this file into the **Applications** folder, as shown in *Figure 2.14*. Enter the Mac username and password if prompted.

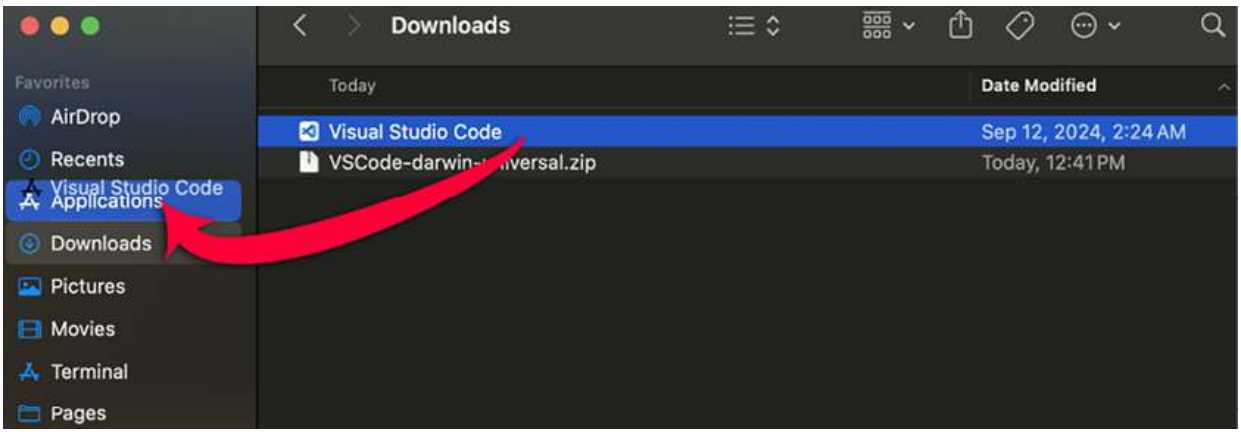


***Figure 2.14**: Copy to Applications folder*

5. Open the **Applications** folder, as shown in *Figure 2.15*, then double-click on the **Visual Studio Code** icon to open the app:



***Figure 2.15**: Start VS Code*

File structure and organization for web projects

Once we have installed VS Code, we need to create a new project folder

where we will save all our HTML, CSS, and JavaScript files for our website. To do this, follow these steps:

1. Go to the **File** menu, then select **Open Folder...**, as shown in the following figure:



***Figure 2.16**: VS Code open folder*

2. In the **Open Folder** dialog box, navigate to the folder you want to save the project files into.

   For example, a new folder named **website** will be created within the **Documents** directory. This folder will serve as the storage location for all files generated during the development process. It is recommended to create a separate folder for each project or website in order to maintain an organized and structured workspace. Refer to the following figure:

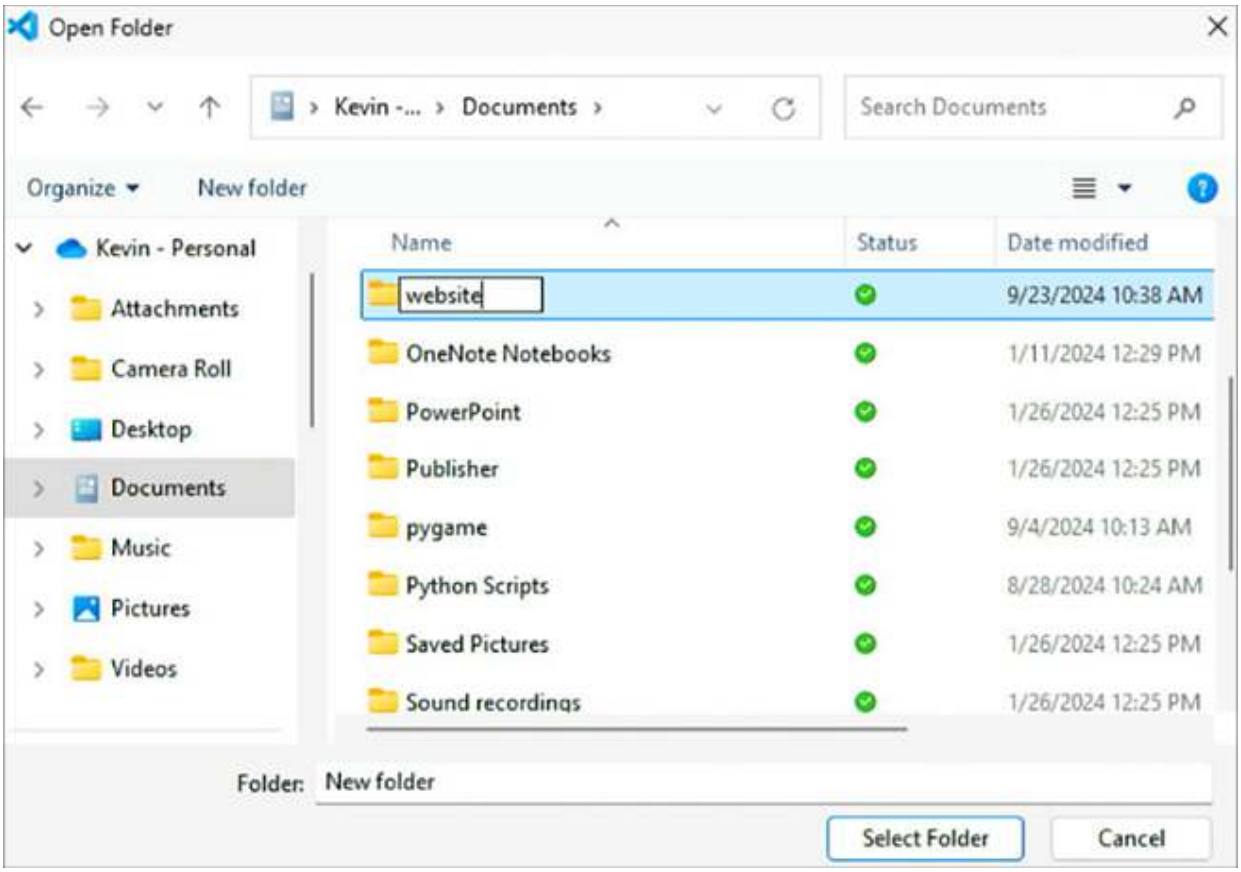


***Figure 2.17**: VS Code create folder*

Test server

Once we have our project folder set up, we need a way to test our web pages. There are various ways to do this; for example, you can use the live server extension in VS Code.

This is particularly useful when working on HTML, CSS, and JavaScript files, as it provides an instant preview of your web pages in a web browser such as Chrome, Edge, or Firefox.

To set it up, follow these steps:

1. Go to the **Extensions** view, as shown in the following figure:



***Figure 2.18**: VS Code live server*

2. Then search for **`Live Server`** and click **Install**. Once the server is installed, close the **Extension: Live Server** tab:

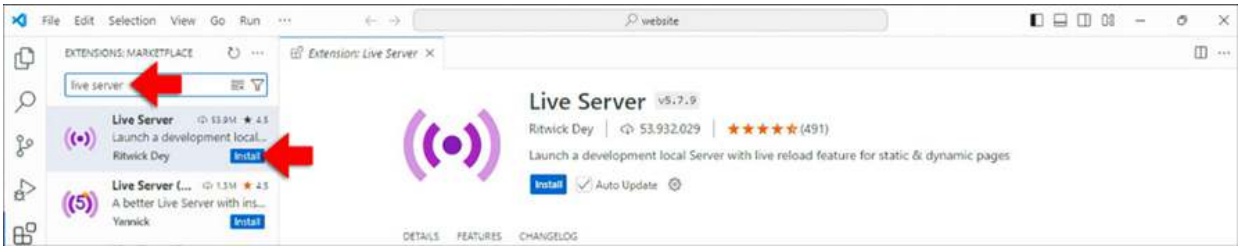


***Figure 2.19**: VS Code live server install*

3. Select the **Explorer** icon on the left-hand side to go back to the **Explorer** view:



***Figure 2.20**: VS Code live server install*

Conclusion

In this chapter, you set up a complete and efficient development environment for web projects. You learned the differences between code editors and IDEs, installed Visual Studio Code, and explored how to customize it for your workflow. You also organized your project files into a structured folder system and added a test server to preview your work in real time. These

foundational skills are essential for writing, managing, and testing web code effectively. With your tools now in place and ready for use, you are prepared to begin creating actual web content. In the next chapter, we will start building your first web page using HTML, the core language of the web.

Points to remember

- VS Code is the recommended code editor.
- The live server extension is a good option for instant previewing of web pages during development.

Exercises

1. What is the difference between a code editor and an IDE?
2. Why is VS Code recommended as a code editor for web development?
3. How do you install Visual Studio Code on Windows or Mac?
4. Why is it recommended to create a separate folder for each web project you work on?
5. How can VS Code help in testing your web projects?

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com



CHAPTER 3

Creating Your First Web Page

Introduction

In this chapter, you will learn the foundational steps to create a simple web page. Starting with how to set up an HTML file, we will explore the basic structure of HTML, including essential elements like the doctype, head, and body tags. You will discover how to display text and headings on your page and then view your web page in a browser. Finally, we will introduce you to browser developer tools, helping you inspect and debug the web page efficiently.

Structure

The chapter covers the following topics:

- Basic HTML structure
- Creating an HTML page
- Displaying text and headings
- Using browser developer tools

Objectives

By the end of this chapter, you will be able to construct a basic HTML document using correct structural elements, including the doctype declaration, **`<html>`**, **`<head>`**, and **`<body>`** tags. You will understand how to create and save an HTML file using a code editor, and how to render it in a web browser.

Additionally, you will be able to use heading elements (**`<h1>`** through **`<h3>`**) and paragraph tags (**`<p>`**) to display text content on a web page in a semantically meaningful way. You will learn how HTML documents are structured and how each part of the code contributes to the final output seen in the browser.

This chapter will also help you build essential workflow habits such as organizing your code for readability and previewing your work using live server tools. You will gain familiarity with using a professional code editor (such as Visual Studio Code) and take advantage of productivity features like Emmet abbreviations to generate HTML more efficiently.

Finally, you will develop a foundational understanding of how to access and utilize browser developer tools. These tools will enable you to inspect and troubleshoot the HTML structure of a web page, test changes in real time, and build confidence in debugging common layout or rendering issues, skills that are crucial as you progress into more advanced web development topics.

Basic HTML structure

An HTML document consists of different elements, which the browser interprets to display content to users. The following *Figure 3.1* shows the basic components of an HTML document and how various elements work together to form a web page:

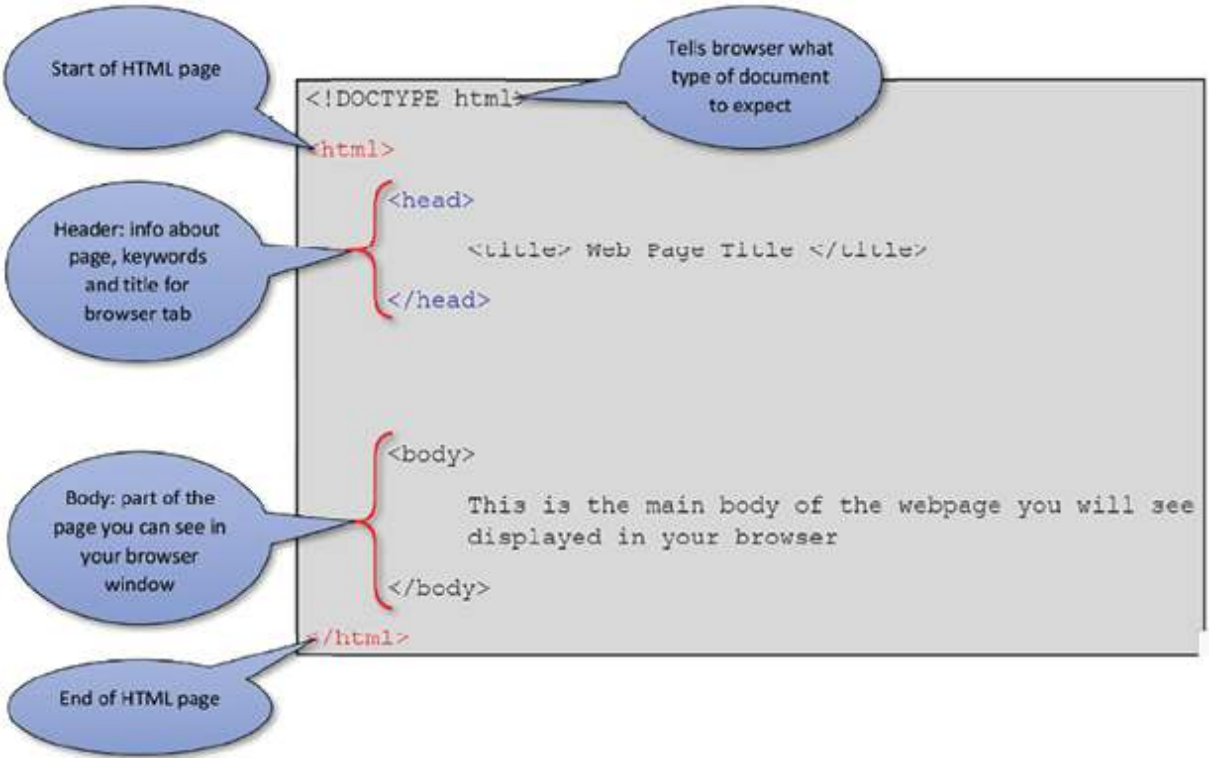


***Figure 3.1**: HTML document structure*

At the top of the document, you will find the document type declaration. This tells the browser what type of document to expect. In this case, **`<!DOCTYPE html>`** for an HTML document.

Next, the **`<html>`** tag marks the beginning of the document. Inside this, there are two main sections: **`<head>`** at the top and **`<body>`** underneath.

The **`<head>`** section contains the **`<title>`** tag and is used to define the title that appears in the browser's tab or at the top of the browser's window when the web page is opened. Other metadata can also be included here, such as viewports, character sets used, and links to external files such as JavaScript and CSS files.

The **`<body>`** section contains all the visible content, such as text, images, videos, and links that the user will see in the browser window when they open the web page.

Creating an HTML file

Before a web page can be displayed in a browser, its underlying HTML file

must be created. An HTML file contains the markup that defines the structure and content of the web page. This section outlines the process of creating a new HTML file using a code editor, with a focus on VS Code. You will learn how to create a properly named HTML file, generate the basic document structure, and prepare the file for viewing in a browser. These initial steps form the foundation of any web development project.

An HTML file can be created using the following steps:

1. In the **EXPLORER** panel on the left-hand side, right-click in the project, then select New File from the pop-up menu, as shown in *Figure 3.2*:

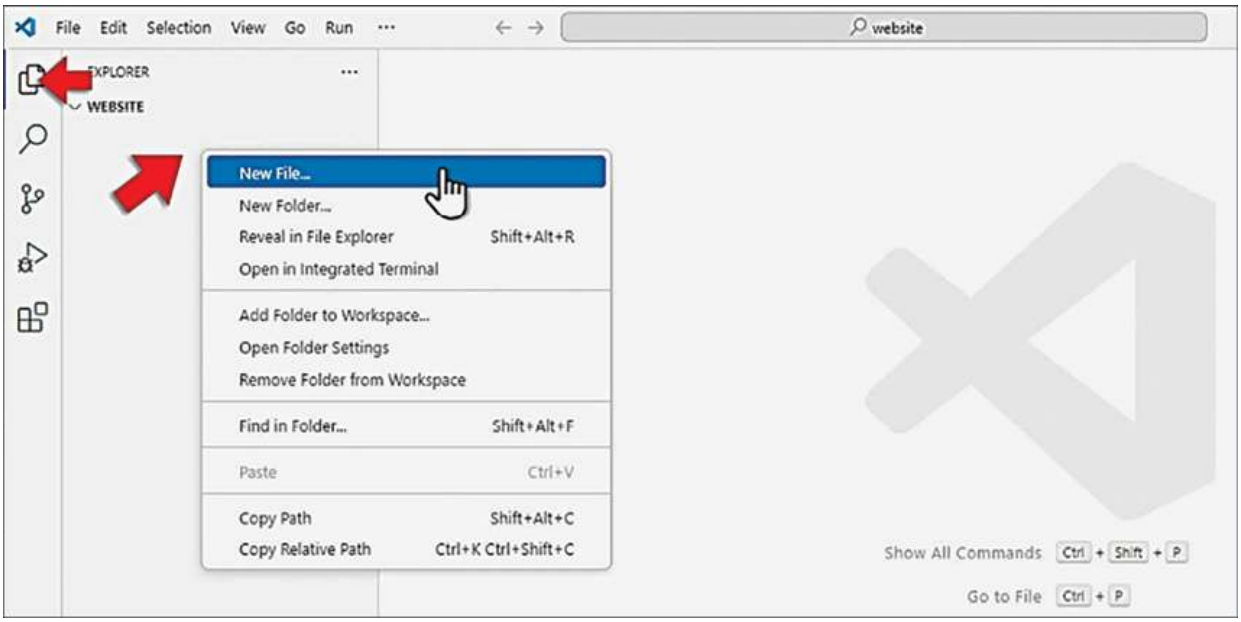


***Figure 3.2**: VS Code create new file*

2. Name the file **`index.html`**. This will open the file for editing, as shown in *Figure 3.3*:

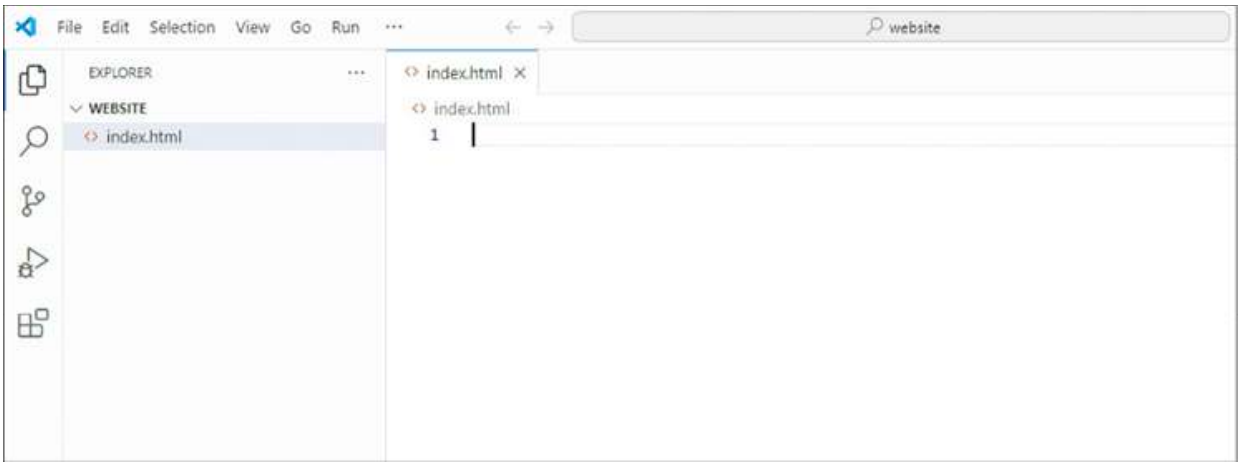


***Figure 3.3**: VS Code blank file*

3. Now, the basic HTML structure can be added, as shown in *Figure 3.4*. VS Code provides a feature called **Emmet**, which is handy for speeding up HTML coding. The basic HTML structure can be generated by typing an exclamation point (**!**) and then pressing the *Tab* key. Refer to the following figure:

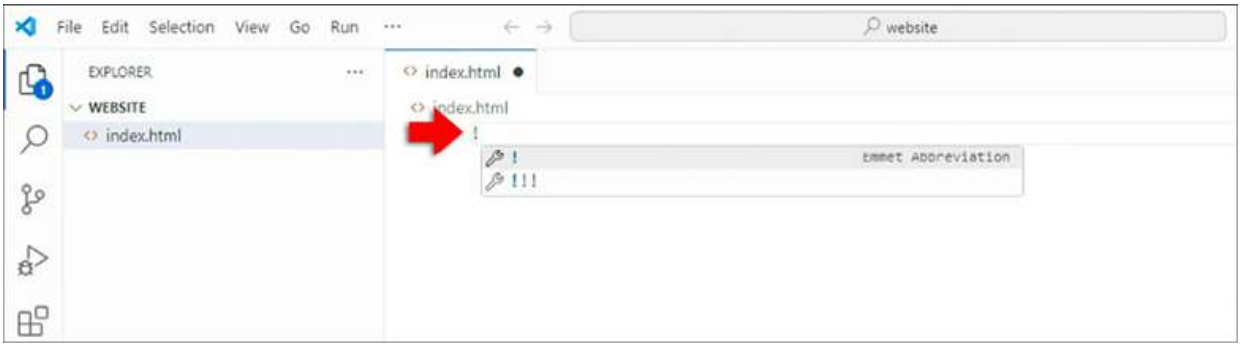


***Figure 3.4**: VS Code Emmet*

4. You will see a basic template of an HTML document you can edit to add the content, as shown in *Figure 3.5*:

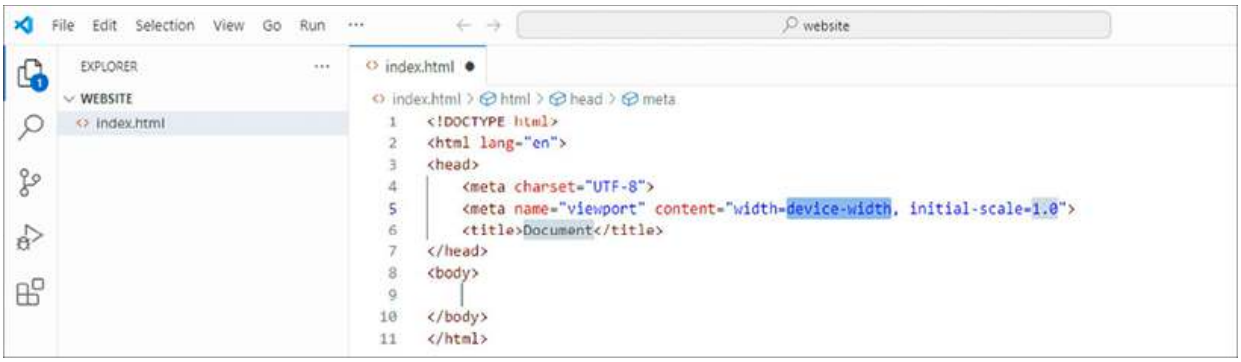


***Figure 3.5**: VS Code HTML template*

Let us take a look at what each of these tags means:

- **`<!DOCTYPE html>`** on *line 1* declares the document type and version of HTML.
- **`<html lang=”en”>`** on *line 2* is the root element of the document. The attribute **`lang="en"`** specifies the language the document's content is written in. For example, **`lang = “en”`** for English, **`lang="es"`** for Spanish, **`lang="fr"`** for French, etc. This tells the browser and any assistive technologies, such as screen readers, what language to expect.
- Between **`<head>`** and **`</head>`** contains meta-information about the document, such as character encoding and the title of the document. Within the **`<head>`** element, you can add metadata, such as **`<meta charset="UTF-8">`**, which specifies the character encoding for the document. You can also add various other elements, such as **`<meta name="viewport" content="width=device-width, initial-scale=1.0">`**, which controls how the web page is displayed on different devices, particularly mobile devices, which is crucial for responsive web design.
- Between **`<body>`** and **`</body>`** contains the content that will be displayed on the web page. This is where the visible content for the website is written.

5. Save the **`index.html`** file. You can select **Save** from the file menu, or press *Ctrl+S* (or *CMD+S* in macOS) on the keyboard.
6. Now, to preview the file using the live server, right-click on the file in the explorer panel on the left-hand side, then select **Open with Live**

Server from the pop-up menu, as shown in *Figure 3.6*:

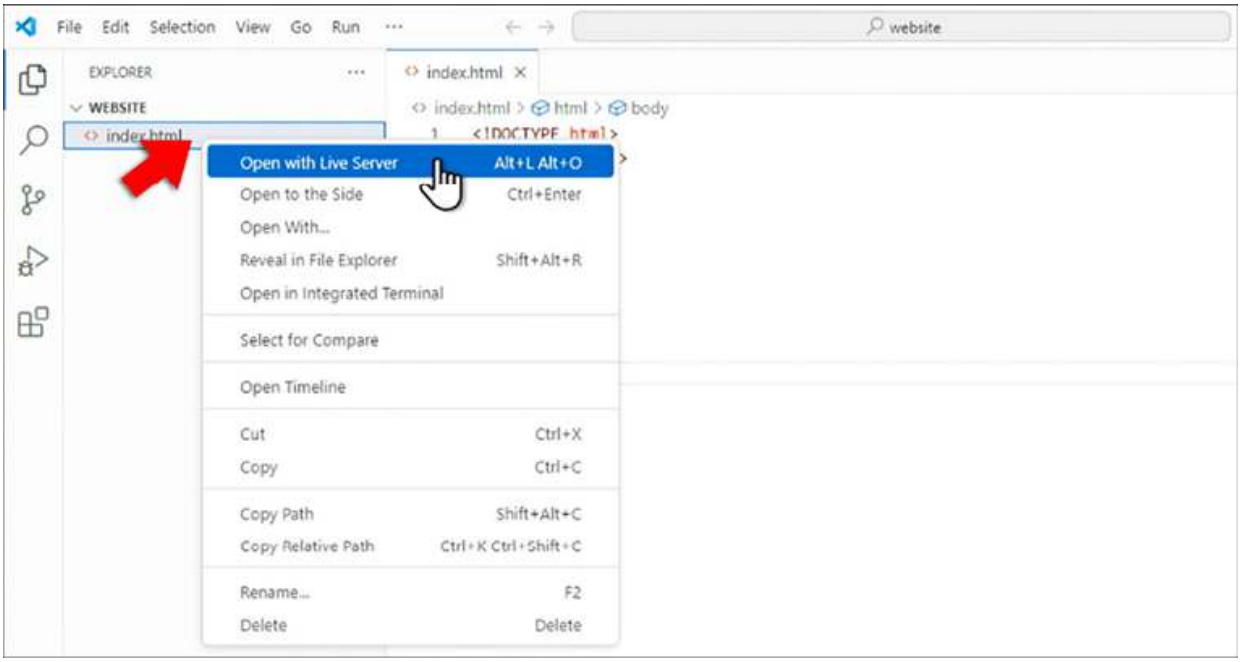


***Figure 3.6**: VS Code open file in live server*

7. You will see the web page open in the default web browser. Now, since this file is only the structure, the page will be blank, as shown in *Figure 3.7*:

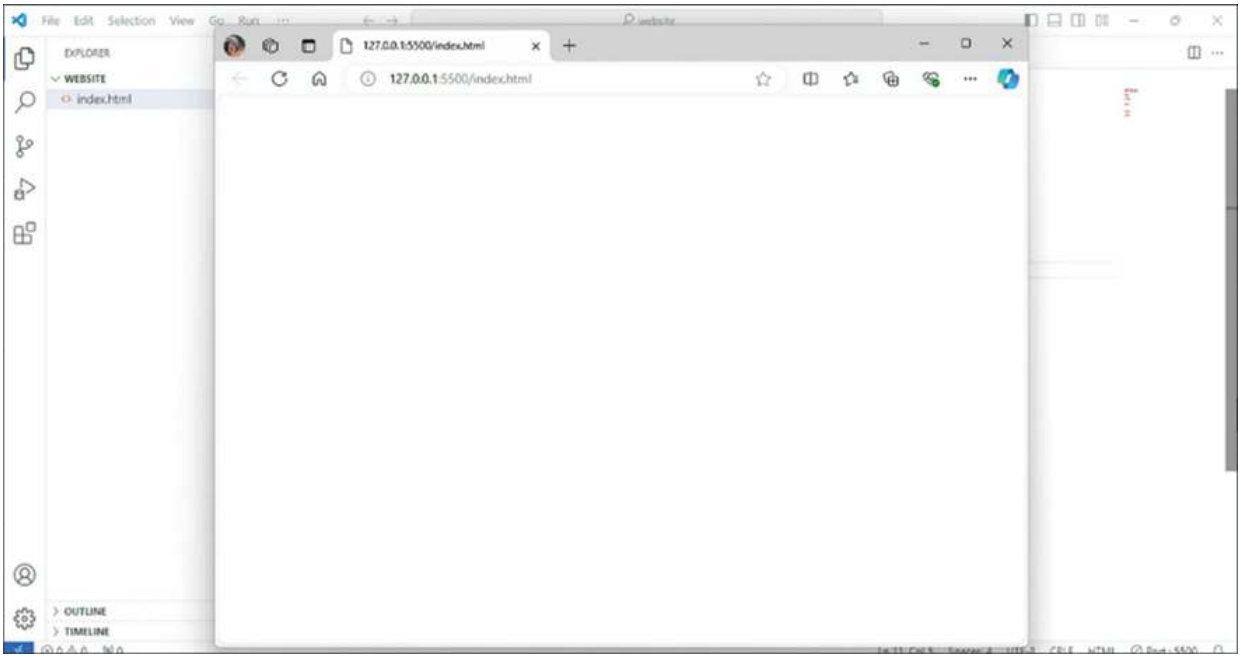


***Figure 3.7**: Page viewed in live server*

Displaying text and headings

Now that we have the basic structure of an HTML document, we can start to add some content. Let us begin by adding a few headings and some text.

Inside the **<body>** section, you can add headings of different sizes. Headings often follow a hierarchy to provide a clear structure to the content, like:

- The main heading of the page uses **<h1>**.
- Written content is often divided into sub-sections with subheadings and often uses **<h2>**.
- If any of these sub-sections are further divided, then **<h3>** is used.

Each heading begins with a start tag, such as **<h1>**, and a corresponding end tag **</h1>**. The content to be displayed on the page is contained between the tags.

For example, in *Figure 3.8*, we have added a main title (using the **<h1>** tag), with a subheading for each section using the **<h2>** tag (section 1 and section 2). Each section has some further subsections using the **<h3>** tag. It can be seen in the browser window how these headings appear on the web page:

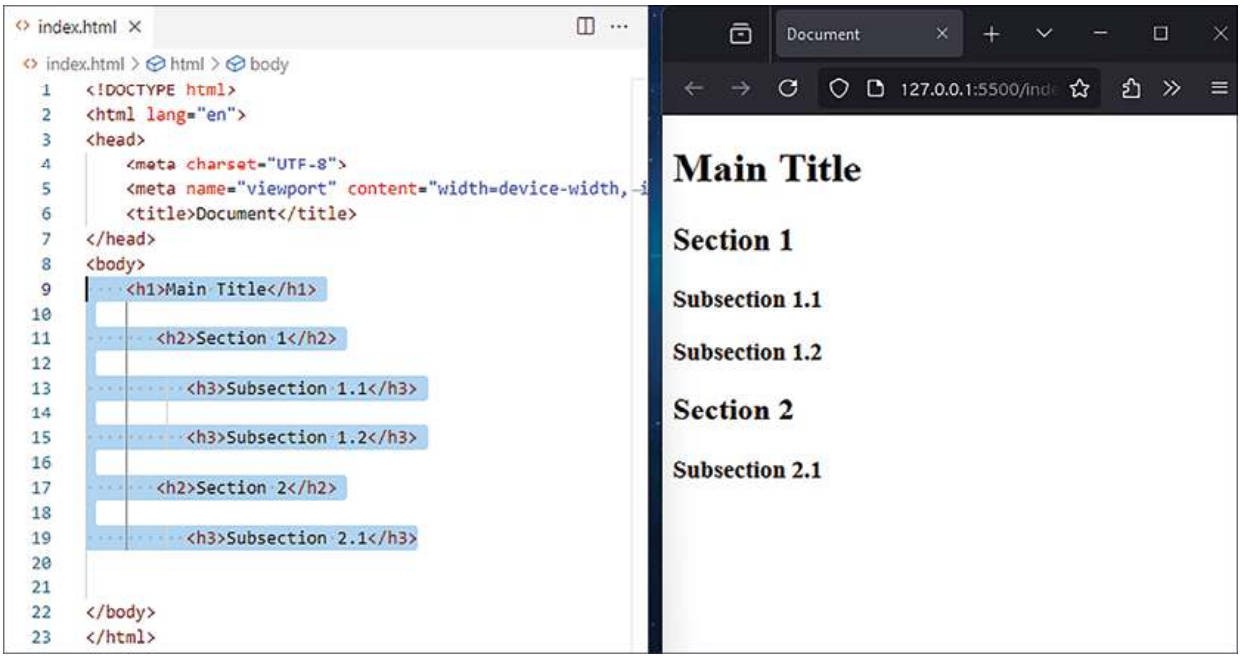


***Figure 3.8**: HTML headings*

You can also add paragraph text. In *Figure 3.9*, we have added a paragraph of text under the section 1 heading **`<h2>`**, using the **`<p>`** tag. Each paragraph you want to add is contained between the **`<p>...</p>`** tags. In the example, you can see some filler text appear under section 1.

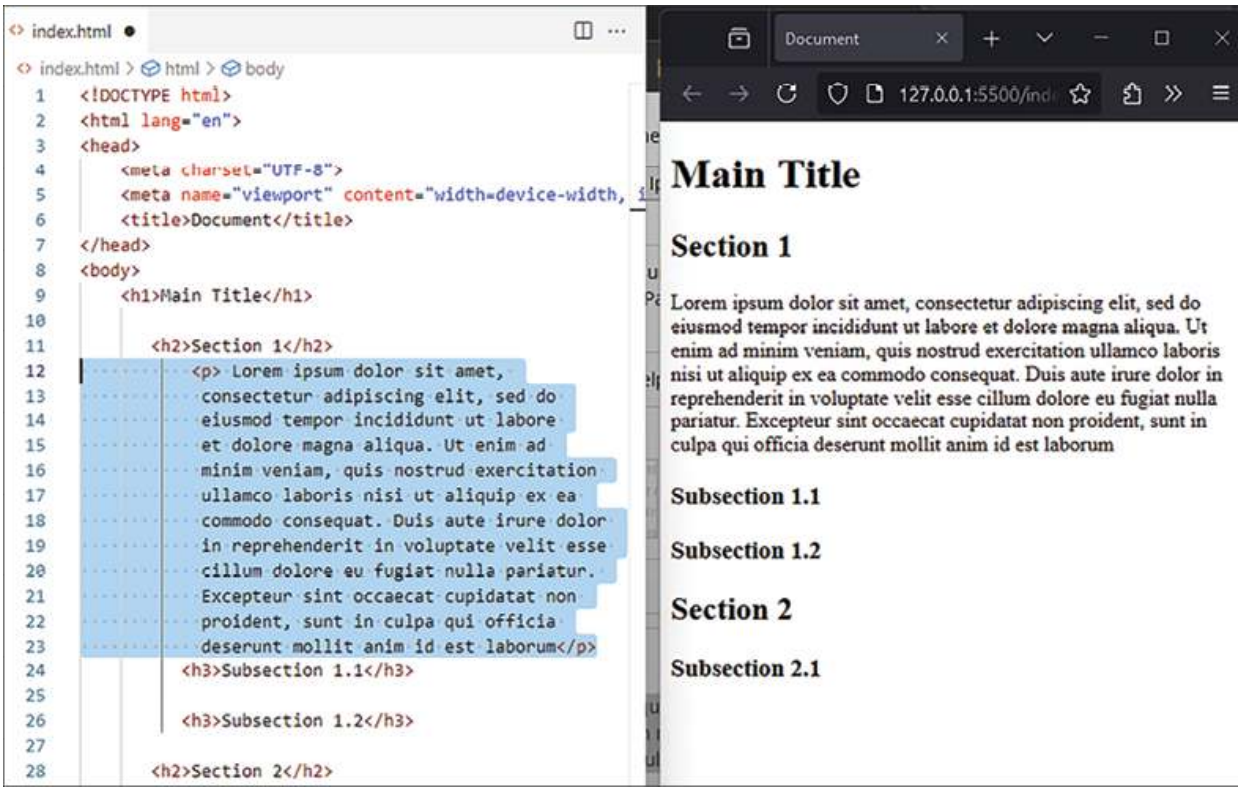


***Figure 3.9**: HTML paragraphs*

Using browser developer tools

Most modern web browsers include built-in tools that allow developers to view and modify the DOM tree, temporarily change HTML or CSS code, analyze network requests, debug JavaScript, and optimize performance. The dev tools panel can be opened by pressing *F12* on the keyboard.

In *Figure 3.10*, we can see a simple website with the dev tools panel open on the right-hand side:

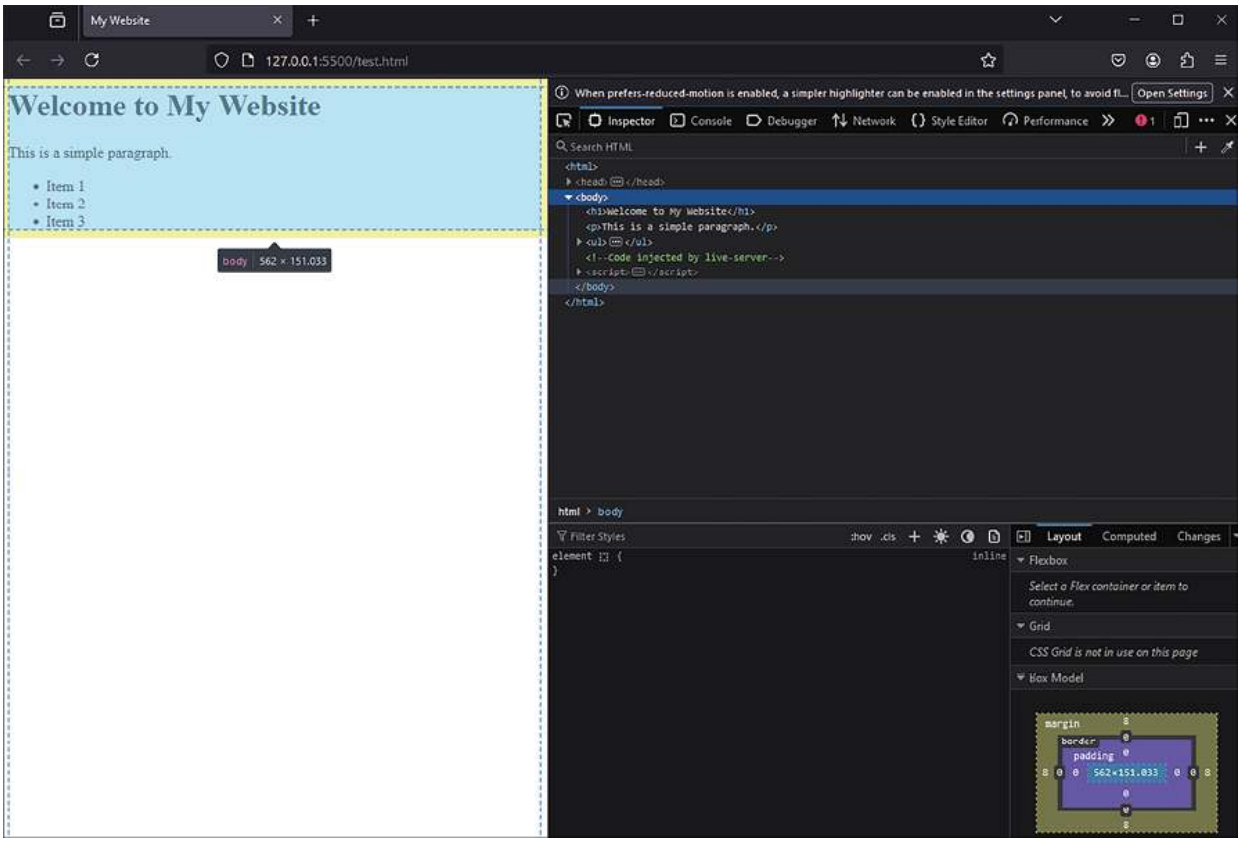


***Figure 3.10**: Browser developer tools*

Conclusion

Now that you understand the basic structure of an HTML document and how to create and view HTML files, you can start creating simple web pages using headings and text. You also have the tools to inspect and debug the code using browser developer tools, which will help analyze how the content is rendered.

In the next chapter, you will expand your skills by learning how to work with various HTML elements used to structure and enrich your content. This includes using tags to format text, create tables, insert hyperlinks, and organize information using ordered, unordered, and nested lists. These tools will allow you to build more structured, readable, and interactive web pages.

Points to remember

- <!DOCTYPE html> specifies the type of document and the HTML version being used.
- The <html> tag is the root element of an HTML document.
- The <head> section contains metadata and resources like the title, character set, and external files.
- The <body> section holds all the visible content displayed on the web page.
- Headings are defined using <h1> to <h6> tags, with <h1> being the highest level and <h6> the lowest.
- Paragraphs are defined using the <p> tag to display blocks of text.
- Browser developer tools can be used to inspect, edit, and debug HTML and CSS in real time.
- Use Emmet shortcuts in Visual Studio Code to quickly create HTML structures.

Exercises

1. What is the purpose of the <!DOCTYPE html> declaration at the beginning of an HTML document?
2. Describe the difference between the <head> and <body> sections in an HTML document.
3. How do you create a new HTML file in Visual Studio Code, and what is the shortcut to quickly generate the basic HTML structure?
4. Which tag would you use to define the main heading of a web page, and how do you create subheadings within the document?
5. What is the purpose of browser developer tools, and how can you access them in a web browser?

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com



CHAPTER 4

HTML Elements and Attributes

Introduction

This chapter introduces the fundamental building blocks of HTML: tags and elements. You will learn how to use attributes to add additional information to elements, enhancing web pages with metadata and functionality. The chapter covers essential text formatting using paragraphs, headings, and line breaks, along with techniques for emphasising content with and tags. Finally, you will create hyperlinks using anchor tags, a core feature for connecting web content.

We will demonstrate this by creating a simple static website for a fictional restaurant called *Anna's Kitchen*.

Structure

The chapter covers the following topics:

- Understanding HTML tags and elements
- Working with text
- Adding tables
- Adding hyperlinks
- Ordered lists

- Unordered lists
- Nested lists

Objectives

In this chapter, you will develop a strong understanding of the core components that make up an HTML document. You will learn how to use HTML tags and elements to structure content on a web page and how attributes can enhance these elements with additional information and styling.

You will gain hands-on experience formatting text using paragraph and heading tags, and learn how to emphasize key content using both presentational (`<b>`, `<i>`) and semantic (`<strong>`, `<em>`) tags that support accessibility and best practices. You will also use inline styles to control the appearance of text, such as font type, color, and background, giving you greater creative control over the visual design of your pages.

Beyond text, you will gain essential layout skills by learning how to construct HTML tables to display structured data clearly, with cell padding, headers, and styling. You will also learn how to use anchor tags to create hyperlinks that connect different pages or websites, a foundational skill for building navigation systems.

In addition, you will explore how to organize content using ordered, unordered, and nested lists, allowing you to present information in a clean and hierarchical format. By combining these skills, you will be able to design practical and well-structured web pages, such as menus, profiles, or information panels.

The chapter uses a hands-on mini project—Anna's Kitchen—to help explain these techniques.

Understanding HTML tags and elements

HTML elements mark parts of the web page, such as headings, text formatting with bold or italic, paragraphs, images, links, and tables. Let us have a look at some simple formatting elements.

Most HTML elements begin and end with HTML tags called **opening** and **closing tags,** respectively. Between these tags, you enclose the content to be displayed on the web page. The opening tag begins with the tag name (in this case, **a** for anchor) and ends with the same tag name preceded by a forward slash, both enclosed in angle brackets **< >**, as shown in *Figure 4.1*:

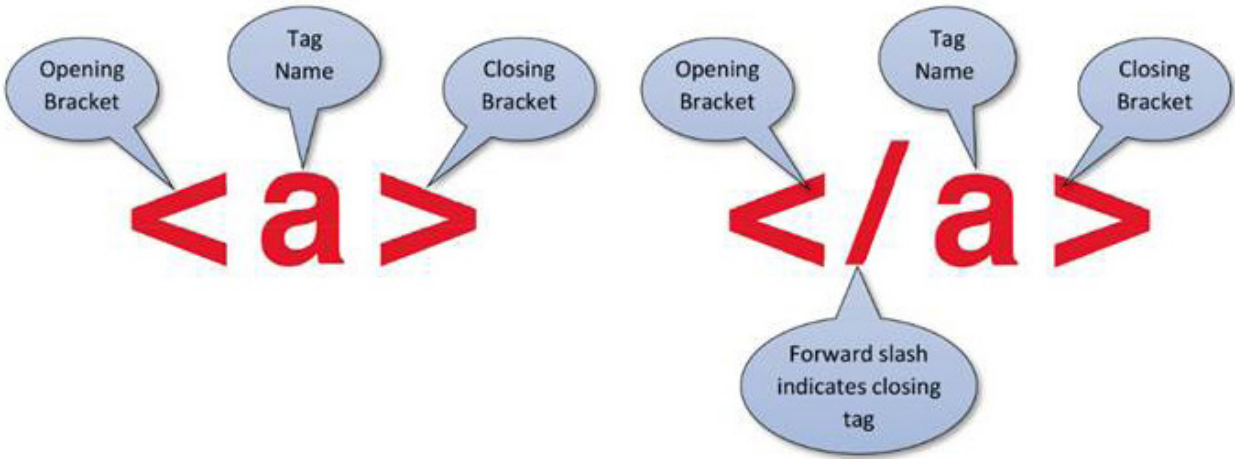


***Figure 4.1**: HTML tags*

In *Figure 4.2*, we see the anchor element, which is used to add hyperlinks to a web page. We can see the whole element consists of an opening **<a>** tag and a closing **** tag. In the opening **<a>** tag, we can include attributes, and between the opening and closing tags, we include the visible content (the bit we want to show up on the web page).

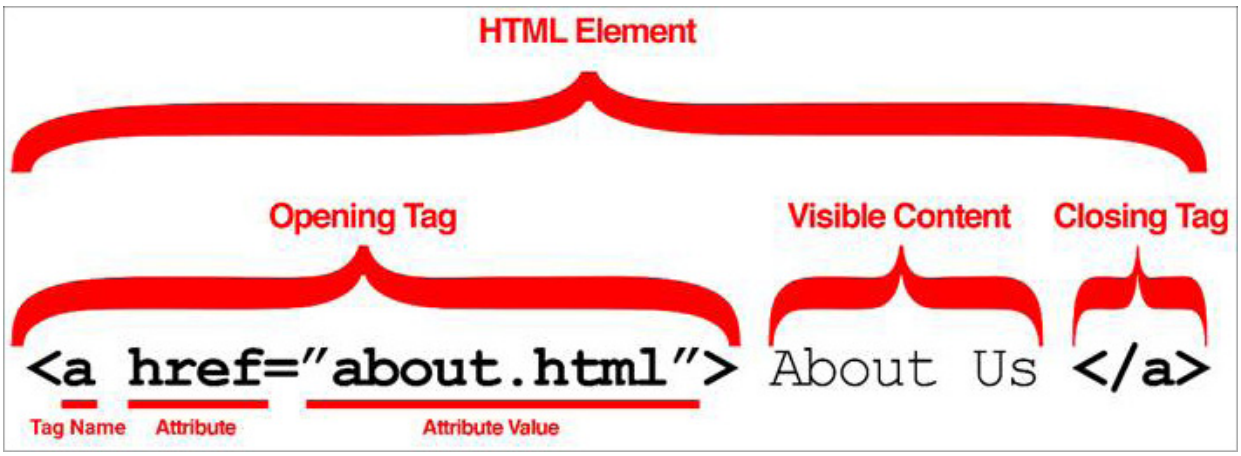


***Figure 4.2**: HTML element structure*

Working with text

Text in HTML is often contained inside specific tags that help format and present it. The simplest way to add text in an HTML document is to use paragraph (**<p>**) and heading tags (**<h1>** to **<h6>**). Let us take a look at how this works.

Headings

Let us add some of these elements to the web page. Start with the heading, using the **<h1>** tag. Here, in the following figure, we can see the **<h1>** element:

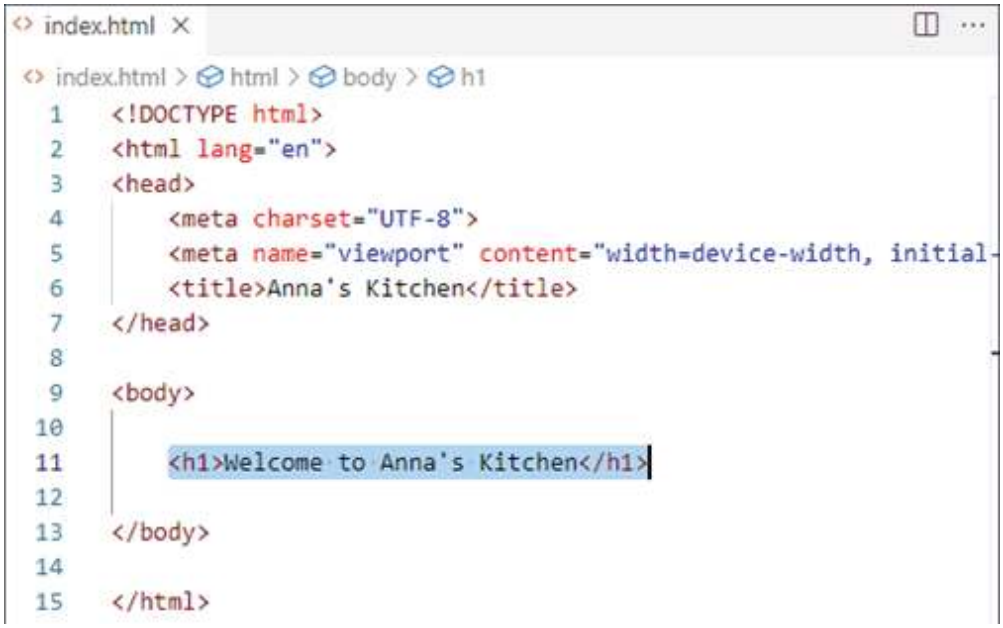


***Figure 4.3**: Adding headings*

This will render as follows in the web browser:



***Figure 4.4**: Page in browser*

In the preceding *Figure 4.4*, the HTML document is open in VS Code on the left. The same document is open in a web browser on the right, and the effect that each element has on the text can be seen, as indicated by the red arrows.

Browsers do not display the HTML elements, but use them to format the content of the page according to their function.

Paragraphs

You can also add paragraphs to your web page. Enclose each paragraph within opening and closing ** **...** ** tags. For example:

```
<p> Every day, our expert chefs prepare a mouth-watering feast of hand-carved meats including beef, turkey, pork and marmalade-glazed gammon. All accompanied by ruffled roasties, a wide range of veg, and giant Yorkshire puddings. Served up by our friendly team, every day of the week, our mouthwatering Sunday roast and weekday carvery are sure to be a hit with all the family. </p>
```

This paragraph will now be added to the website. As shown in *Figure 4.5*, the paragraph has been inserted beneath the heading. After saving the changes, the updated content will be rendered in the browser.



***Figure 4.5**: Adding paragraphs*

This will render as follows in the web browser:

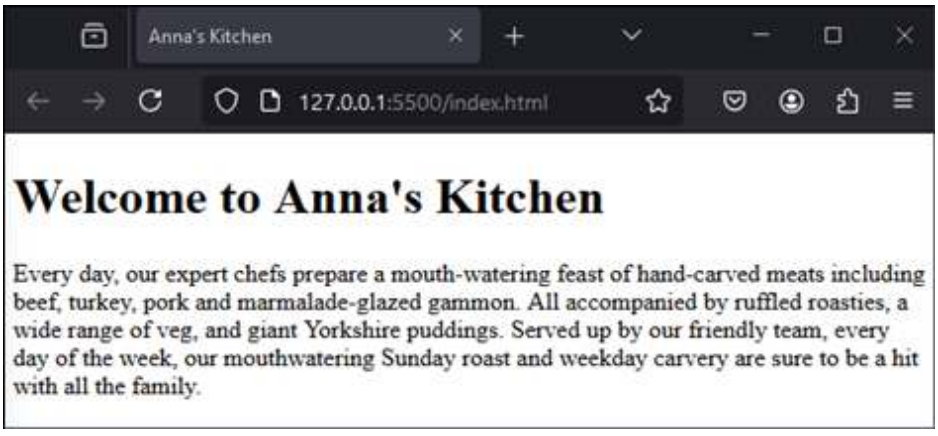


***Figure 4.6**: Paragraph*

Bold text

Text can also be emphasized using bold or strong formatting. This is achieved by enclosing the relevant word or phrase within the appropriate HTML tags. For example:

```
<b> beef, turkey, pork and marmalade-glazed gammon</b>
<strong> beef, turkey, pork and marmalade-glazed gammon
</strong>
```

Have a look at the example. In *Figure 4.7*, the **<b>** bold tags around the highlighted sentence have been added. You can see in the browser that this text appears bold:



***Figure 4.7**: Adding bold text*

This will render as follows in the web browser:



***Figure 4.8**: Bold text in paragraph*

The text **`beef, turkey, pork and marmalade-glazed gammon.`** appears in bold or strong text.

Italic text

You can also make text italic or emphasized, often used for idiomatic expressions, technical terms, or phrases in another language. The **`<i>`** tag marks text that is usually displayed in italic type. You can also use the **`<em>`** tag. The main difference is that the **`<em>`** tag is used to indicate emphasis. It also renders the text as italic by default, but its main purpose is to signal that the text is more important or emphasized, which can be useful for screen readers and other assistive technologies.

To use these tags, just surround the word or words with the **`<i>`** or **`<em>`** tags, as follows:

```
<i> Served up by our friendly team, every day of the week, our mouthwatering Sunday roast and weekday carvery are sure to be a hit with all the family. </i>
```

Let us add this to our website. It can be seen in *Figure 4.9* that italic tags around the sentence have been added, and you can see in the browser on the right that the text appears in italic type:

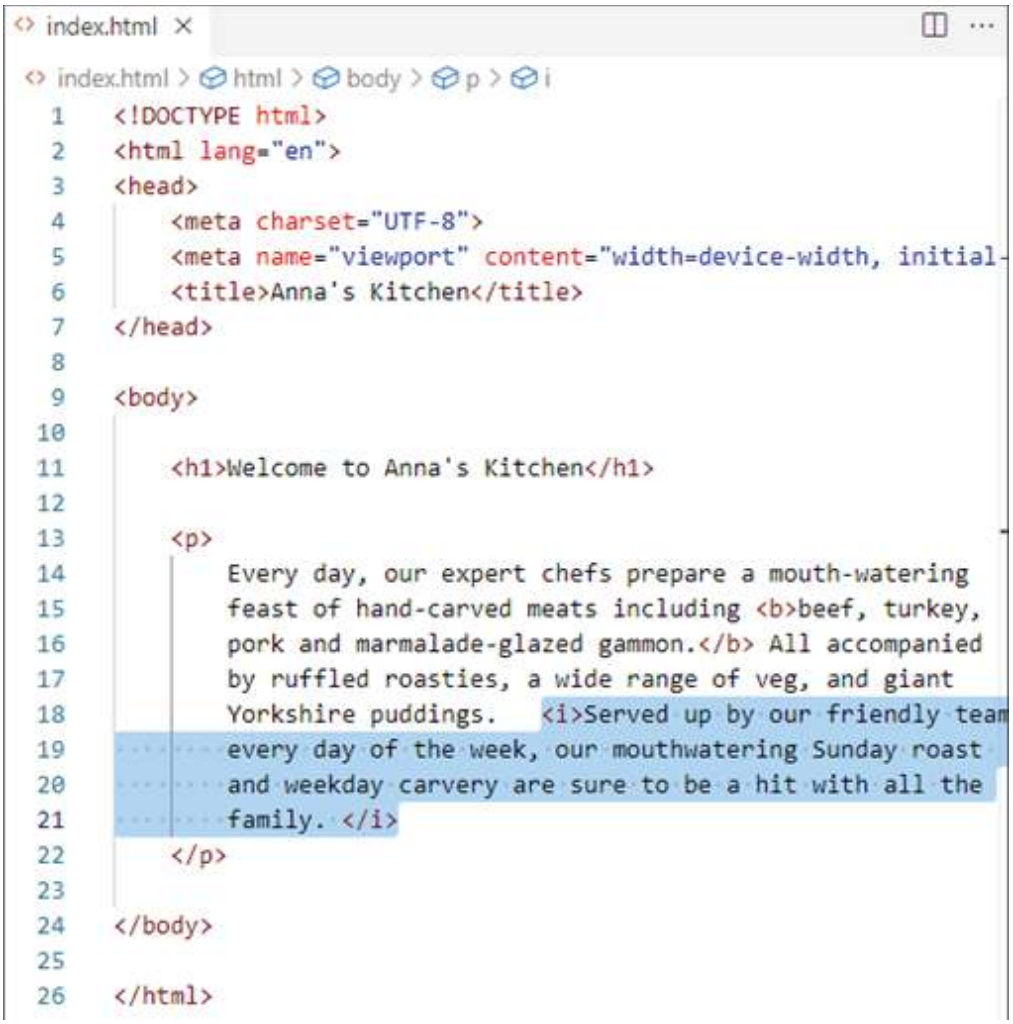


***Figure 4.9**: Adding italic text*

This will render as follows in the browser:



***Figure 4.10**: Italic text in paragraph*

Text color

To change the text color, add the **`style`** attribute to a style. Set the attribute to **`color`**, then select a color from the list of HTML colors, as follows:

`<H1 style = “color:Yellow;”>`

Let us add this to our website. We can add the style attribute to the opening **`<h1>`** tag in the heading. In *Figure 4.11*, the title font color has been changed to yellow:

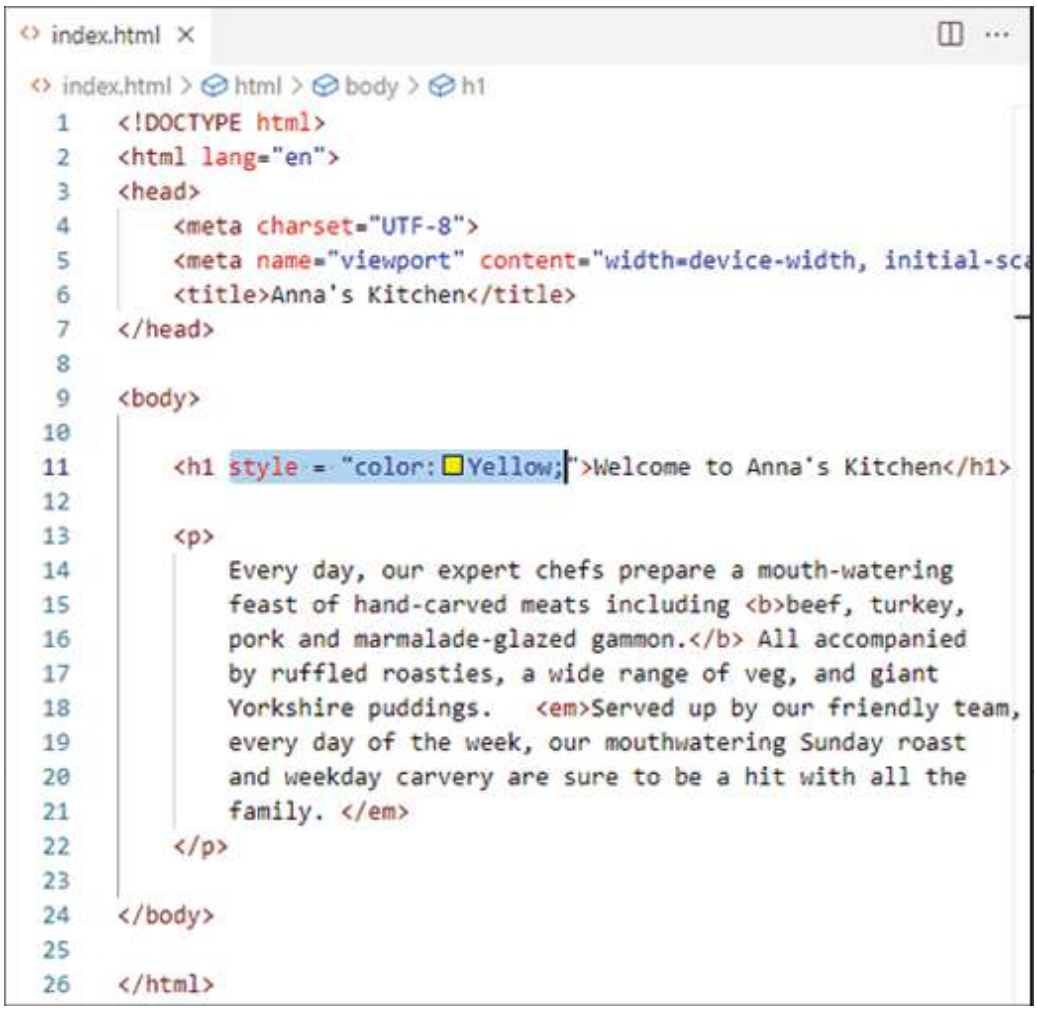


***Figure 4.11**: Changing text color*

This will render as follows in the browser:

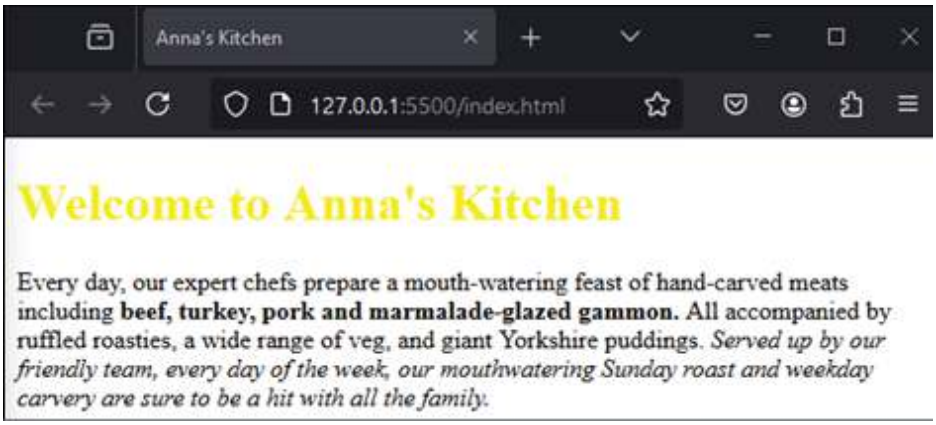


***Figure 4.12**: Heading text color*

Fonts

To change the font, add the **`style`** attribute to a style. Set the attribute **`font-family`** to the font name you want. In this example, Helvetica has been used:

`<h2 style = “font-family:Helvetica;”>The Home of the Roast!</h2>`

Let us add this line to the HTML file. In *Figure 4.13*, it can be seen that the font on the subheading has changed to **`Helvetica`**:



***Figure 4.13**: Using fonts*

This will render as follows in the browser:

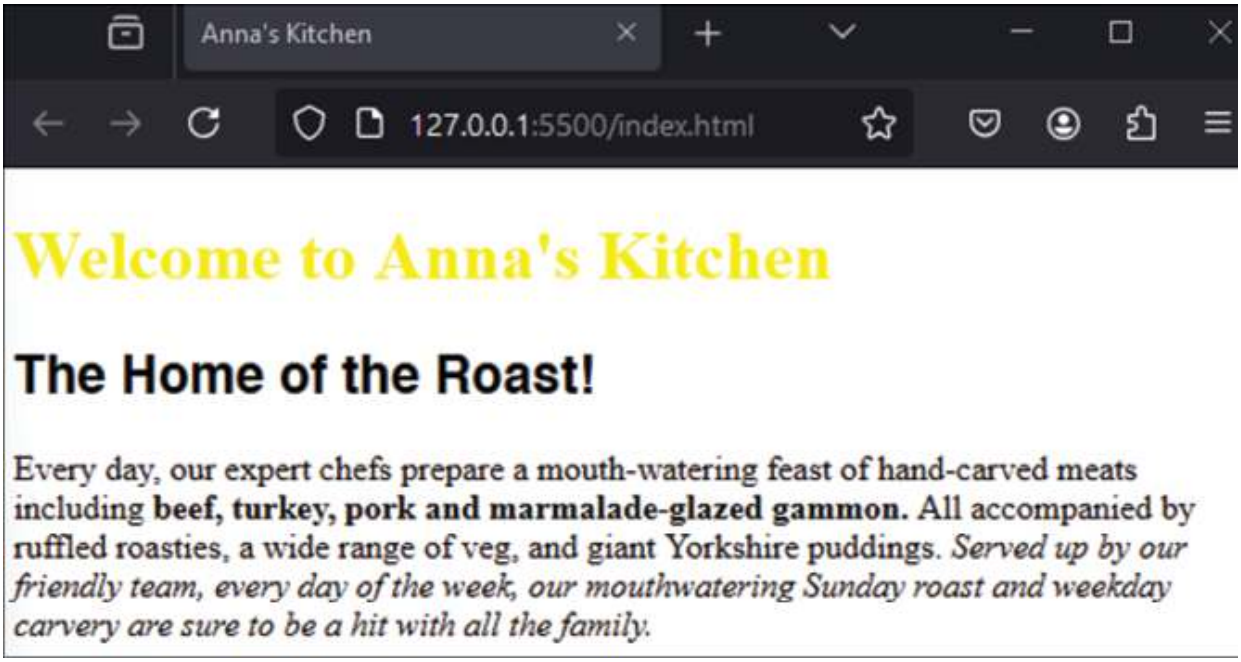


***Figure 4.14**: Font change*

You can choose from a variety of fonts. Not all of them are supported by all browsers, but most of them are. The following are some fonts that are available on most operating systems and devices:

- Arial
- Verdana
- Tahoma
- Georgia
- Times New Roman
- Courier New
- Trebuchet MS
- Lucida Sans

To use any of them, just add the name to the style attribute in the opening HTML tag. For example, to use **`Tahoma`** instead:

```
style = “font-family:Tahoma;”
```

Page background color

To change the background color on any object, add the **`style`** attribute. Set it to **`background-color`**, then choose a color from the HTML colors list, as follows:

`<body style = “background-color:Orange;”>`

Have a look at the code in *Figure 4.15*:



***Figure 4.15**: Changing the background color of the page*

This will render as follows in the browser:

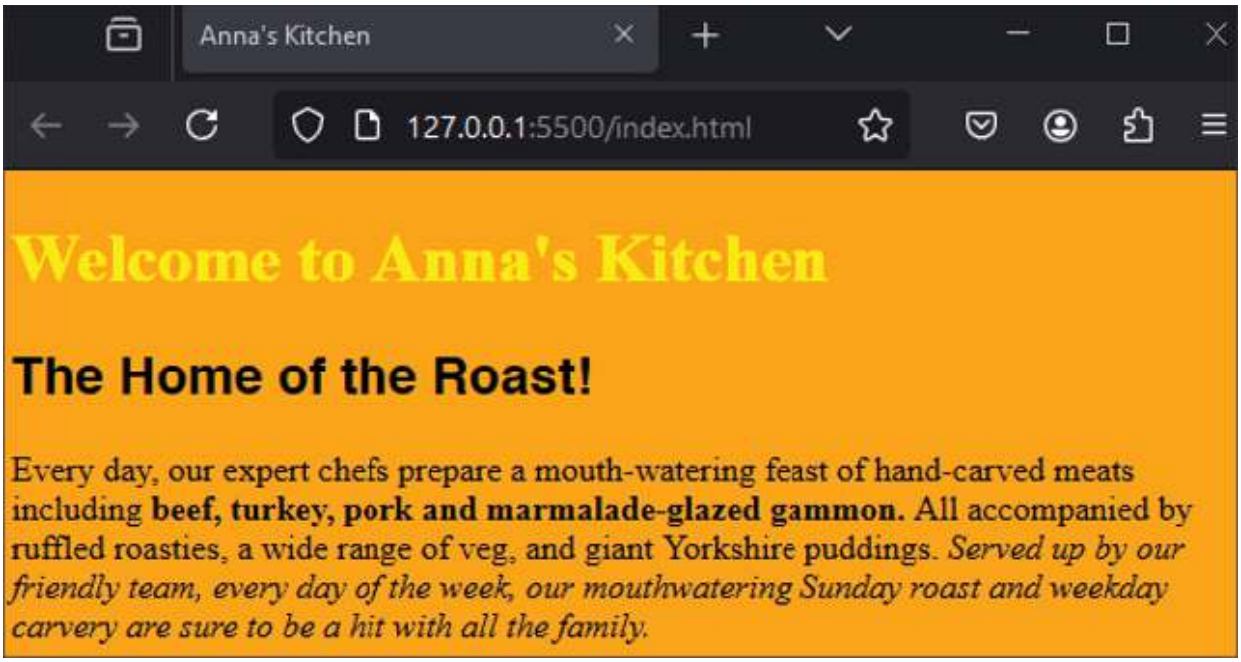


***Figure 4.16**: Background color*

You can also change the background color of other objects, such as the paragraph background, to:

```
<p style = “background-color:Gold”>
```

Let us add this to the website shown in *Figure 4.17*:



```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale
    <title>Anna's Kitchen</title>
</head>

<body style = "background-color:Orange;">

    <h1 style = "color:Yellow;">Welcome to Anna's Kitchen</h1>

    <h2 style = "font-family:Helvetica;"">The Home of the Roast!</h2

    <p style = "background-color:Gold">
        Every day, our expert chefs prepare a mouth-watering
        feast of hand-carved meats including <b>beef, turkey,
        pork and marmalade-glazed gammon.</b> All accompanied
        by ruffled roasties, a wide range of veg, and giant
        Yorkshire puddings.  <em>Served up by our friendly team,
        every day of the week, our mouthwatering Sunday roast
        and weekday carvery are sure to be a hit with all the
        family. </em>
    </p>

</body>

</html>
```

***Figure 4.17**: Changing background color of element*

This will render as follows:



***Figure 4.18**: Element background color*

Adding tables

Tables in HTML are used to present structured data in rows and columns. Whether displaying a product list, a menu, or a timetable, tables offer a logical layout for organizing information. This section introduces the basic tags used to construct tables and demonstrates how to create a simple, styled table for your web page.

To add tables, a basic template can be used. We start with the **`<table>`** tag.

Then, to add rows, we use the **`<tr>`** tag, which stands for **table row**.

To add each cell, we use the **`<td>`** tag, which stands for **table data**. The top row of the table usually contains headings for each column, so we use the **`<th>`** tag for this, which stands for **table header**. We can see how the table is constructed and how it appears when rendered in a web browser in *Figure 4.19*:

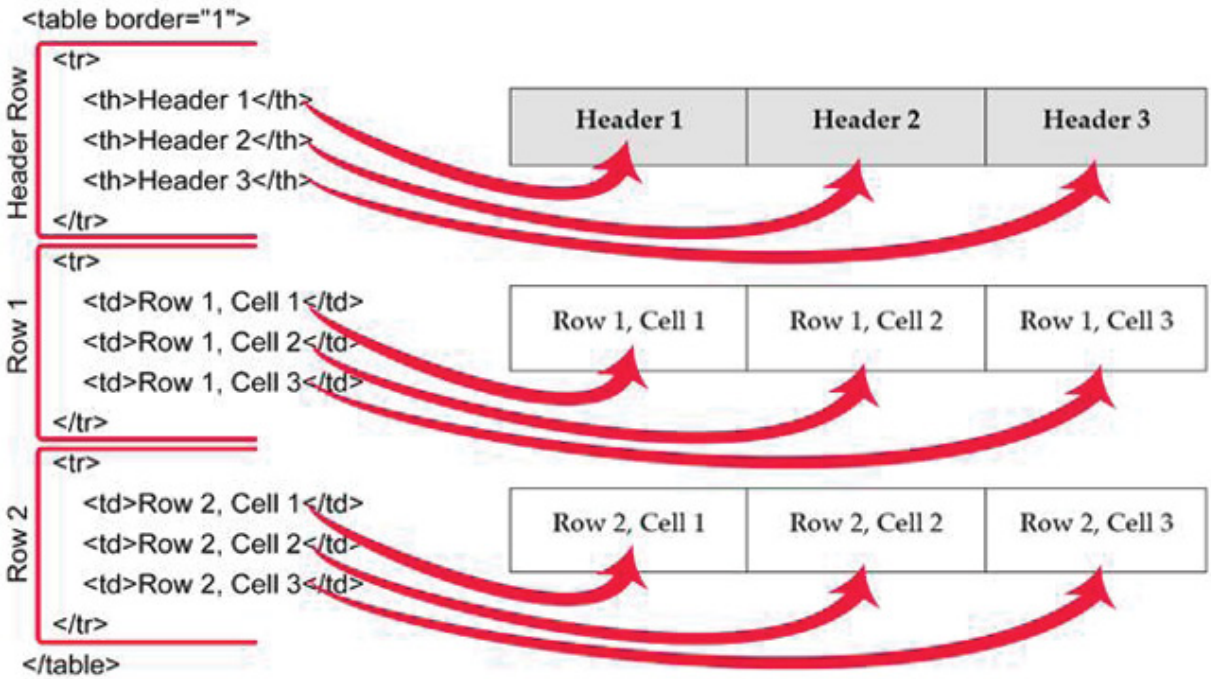


***Figure 4.19**: Table construction*

Now that we know how to create tables, let us add today's menu as a table to our web page. We will change the background color of the table to gold and set the border to 1, as:

```
<table style = "background-color:gold" border="1" >
```

Next, we add our rows starting with the **<tr>** tag, as:

```
<tr>
```

Now, the first row is the header, and we want to add two cells, one for the dish and one for the price. These cells will create the columns in our table. For each cell, we want to pad out the content so it is not so close to the border. We can do this by adding some padding to the style attribute. In this case, we are adding 10 pixels of padding around the cell, as:

```
<th style="padding: 10px;">Dish</th>
<th style="padding: 10px;">Price</th>
```

Once we have added the data, we end the row with a closing **</tr>** tag, as:

```
</tr>
```

Once the first row is done, we can do the same with the other rows. Notice that with the cells in the remaining rows, we use the **<td>** tag:

```
<tr>
    <td style="padding: 10px;">Roast Beef</td>
    <td style="padding: 10px;">$15</td>
</tr>
<tr>
    <td style="padding: 10px;">Turkey &
Stuffing</td>
    <td style="padding: 10px;">$14</td>
</tr>
<tr>
    <td style="padding: 10px;">Gammon &
Marmalade</td>
    <td style="padding: 10px;">$13</td>
</tr>
```

Let us add this to our website. In *Figure 4.20*, the table has been added to the bottom of the web page under an **<h3>** heading:



***Figure 4.20**: Adding a table to our HTML page*

This will render as follows in the web browse:

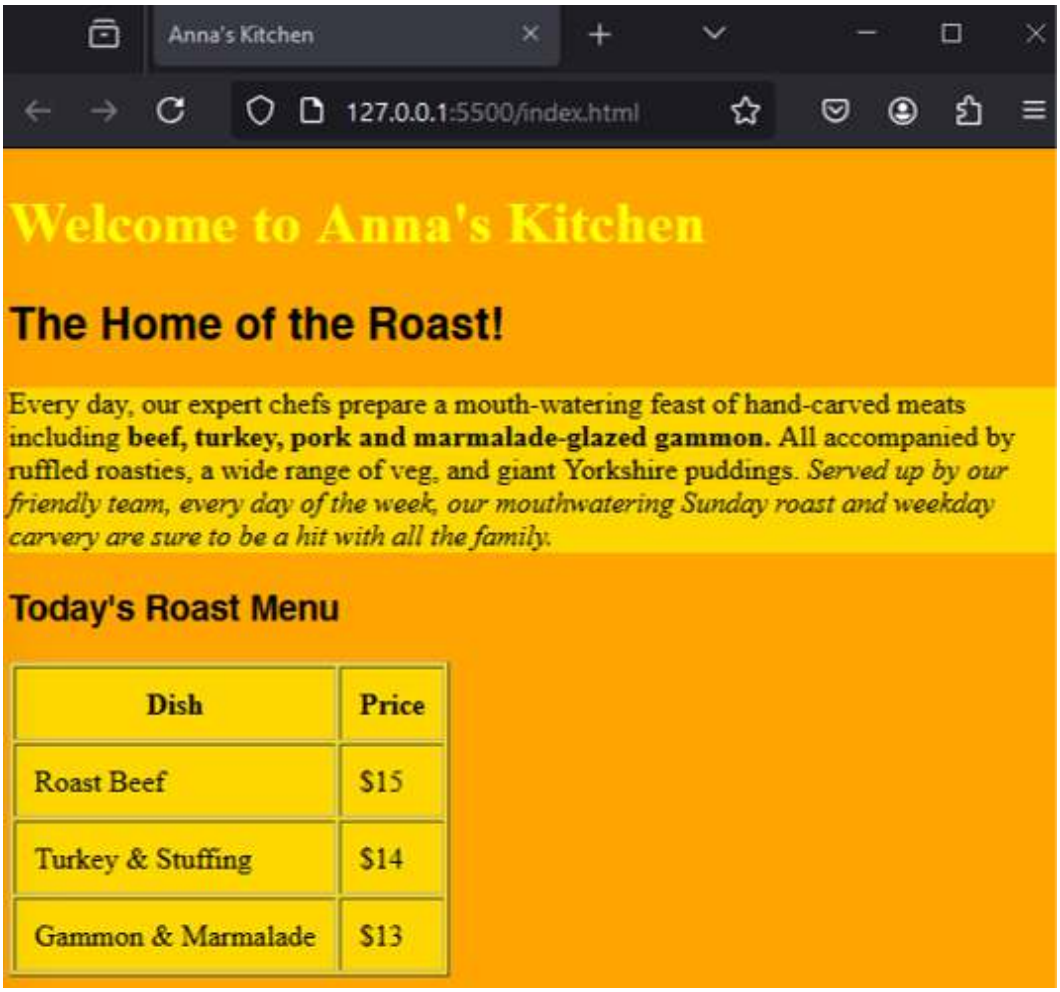


***Figure 4.21**: Table rendered in browser*

Adding hyperlinks

Adding hyperlinks to a web page is done using the anchor tag, which creates clickable links for the user to click on, as:

```
<a href= "url to page"> Link name to display on page </a>
```

The `<a>` tag creates a hyperlink, as:

```
<a href="/menu">Click here for full menu</a>
```

In this case, it links to the `"/menu"` page. The link will be displayed as the text `Check out our full menu here`.

Let us add this link under the menu. As shown in *Figure 4.22*, a link to the `menu.html` page has been added. In the browser window, you can see that

`Click here for full menu` is now a clickable link. If you click on this link now, it will not work until we create the **`menu.html`** file:



Figure 4.22*: Adding a hyperlink*

This will render as follows in the web browser:



***Figure 4.23**: Hyperlink in browser*

To create the **`menu.html`** file, in the explorer on the left-hand side, right-click, then select **New File** from the menu, as shown in *Figure 4.24*:

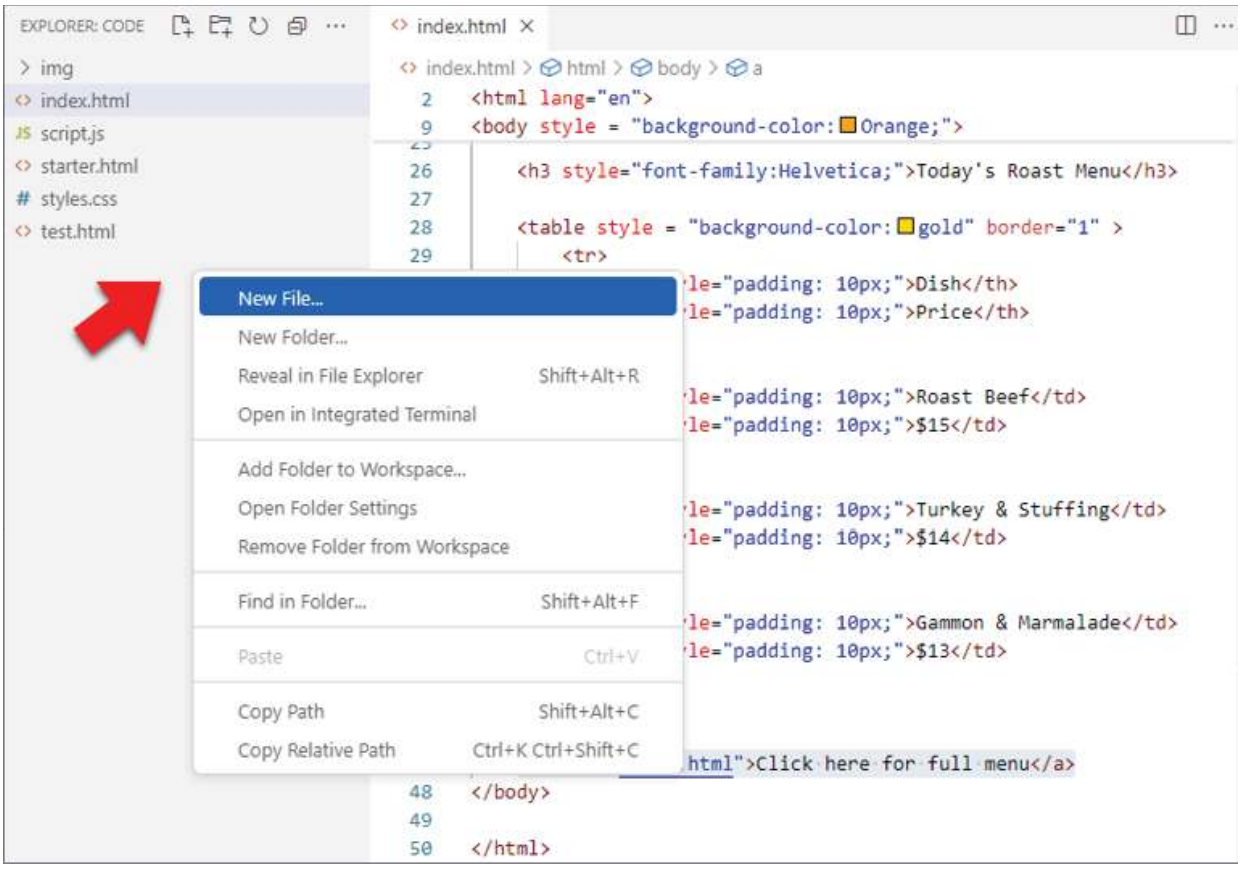


***Figure 4.24**: Creating a new file*

Enter the file name: **`menu.html`**, as shown in *Figure 4.25*. Now, you can add the code to the menu page:

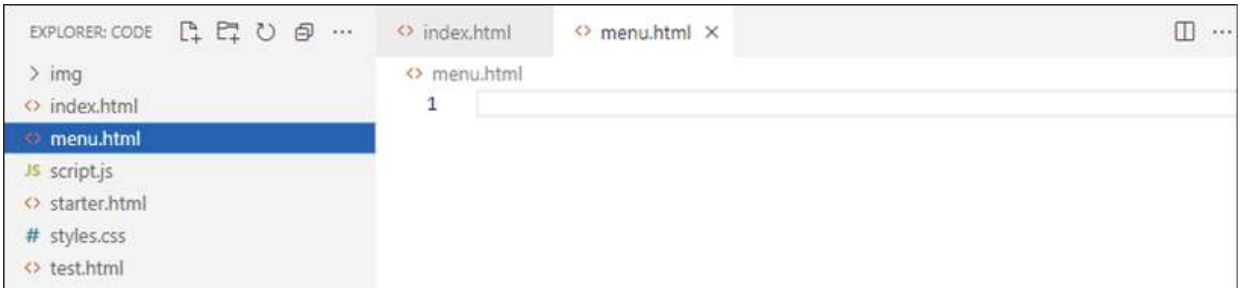


***Figure 4.25**: Naming the new file*

Ordered lists

An ordered list is a collection of items that are numbered sequentially. We use the **`<ol>`** element to create an ordered list, and each item within it is

defined using the **<li>** (list item) tag, as follows:

```
<ol>
  <li>First item</li>
  <li>Second item</li>
  <li>Third item</li>
</ol>
```

This will show as:

1.First item
2.Second item
3.Third item

You can customize the numbering style using the type attribute as follows:

- **type="1"** is decimal numbering (default).
- **type="A"** for uppercase letters (A, B, C, ...).
- **type="a"** for lowercase letters (a, b, c, ...).
- **type="I"** for uppercase Roman numerals (I, II, III, ...).
- **type="i"** for lowercase Roman numerals (i, ii, iii, ...).

So, in order for the numbering to use letters instead of numbers, the attribute **type="A"** needs to be added to the opening tag, as:

```
<ol type="A">
  <li>First item</li>
  <li>Second item</li>
  <li>Third item</li>
</ol>
```

This will show as:

A.First item
B.Second item
C.Third item

Unordered lists

An unordered list is a collection of items that are not numbered or sequential. We use the **<ul>** element to create an unordered list, and like

ordered lists, each item is defined using the **<li>** tag. Unordered lists display items with bullet points by default:

```
<ul>
  <li>First item</li>
  <li>Second item</li>
  <li>Third item</li>
</ul>
```

This will render as:

- **First item**
- **Second item**
- **Third item**

You can change the bullet style using the type attribute as follows:

- **type ="disc"** for filled circle (default).
- **type="circle"** for hollow circle.
- **type="square"** for square bullets.

So, to make a hollow circle instead of a solid bullet, the attribute **type="circle"** needs to be added to the opening tag, as:

```
<ul type="circle">
  <li>First item</li>
  <li>Second item</li>
  <li>Third item</li>
</ul>
```

This will render as:

- **First item**
- **Second item**
- **Third item**

Nesting lists

Nesting means placing one list inside another list. This can be done with both ordered and unordered lists, or even a combination of both.

In the example, we have an ordered list (**<ol>**) for the main items. Then,

inside the second list item, we have added another ordered list (highlighted in bold), as:

```
<ol>
  <li>Main item 1</li>
  <li>Main item 2
    <ol>
      <li>Sub-item 1</li>
      <li>Sub-item 2</li>
    </ol>
  </li>
  <li>Main item 3</li>
</ol>
```

This will render as:

1. **Main item 1**
2. **Main item 2**
   1. **Sub-item 1**
   2. **Sub-item 2**
3. **Main item 3**

Conclusion

HTML elements and tags form the foundation of all web content, providing the structure and semantics that browsers use to render pages accurately. In this chapter, you have gained practical experience using essential elements such as headings, paragraphs, semantic text tags, tables, hyperlinks, and various types of lists. You have also explored the use of inline styles to control font families, text color, and background presentation—skills that are vital for shaping both the structure and appearance of your content.

By mastering these core techniques, you are now equipped to create well-organized, readable, and semantically meaningful web pages. You have also seen how these elements can be combined to build practical layouts, such as a restaurant menu page, using real-world examples. Understanding how to correctly structure and style HTML content lays a critical foundation for all future web development work.

In the next chapter, you will extend your skills by learning how to embed multimedia content into your web pages. This includes adding static images to visually enhance your content, embedding audio files for background sound or narration, and incorporating videos to deliver rich, interactive media experiences. These elements will enable you to create more dynamic and engaging websites that go beyond text and layout, appealing to a wider range of users and use cases.

Points to remember

- HTML elements and tags structure and present content on a web page.
- Tags typically include an opening tag, content, and a closing tag (e.g., ...).
- Headings are created using <h1> to <h6>, with <h1> being the largest and <h6> the smallest.
- Paragraphs are defined using the tag to add text blocks.
- Links are created with the <a> tag, and attributes like href specify the destination URL.
- Tables use <table>, <tr>, <th>, and <td> tags to organize data into rows and columns.
- The <b> and <i> tags make text bold and italic, while and emphasize content.
- Inline styles, like style=color:red, can be added to change text or background colors.

Exercises

1. Which HTML tag is used to create the largest heading on a web page?
2. Write the HTML code to create a paragraph that contains the text: *Welcome to our restaurant!*
3. Using the anchor (<a>) tag, create a link that points to menu.html and displays the text *View our menu*.
4. Write the HTML code to create a table with two columns: *Dish* and

Price.

5. What HTML tag would you use to make text appear in italic format?
6. How do you create ordered and unordered lists?

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com



CHAPTER 5

Images, Video, and Audio

Introduction

This chapter explores how to enhance a website by incorporating multimedia elements that enrich the user experience and increase engagement. While previous chapters have focused on structuring content using text and layout techniques, here you will learn how to add meaningful visual and auditory elements to bring your web pages to life.

You will begin by learning how to add images using the <img> tag, including best practices for accessibility through the use of the alt attribute and performance improvements through image optimization and lazy loading. You will explore common image formats such as JPEG, PNG, and WebP, and understand when and why to use each format depending on the content.

Next, you will discover how to embed video files using the <video> tag. You will learn how to configure playback controls, adjust size and layout, and use attributes such as autoplay, muted, and controls to deliver a user-friendly experience. Considerations for bandwidth, file size, and hosting options such as *YouTube*, *Vimeo*, and *CDNs* will also be discussed.

Finally, you will learn how to embed audio files using the <audio> tag and understand how to incorporate sound effectively without overwhelming or distracting users.

Structure

The chapter covers the following topics:

- Adding images
- Embedding videos
- Embedding audio files

Objectives

By the end of this chapter, you will be able to add images and embed videos and audio into an HTML page.

Adding images

To add an image, we use the **`<img>`** tag. In the opening tag, you add the location of the image to the source attribute. You should also add alternative text for accessibility:

```
<img src="path-to-your-image " alt="Description of the image">
```

In this line, the **`src`** attribute specifies the path or URL to the image you are adding to the web page.

The alt attribute provides an alternative text description of the image. This text is displayed if the image fails and is used by accessibility devices, such as screen readers, that use this text to describe the image to visually impaired users. In *Figure 5.1*, an image has been added to the first paragraph:

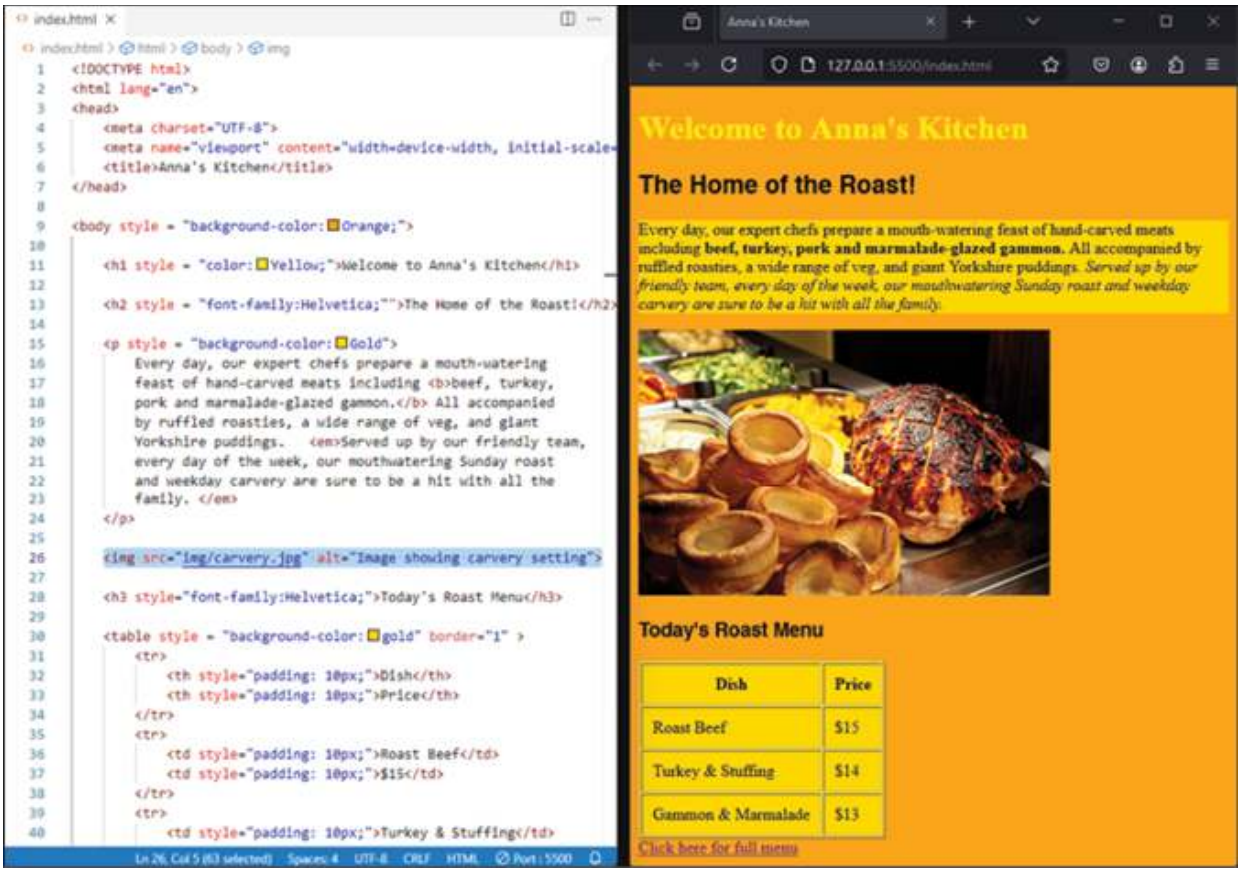


***Figure 5.1**: Adding an image*

Now, when you add images, you should make sure they are optimized and sized correctly. Scale the images to the correct size in an image editor and save them optimized for the web. You can also use tools, such as *TinyPNG* or *ImageOptim*, to compress the images to reduce the file size without sacrificing too much quality.

Common image formats include the following:

- JPEG, which is best for photographs and images with lots of colors.
- PNG can also be used for photographs, but is best for images needing transparency or sharp edges (logos, icons).
- WebP is best for modern web usage (smaller, high-quality images).

You can also size images using the width and height attributes directly in the **<img>** tag, but this method is less flexible and does not optimize the image:

```
<img src="carvery.jpg" width="400" height="259" alt="Example Image">
```

Finally, if you have a lot of images on the web page, you should consider the

lazy loading feature. **Lazy loading** is an optimization technique that defers the loading of images until they are about to enter the viewport. However, do not lazy load the images that are immediately visible when the page loads. This is known as **above the fold** and refers to the part of a web page that is visible without any scrolling down when the page initially loads. **Below the fold** refers to the part of a web page that is only visible after the user scrolls down. We can see how this works in *Figure 5.2*. The first image is loaded, while the others that are further down the page are not loaded until the user scrolls down the page:



***Figure 5.2**: Demonstrating lazy loading*

This technique helps to improve performance and reduce initial load times of larger web pages. So, except for the images that appear above the fold, we can change the loading attribute to lazy using the following code:

```
<img src="image2.jpg" width="400" height="259" alt="Second Image" loading="lazy">
```

Embedding videos

You can add videos to a web page using the built-in **<video>** tag. However, keep in mind that videos are large, so your web server will need to be configured to handle video. Serving video content can consume significant bandwidth, especially if the site experiences high traffic. To get around this problem, you could use a **content delivery network** (**CDN**), such as *Akamai*, *CloudFlare*, or *Amazon CloudFront*, or a video hosting service, such as *YouTube* or *Vimeo*, to serve your videos.

There are various video formats that you could use, however, MP4 (H.264) seems to be the most common. You could also use WebM, MOV, and HEVC (H.265).

To insert a video directly into the web page, use the following code:

```
<video>
  <source src="video.mp4" type="video/mp4">
  Your browser does not support the video tag.
</video>
```

We can resize the video using the height and width attributes, and also autoplay the video. Keep in mind that most browsers block autoplay videos that contain sound, but if you mute the video, it will autoplay without requiring user interaction. This is a best practice to avoid unwanted audio playback:

```
<video width="400" height="223" autoplay muted>
  <source src="video.mp4" type="video/mp4">
  Your browser does not support the video tag.
</video>
```

If you want the user to be able to control the video themselves, use the controls attribute, as follows:

```
<video width="400" height="223" controls>
  <source src="video.mp4" type="video/mp4">
```

```
  Your browser does not support the video tag.
</video>
```

Let us add a video to our website. In *Figure 5.3*, a video under the first paragraph has been added. It has been resized using the height and width attributes. The video has also been set to autoplay and muted, so it starts playing as the page is loaded:

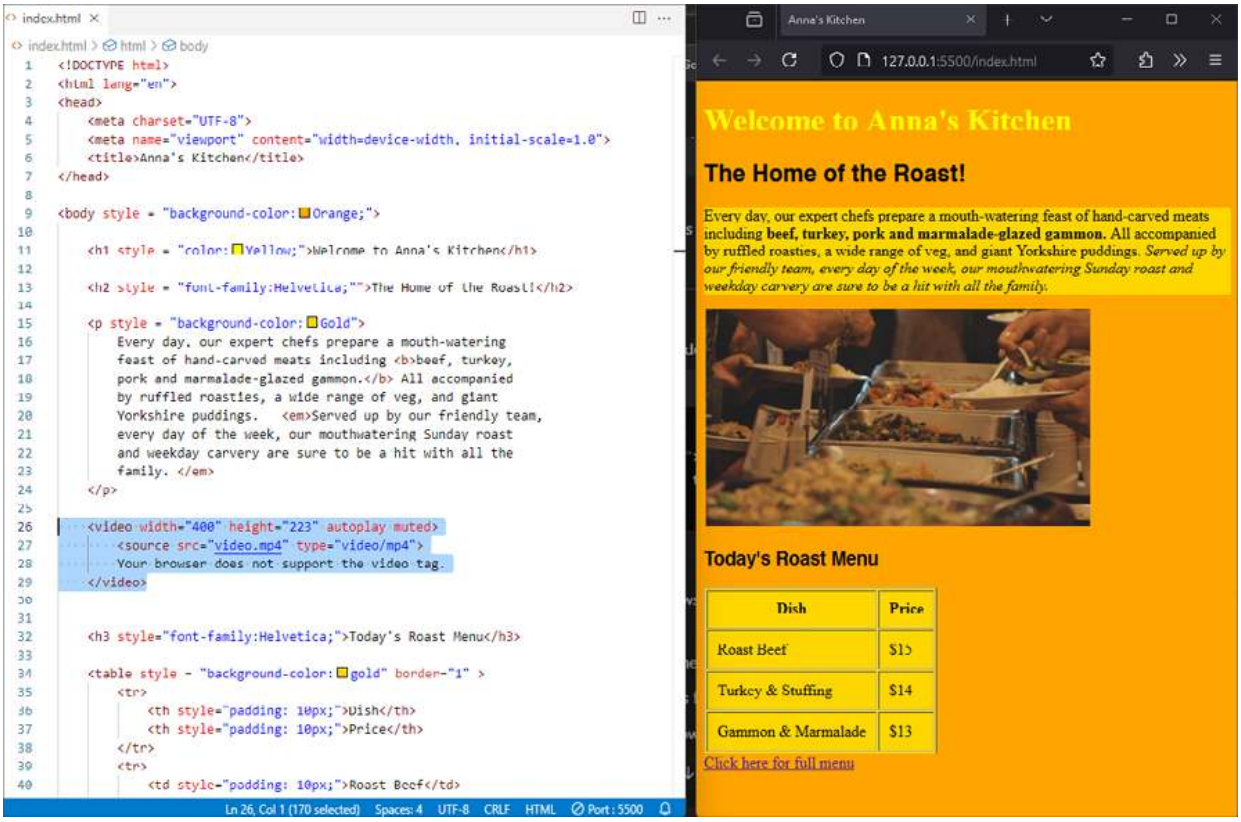


***Figure 5.3**: Embedding video*

Embedding audio files

To embed audio files, you can use the **`<audio>`** tag. This provides native support for audio playback and control. Keep in mind that adding audio to a web page can enhance the user experience, but it should be done carefully to avoid distraction and irritability. Only add audio to your site when it actually enhances the user experience. For example, this could be a podcast related to the content on your web page, as follows:

```
<audio controls>
```

```
  source src="audiofile.mp3" type="audio/mpeg">
  Your browser does not support the audio element.
</audio>
```

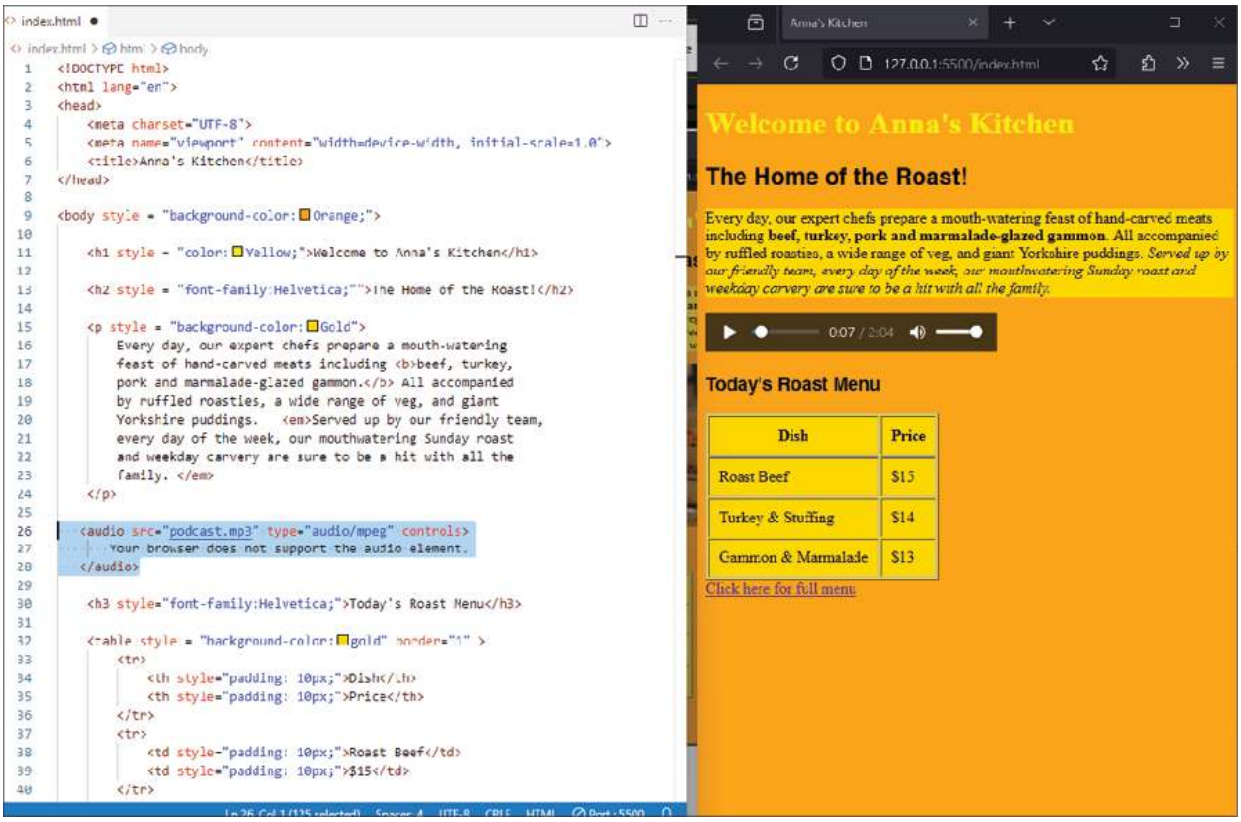


***Figure 5.4**: Embedding audio*

Conclusion

In this chapter, you explored how to enhance your web pages by incorporating multimedia elements—images, videos, and audio—that enrich user engagement and provide a more dynamic browsing experience. You learned how to add images using the <img> tag, apply best practices for accessibility with alt attributes, and optimize performance through techniques such as compression and lazy loading. Various image formats were discussed, including JPEG for photographs, PNG for transparent images, and WebP for modern, efficient delivery.

You also looked at how to embed video content using the <video> tag, including how to adjust playback behavior with attributes like autoplay,

muted, and controls. Considerations for hosting and bandwidth management—such as using content delivery networks or third-party services like YouTube—were highlighted to ensure scalability and smooth playback. Additionally, you learned how to embed audio using the <audio> tag, along with guidelines for incorporating sound in a way that complements rather than distracts from the content.

Carefully integrating multimedia with appropriate formatting, accessibility features, and performance strategies is vital for creating professional, user-friendly web experiences.

In the next chapter, you will begin working with HTML forms, which allow users to input data and interact with your site. You will learn how to create forms, use various input types, add labels, group elements, and implement both form processing and validation techniques to collect and manage user input effectively.

Points to remember

- Always use the alt attribute in the <img> tag to provide descriptive text for accessibility and a fallback in case the image fails to load.
- Optimize images by compressing and resizing them before adding them to the web page to improve load times.
- Choose the appropriate image format depending on its use:
  - JPEG for photographs and images with many colors.
  - PNG for images with transparency or sharp edges.
  - WebP for modern web usage with smaller file sizes and high quality.
- Consider using lazy loading (loading=lazy) for images below the fold to reduce initial page load time.
- Avoid using width and height attributes directly in the <img> tag for scaling; instead, use CSS to ensure flexibility and responsiveness.
- Use external video hosting services or a CDN for serving large videos to reduce server load and bandwidth consumption.
- Include the controls attribute in <video> and <audio> tags to allow user control over media playback.

- Avoid autoplay videos with sound, as this can disrupt user experience. Mute videos when setting them to autoplay.
- Embed audio content only when it enhances the user experience and is relevant to the web page content.
- Utilize semantic HTML tags (<figure>, <figcaption>, etc.) to group multimedia elements with descriptive captions for improved structure and context.

Exercises

1. Use the <img> tag to add an image to a web page with both src and alt attributes.
2. Explain the difference between JPEG, PNG, and WebP image formats and when to use each.
3. What is lazy loading, and how can it be implemented for images using the <img> tag?
4. Add an image to a web page and set its width and height using attributes in the <img> tag. Why might this not be the most optimal method?
5. Describe the benefits of using a CDN or video hosting service for embedding videos.
6. How do you embed a video using the <video> tag, and what attributes can be added to control playback behavior?
7. Insert a video into an HTML document that autoplays and is muted by default. Why is it recommended to mute videos that autoplay?
8. When would you use the controls attribute in the <video> tag, and what functionality does it provide?
9. Explain the role of the <audio> tag in embedding audio files. What are the best practices for using audio on a web page?
10. Add an audio file to a web page using the <audio> tag and enable user playback controls. Why is it important to use audio sparingly on web pages?

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com



CHAPTER 6

HTML Forms: Collecting User Input

Introduction

This chapter covers the essentials of building and structuring forms in HTML for effective user input collection. It starts with creating basic forms, exploring various input types like text, email, password, and date. Readers will learn how to organize forms using fieldsets and legends, add meaningful labels and placeholders, and understand form submission processes. The chapter also covers HTML5 validation, helping ensure data accuracy without relying solely on JavaScript.

Structure

The chapter covers the following topics:

- Introduction to HTML forms
- Creating a form
- Input types
- Adding labels
- Grouping form elements

- Form processing
- HTML form validation

Objectives

By the end of this chapter, you will be able to construct basic HTML forms with various input types, labels, and grouped elements. You will also learn how to process and validate the form's input.

Introduction to HTML forms

Forms allow users to submit data to a server. On most websites, you will find various forms, such as a login page, registration page, or search bar. *Figure 6.1* is an example of a login form on a website:

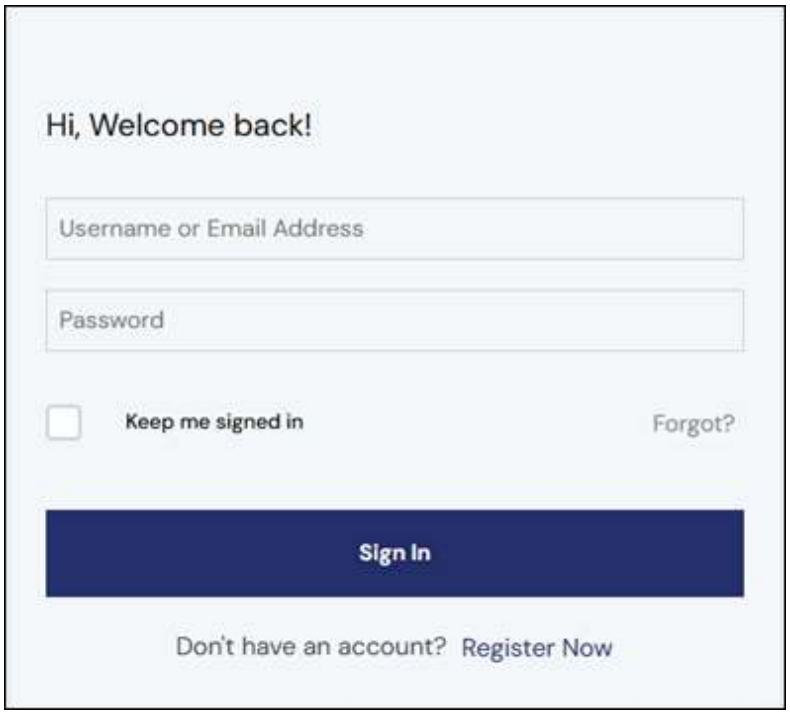


***Figure 6.1**: Login form*

You might also see registration and contact forms. These allow users to sign up for a service on a website using the registration form or send the business a message using a contact form, as shown in *Figure 6.2*:

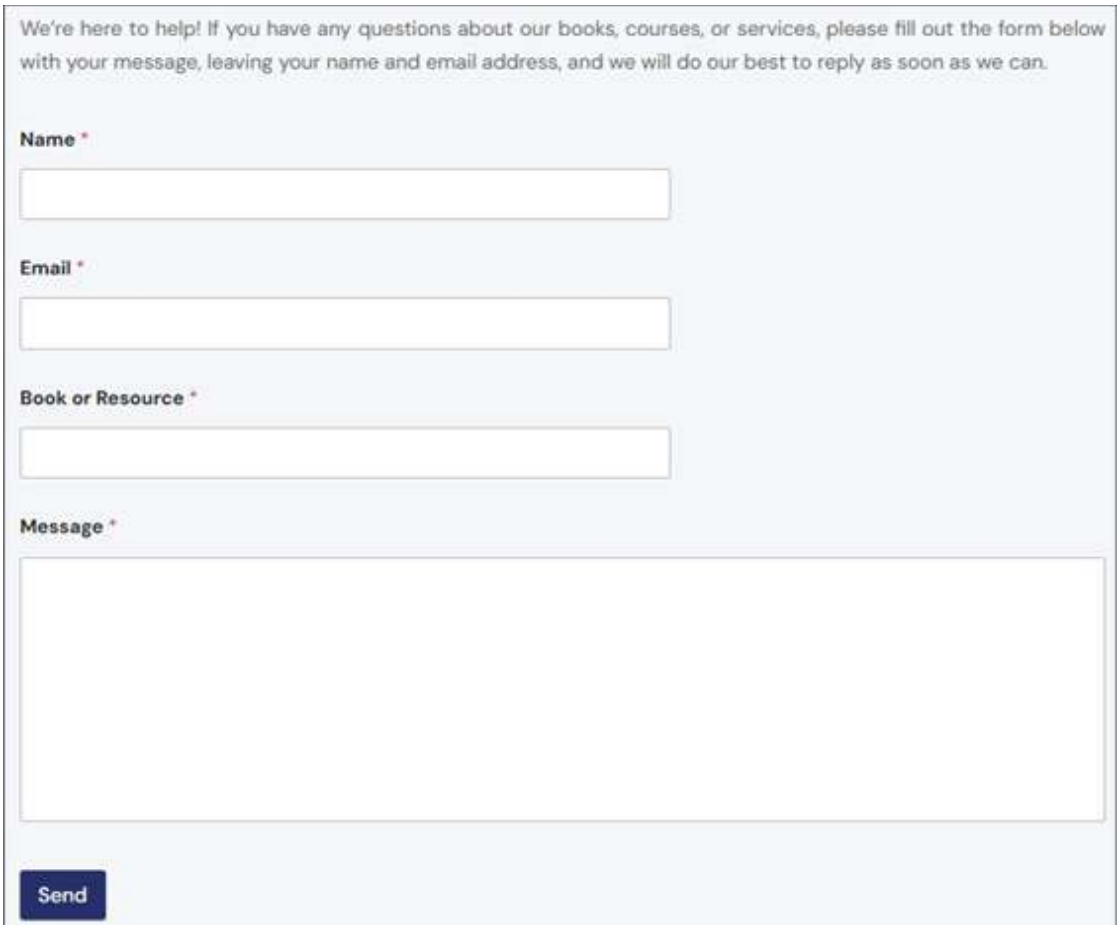


***Figure 6.2**: Registration form*

When submitting data using a form, there are two submission methods: GET and POST. These two methods determine how the data is sent to the server.

The GET method appends form data as a query parameter to the URL. It is commonly used for non-sensitive data submissions, such as search queries or filtering options, where data visibility is not a concern. Refer to the following figure:

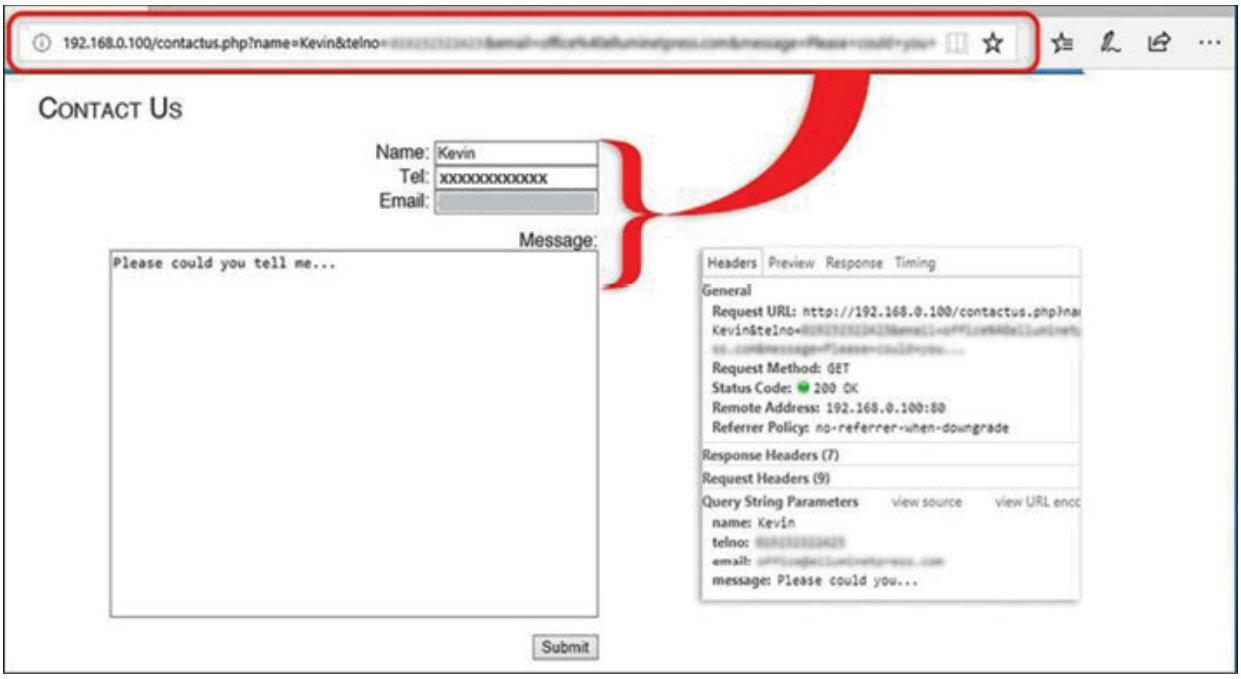


***Figure 6.3**: GET method*

The POST method sends form data as a separate HTTP message body, which means the data is not visible in the URL. This makes POST ideal for submitting sensitive or large data, such as login credentials or form submissions that modify server-side resources. Refer to the following figure:

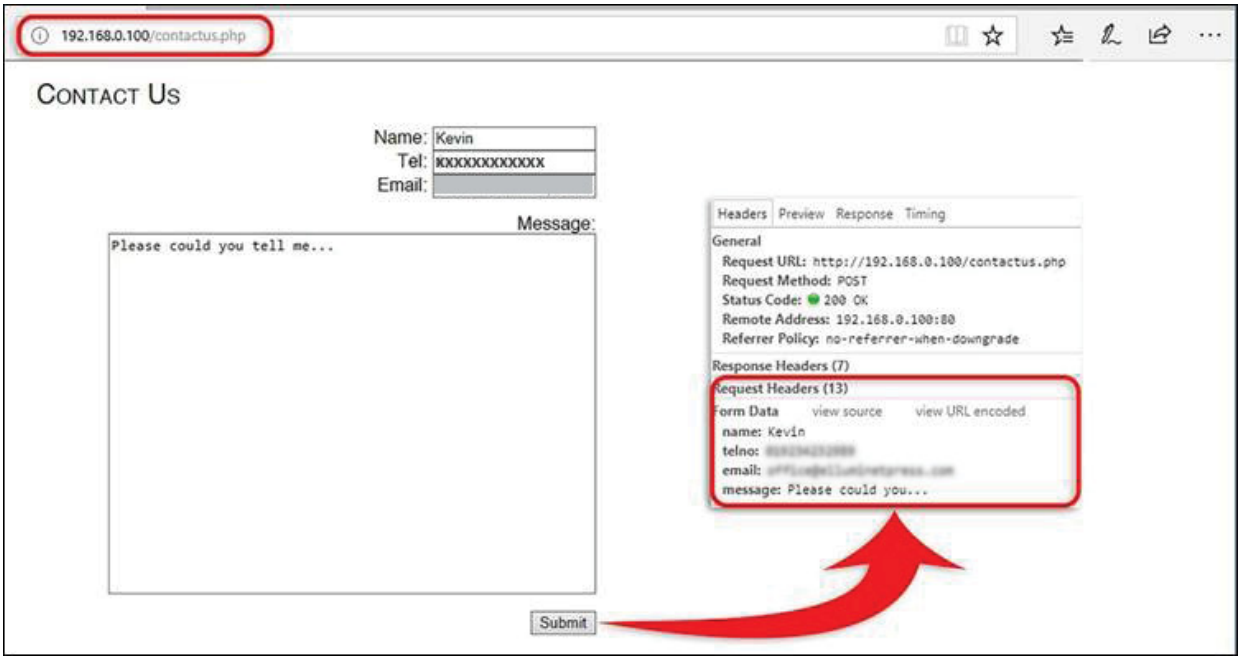


***Figure 6.4**: POST method*

The data from the form is processed by a script on the web server. This script can be written in a programming language, such as Node.js, Python, PHP, or C#, depending on how the backend of the website has been developed. In most cases, the script receives the form data using the POST method. The data received from the form can be stored in a database or sent as an email.

Creating a form

To create a basic form, we use the **<form>** element. This element contains all the form fields and defines a submission method using the action attribute, as follows:

```
<form action="script_to_process_data" method="POST">
…
</form>
```

Let us build a form for our website using the following steps:

1. Create a new file called **book.html**.
2. In the blank document, press **!**, then tab to create a blank HTML template, as shown in the following figure:



***Figure 6.5**: Creating an HTML template with Emmet*

3. Next, in the body section of the HTML document, add a heading **<h1>** and an instructional paragraph using the **<p>** tags, as shown in the following figure:

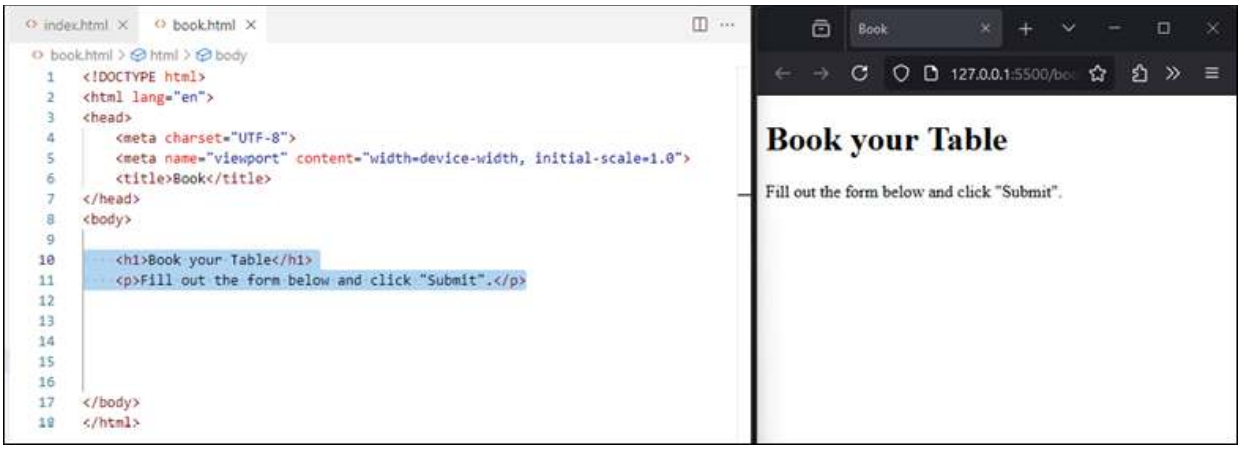


***Figure 6.6**: Adding a heading and paragraphs*

In the next few sections, we can construct a simple booking form.

Input types

HTML forms use various input types to collect user data. The most common are as follows:

- Text field
- Password field
- Email field
- Number field
- Tel field
- URL field
- Checkbox
- Radio button

These are known as **input types**. To add these to the form, we include them between the **`<form>…</form>`** tags.

Let us add a text field on the website to allow the user to enter a name. To do this, we use the **`<input>`** element as follows:

```
<input type="text" id="name" name="name" placeholder="Enter your name">
```

We also need the ID, name, and placeholder attributes. The ID is used to identify the input element within the document uniquely and can be used for

linking labels or referencing in scripts. The name attribute is used to specify the name of the input and is used to identify the data when processing the form. The placeholder attribute allows you to add temporary text to the field as a prompt for the user as to what data to enter.

We can also add an email field for an email address. This will check to see if the user has entered an email address, as follows:

```
<input type="email" id="email" name="email" placeholder="Enter your email">
```

In the following figure, two fields have been separated using the break line tag **
**:

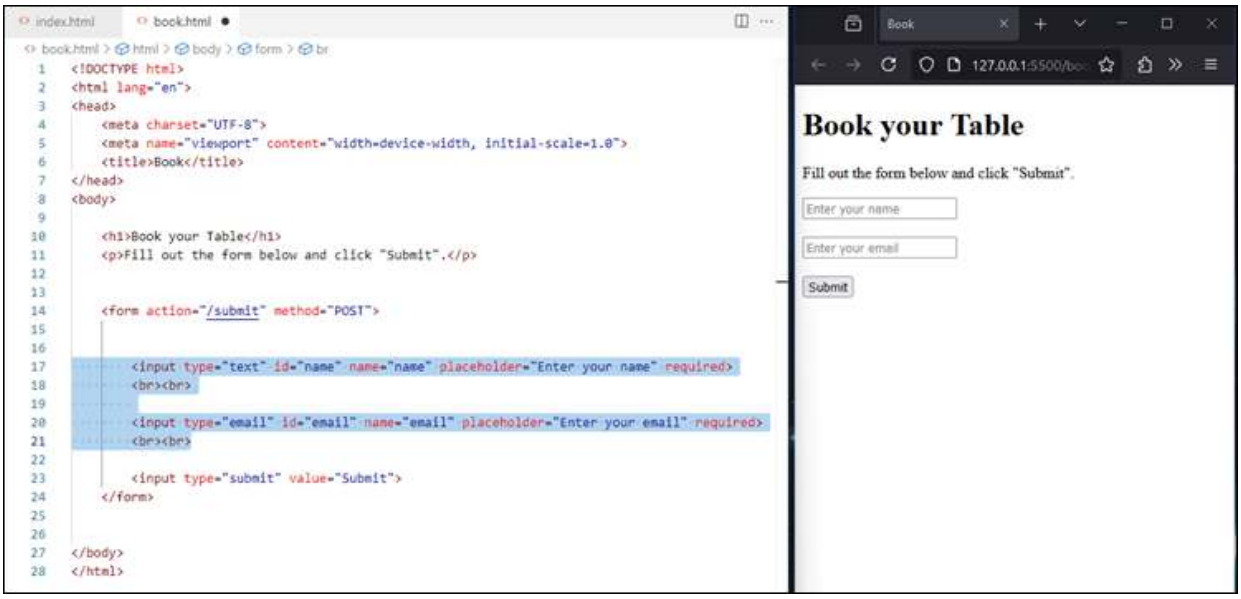


***Figure 6.7**: Adding input types*

In some cases, you do not want the user to leave a field blank. So, to make the field mandatory, we add the required attribute. This is used to indicate that an input field must be filled out before submitting the form. It ensures that the user does not leave the field empty.

```
<input type="email" id="email" name="email" placeholder="Enter email" required>
```

Adding labels

Carrying on from the previous section, we also need to add labels to the fields on our form, so the user knows what information is being requested. To do this, we use the **<label>** element. To add a **label** to the **name** input field, we include the **for** attribute in the opening tag of the **label** element and set this to match the ID attribute of the field we are labelling (in this example, the **name** input field has an ID set to **name**), as follows:

```
<label for="name">Name:</label>
```

In *Figure 6.8*, labels to the **name** and **email** input fields have been added:

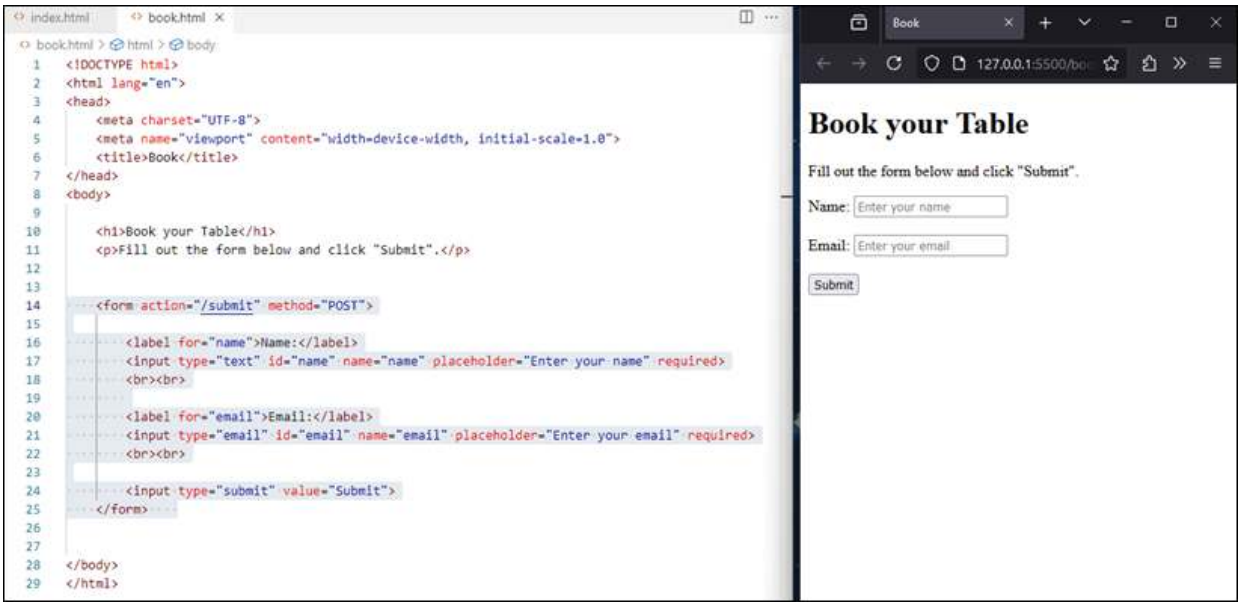


***Figure 6.8**: Adding labels*

Grouping form elements

When creating complex forms with multiple fields, it is essential to organize and group these fields to improve readability and usability. By grouping form elements and organizing them logically, users will understand which fields are connected.

To group fields, the **<fieldset>** and **<legend>** elements are used.

For example, expanding the booking form from the previous section, we can group contact or personal details, such as name, email, and phone number, into one **fieldset**, and we can group the booking details, such as date, time, and number of guests, into another **fieldset**, as follows:

```
<fieldset>
    <legend>Personal Information</legend>

    <label for="name">Name:</label>
    <input type="text" id="name" name="name"
placeholder="Enter name" required>
    <br><br>
    <label for="email">Email:</label>
    <input type="email" id="email" name="email"
placeholder="Enter email" required>
    <br><br>

    <label for="phone">Phone:</label>
    <input type="tel" id="phone" name="phone"
placeholder="Enter phone number">
    <br><br>
</fieldset>
```

Let us add this to the booking form as follows:

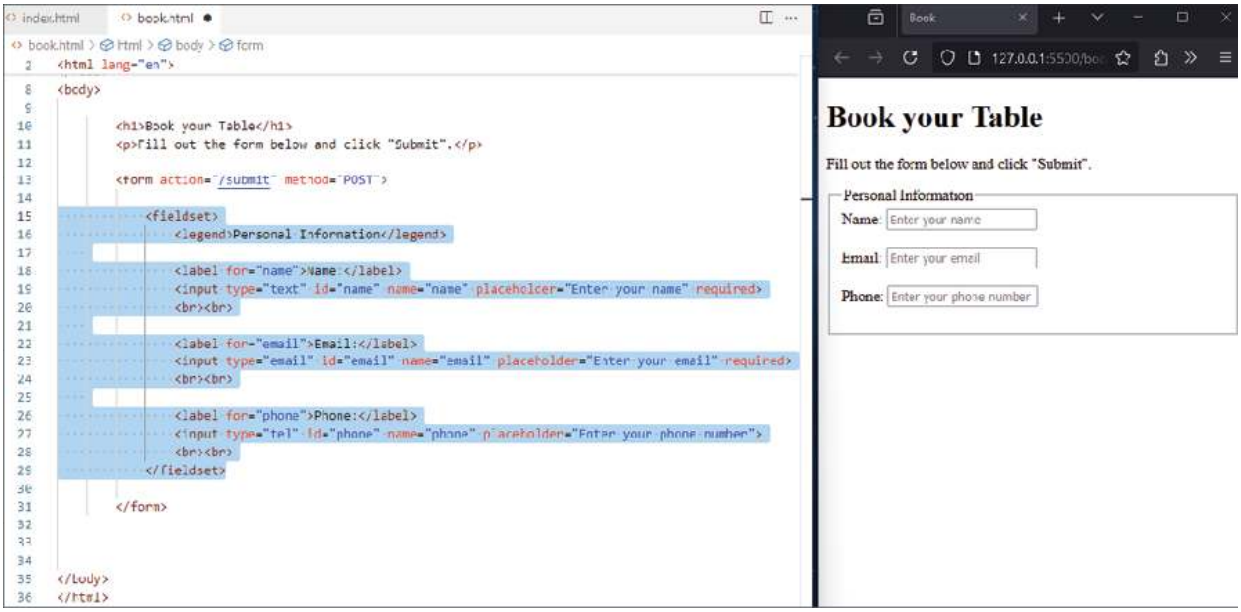


***Figure 6.9**: Grouping elements*

Next, we can add the other **`fieldset`**, as follows:

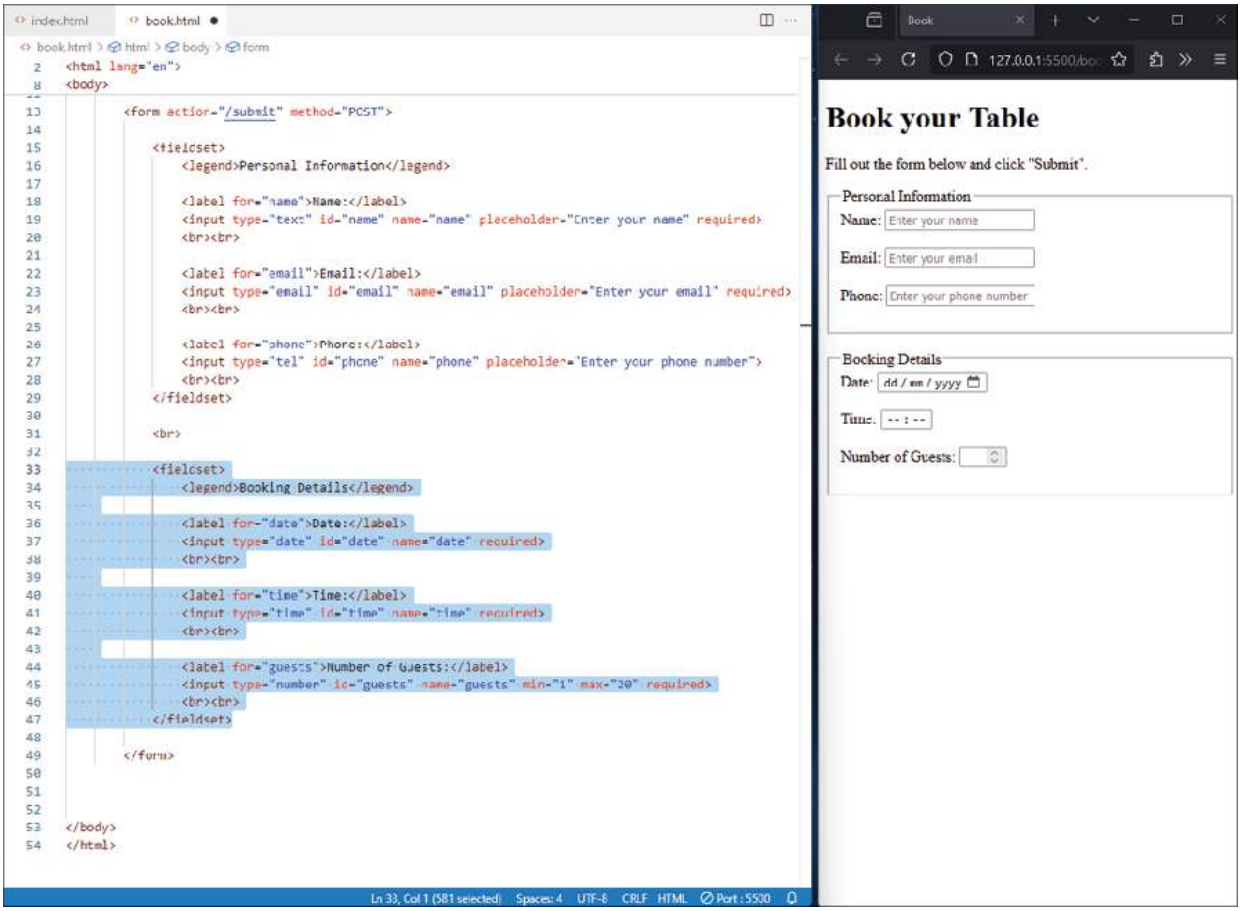


***Figure 6.10**: Adding the other fieldset*

At the bottom of the form, we can add the submission button, as shown in *Figure 6.11*:

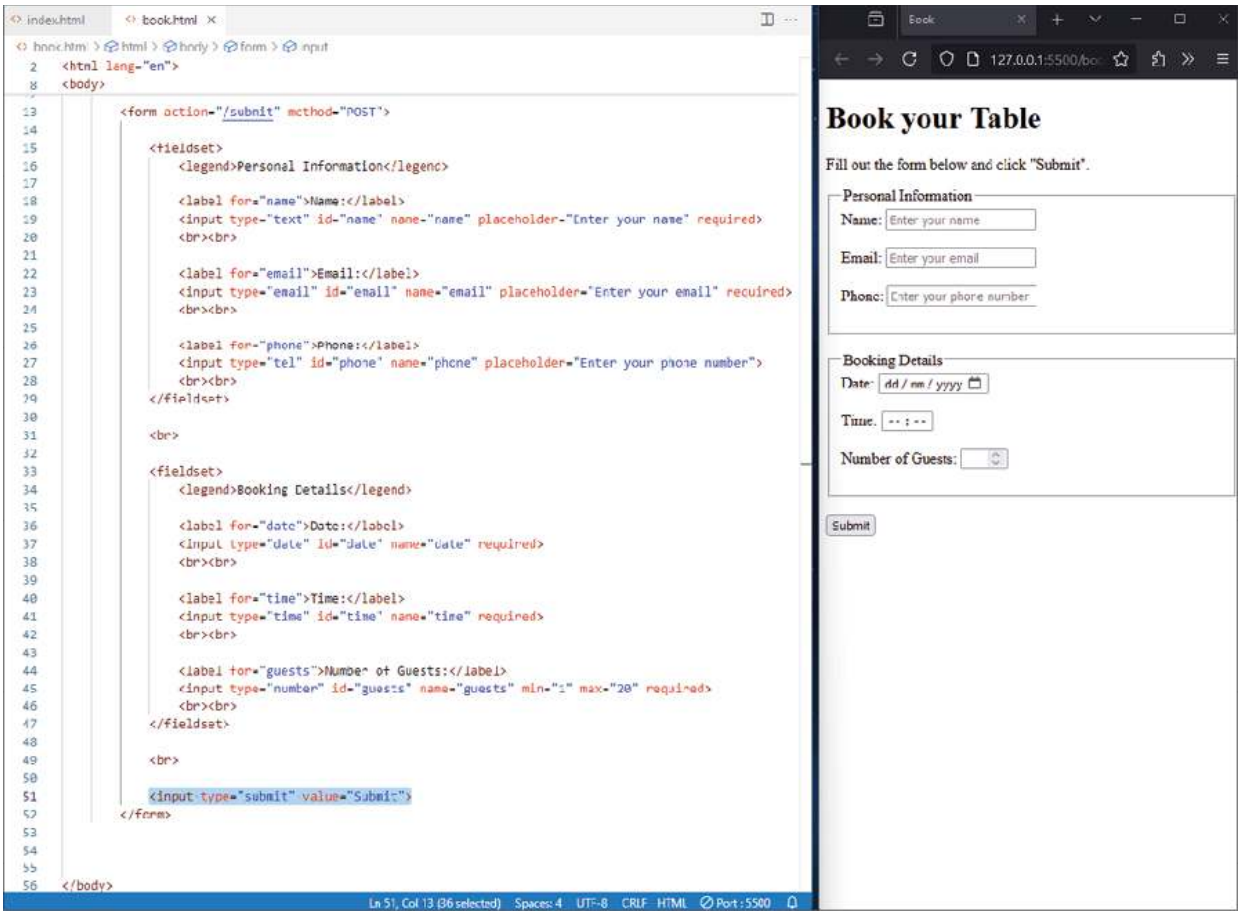


***Figure 6.11**: Adding a submit button*

Form processing

If we click on the submit button in our form, nothing will happen. This is because we have not set the action attribute in the opening tag of the form element.

```
<form action="/submit" method="POST">
```

For example, if we had a script running on a server, we could add it here. In the following demo, a JavaScript Node.js app is running to process the form data:

```
<form action="http://localhost:3000/submit"
method="POST">
```

A server.js script has been included for you to try out yourself. To run it, you will need Node.js installed on your machine. To run the app, make sure you are currently in the directory where the **`server.js`** file is saved. You can

use the **`cd`** command if you need to change the directory.

```
cd /path/to/your/directory
```

Type the following line at the command prompt:

```
node server.js
```

We will take a more in-depth look at JavaScript later in this book, but for now, the JavaScript file is included so you can see how form submission works and try it out yourself.

In *Figure 6.12*, we can see the **`server.js`** app running in the command prompt on the bottom right corner with the received data. Also, in the browser on the right-hand side, we can see the server's response to the form submission:

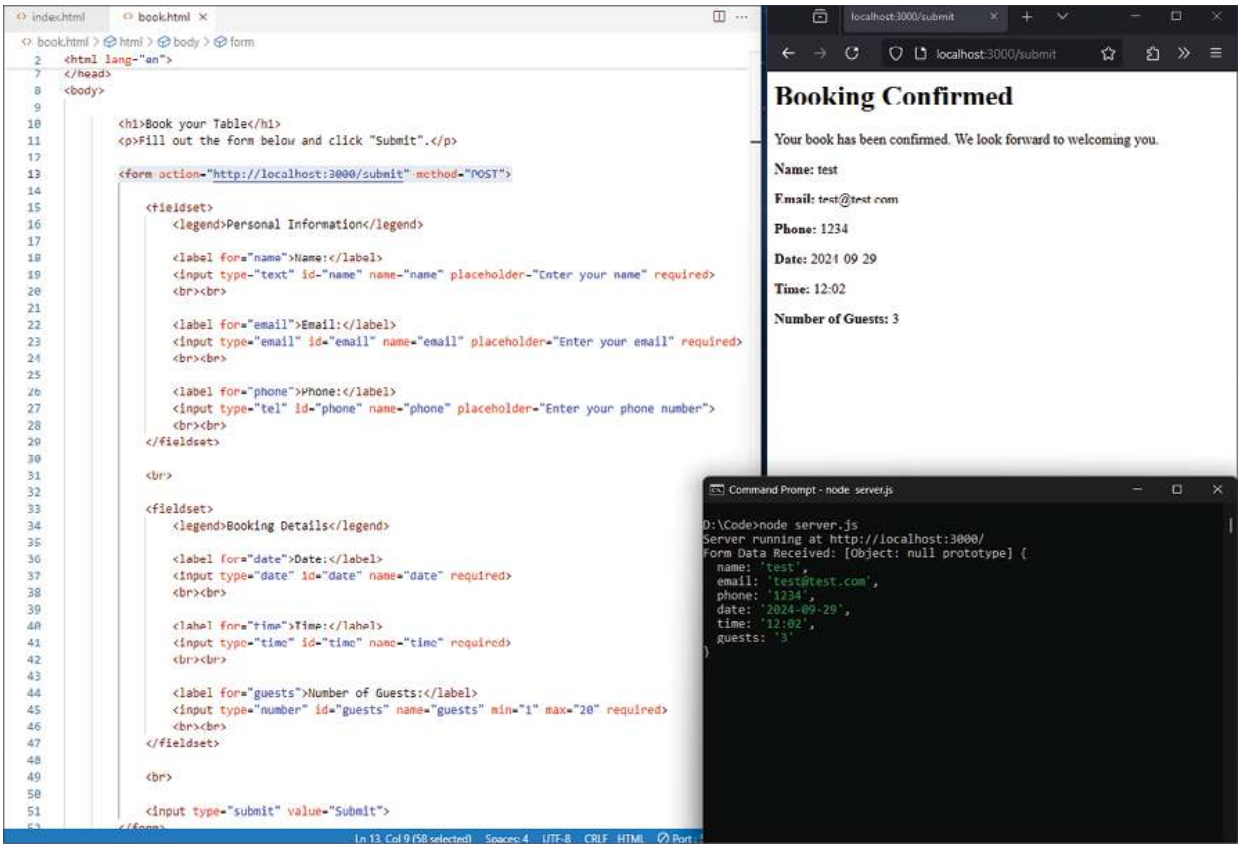


***Figure 6.12**: Processing form data with a server JavaScript app*

This is just a simple demonstration, but the data could be emailed to an event coordinator at the restaurant, or it could be stored in a database along with the user's account.

HTML form validation

HTML form validation ensures that the data submitted by a user in a form meets the required format or conditions before sending it to a server. You can use various validation attributes to ensure that the input data is in the correct format and within specified constraints before submitting a form. These attributes are built into HTML5, making it easy to enforce input rules directly within the HTML code. You can do this by adding the attributes directly to the HTML elements in your form.

The required attribute makes the **input** field mandatory. If a user tries to submit the form without filling in this field, an error message appears, as follows:

```
<input type="text" name="username" required>
```

The **type** attribute enforces specific data types like **email**, **number**, **URL**, **date**, etc. For example:

```
<input type="email" name="email">
```

The **min** and **max** attributes set minimum and maximum values for numeric and date inputs:

```
<input type="number" name="age" min="18" max="99">
```

The **minlength** and **maxlength** attributes restrict the number of characters in a text field:

```
<input type="text" name="password" minlength="8" maxlength="16">
```

The pattern attribute allows you to use a regular expression to specify a pattern that the input must match:

```
<input type="text" name="zipcode" pattern="[0-9]{5}">
```

The step attribute defines the increment step for number inputs (useful for currencies, quantities, etc.):

```
<input type="number" name="quantity" step="1">
```

Conclusion

HTML forms provide user interaction on the web, enabling visitors to

submit data for tasks such as logging in, signing up, submitting feedback, or placing orders. In this chapter, you gained a foundational understanding of how to create and structure forms using the <form> element along with various input types, such as text fields, checkboxes, radio buttons, and drop-down menus.

You also learned how to associate form controls with labels to improve usability and accessibility, group related form fields using <fieldset> and <legend>, and implement form submission through methods like GET and POST. Additionally, you explored how to validate user input using both basic HTML attributes and form design best practices, preparing your forms for integration with server-side processing using technologies such as PHP.

By the end of this chapter, you should be able to construct structured, user-friendly, and functional web forms that collect data efficiently and securely.

In the next chapter, you will move on to CSS, which controls the visual design and layout of your web pages. You will learn how to use selectors, properties, and values to apply styles to HTML elements, manage fonts and colors, add effects and animations, and understand the box model and display properties. The chapter will also introduce you to Sass, a CSS preprocessor that streamlines and enhances stylesheet development.

Points to remember

- Forms can use the GET or POST method for submission.
  - GET appends data to the URL
  - POST sends data in the HTTP body and is ideal for sensitive or large data submissions.
- Common input types include text, password, email, number, tel, URL, checkbox, and radio button.
  - Use the id attribute to identify input elements uniquely.
  - Use the name attribute to specify input names for data identification.
  - The placeholder attribute guides users on expected input.
  - The required attribute ensures that input fields cannot be left empty before submission.

    - <label> elements should use the for attribute to link labels to input fields by their ID.
- <fieldset> and <legend> elements help group related form elements for better organization and readability.
- The action attribute in the <form> tag specifies the server-side script that processes submitted data.
- Server-side technologies, such as Node.js, PHP, or Python, can be used to process form data and perform actions like storing in databases or sending emails.
- To test form submissions locally, set up a local server (e.g., Node.js) and run it to handle form requests and responses.

Exercises

1. What are the two methods used for submitting form data to the server, and how do they differ?
2. Which form submission method would be ideal for transmitting sensitive information, and why?
3. Name at least four common input types used in HTML forms.
4. What is the purpose of the required attribute in a form input field?
5. How do you associate a label with an input field using the <label> element?
6. Why is it important to group form elements using the <fieldset> and <legend> elements?
7. What attribute must be set in the <form> element to specify the server-side script that processes the form data?
8. Describe how form validation can be implemented to ensure correct data submission.
9. What server-side technologies can be used to handle form data processing? Provide at least three examples.
10. What steps are needed to run a Node.js server to process form submissions in the provided example?

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com



CHAPTER 7

Introduction to CSS

Introduction

This chapter provides a comprehensive overview of CSS, covering its fundamental syntax, including selectors, properties, and values. It explores different methods for linking CSS to HTML, external, internal, and inline styles. Readers will learn how to use colours and fonts effectively, understand the box model (margin, padding, border), and grasp display properties (block, inline, inline-block). By the end of this chapter, you will be equipped with the essential skills to style and layout web pages effectively.

Structure

The chapter covers the following topics:

- Understanding CSS
- Selectors, properties, and values
- Linking CSS to HTML
- Working with colors and fonts
- Adding special effects
- Transitions

- Animations
- Using the box model
- Understanding display properties
- Introduction to Sass

Objectives

By the end of this chapter, you will understand the basics of CSS and what selectors, properties, and values are. You will learn how to link external CSS files to your HTML, as well as understand the box model and display properties.

Understanding CSS

CSS is used to define and customize the styles and layouts of your web pages. This means you create style sheets that specify the design, structure, and responsiveness of a web page for different devices, from desktop computers to smartphones. Using CSS, you can control how HTML elements are displayed on screen by specifying properties such as colors, fonts, spacing, and positioning. These are known as **style rules**.

These styles are called **cascading** because they can come from multiple sources, such as external stylesheets, internal stylesheets, and inline styles. CSS uses a hierarchy to determine which styles are applied based on specificity and importance. Specificity refers to how the styles are prioritized depending on the type of selector used (IDs, classes, or tag names. This depends on how precisely a style targets an element (IDs, classes, etc). Styles that have higher specificity, are marked !important, or are defined later in the cascade, take precedence over others. This ensures that styles with greater specificity or importance override less targeted styles, resulting in the correct appearance of elements. The following figure shows how the cascade works:

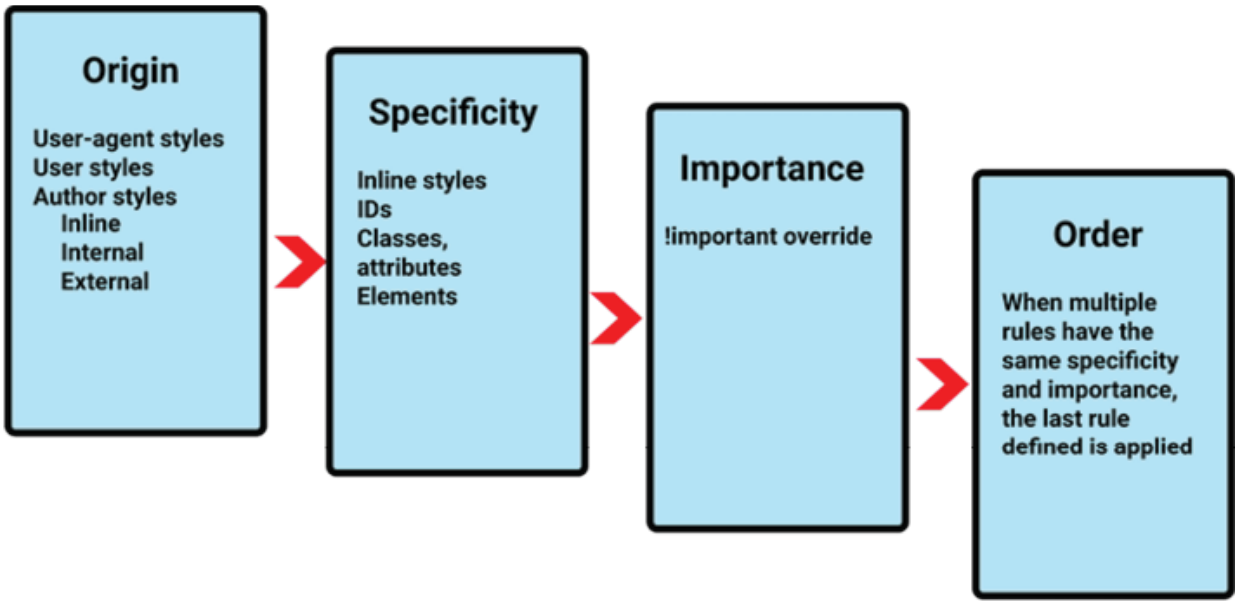


***Figure 7.1**: CSS cascade*

Let us discuss the preceding figure in detail as follows:

- The origin specifies where the styles are stored:
  - User-agent styles are the default styles provided by the browser.
  - The user in their browser settings defines user styles.
  - The developer of the web page defines author styles. These styles can be directly in the style attribute of the HTML element (inline), internally declared in the <head> element, or defined externally in a separate CSS file.
- Specificity follows the following hierarchy:
  - Inline styles are defined directly within an HTML element (e.g., **`<div style="color: red;">`**).
  - ID selectors using the ID attribute (e.g., **`#header`**).
  - Classes, attributes selectors (e.g., **`.button`**, **`[type="text"]`**).
  - Element types (e.g., **`div`**, **`p`**).
- The important rule overrides all other rules, regardless of origin or specificity. If there are conflicting important rules, the one with higher specificity is applied.
- When multiple rules have the same specificity and importance, the last

rule defined in the stylesheet (or imported stylesheet) is applied. This is known as the **last rule applied** principle.

Selectors, properties, and values

CSS style rules are composed using selectors, properties, and values. A selector specifies the HTML element(s) to be styled. The property defines the specific aspect of the element you want to change, such as color, font size, or margin, and the value specifies the exact style setting for the given property, as we can see in the following figure:

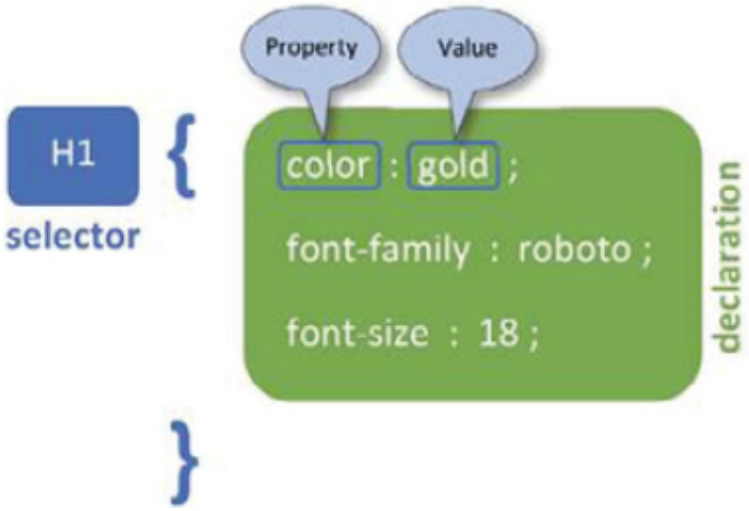


***Figure 7.2**: CSS selector*

Selector types

There are various types of selectors that allow you to target the specific HTML elements you want to style. Let us take a look at some of the most common ones.

Universal selector

The universal selector, represented by an asterisk (*****), is used to target every single element on a web page. This is useful for applying a general style to all elements at once. For example, setting a base style for page margins or paddings, as:

```
* {
```

```
    margin: 0;
    padding: 0;
}
```

Type selector

The type selector (also known as an **element selector**) is used to select all instances of a particular HTML element type. You can think of it as applying styles based on the HTML tag name itself. These selectors are often used to set general styles for HTML elements, such as headings, paragraphs, or lists, as:

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

This will style all `<h1>` elements with a font size of 24px.

Class selector

The class selector is one of the most common selectors and is used to style elements that have a specific class attribute. The class selector enables you to style multiple elements consistently without affecting other elements of the same type, as:

```
.highlight {
    background-color: yellow; /* Highlights text with a yellow background */
}
<p class="highlight">This text is highlighted</p>
```

This will give the styled text a yellow background.

Linking CSS to HTML

The style definitions are normally saved in external .css files. With an external stylesheet file, you can change the look of an entire website by changing just one file.

Inline styles

Inline styles are added directly to the HTML element using the style attribute. This method allows you to apply CSS directly to a single element without affecting any other elements on the page. You can do this using the style attribute in the opening HTML tag, as:

```
<p style="color: red; font-size: 20px;">This text is styled with inline CSS</p>
```

These are used to apply a unique style to a single element.

Internal styles

Internal styles are defined within the **<style>** element inside the **<head>** section of an HTML document. This method allows you to keep the styles separate from the HTML elements themselves, but still within the same file, as follows:

```
<!DOCTYPE html>
<html>
<head>
    <style>
        p {
            color: blue;
            font-size: 18px;
        }
    </style>
</head>
<body>
    <p>This paragraph is styled using internal CSS</p>
</body>
</html>
```

This is used if a single HTML page has a unique style.

External styles

External styles are defined in a separate .css file, which is then linked to the HTML document using the **<link>** element, as shown in *Figure 7.3*:

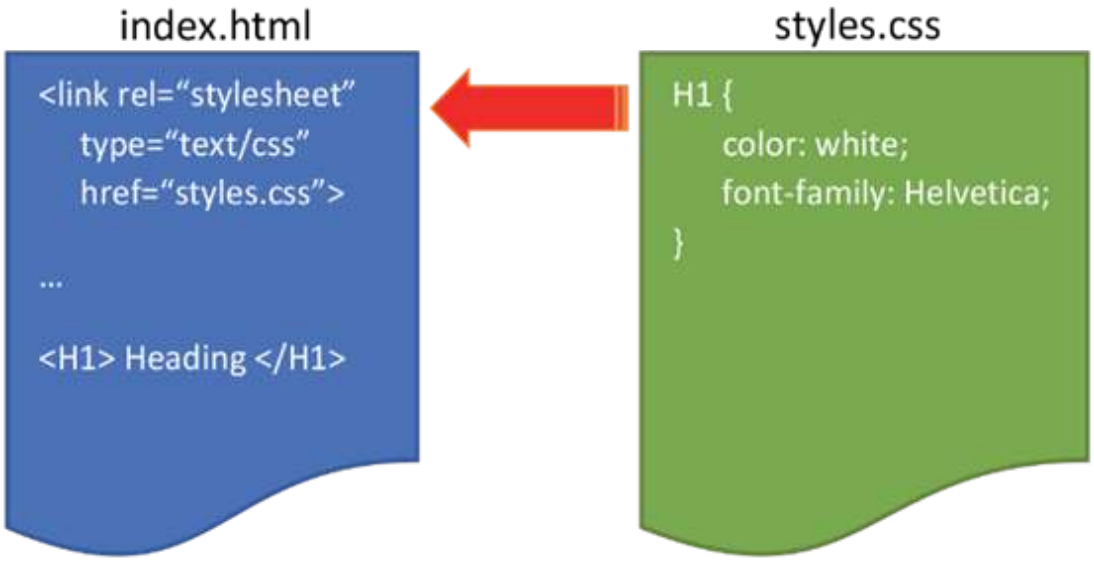


***Figure 7.3**: Linking external CSS*

In the following HTML code, the **`<link>`** element in the **`<head>`** section links to an external CSS file named **`styles.css`**, where all the styles have been defined:

```
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <p>This paragraph is styled using an external CSS file</p>
</body>
</html>
```

This approach is best for multi-page websites and scalable projects, as it allows code reuse and makes maintenance a lot easier.

Working with colors and fonts

Working with colors and fonts involves setting various properties and values to control the appearance of the elements on a web page.

Named colors

There are various ways to add colors, and one of the easiest ways to do this is to use the predefined colors. The following are a few common colors:

- black
- white
- gray
- lightgray
- darkgray
- silver
- darkblue
- orange
- gold

If you are using VS Code, you can also use the color selector to set the color. This color selector appears when you type a color attribute and press the colon, as shown in *Figure 7.4*:



***Figure 7.4**: VS Code color selector*

To change the colors of the various elements, you can add different attributes, such as the following, to specify a color:

- **`color`**, which sets the text color of an element.
- **`background-color`**, which sets the background color of an element.

- **`border-color`**, which sets the border color of an element.

Let us take a look at an example. As illustrated in *Figure 7.5*, a CSS file named **`fontandcolor.css`** has been created, in which a set of style rules has been defined to modify the color of heading elements. In this instance, the **`h1`** type selector has been used to apply the defined styles uniformly to all **`<h1>`** elements across the web page.

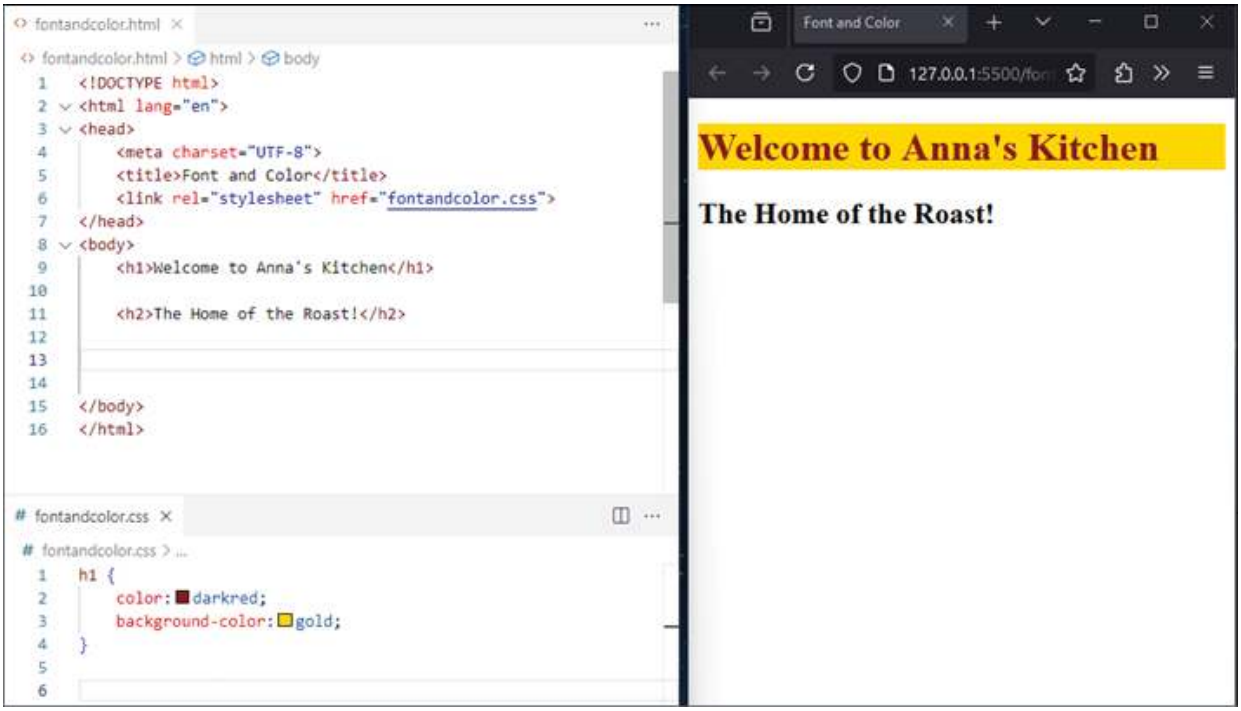


***Figure 7.5**: Changing font and background colors*

To style specific elements of the page, use a class selector. Suppose we want to style the first paragraph differently from the second. Here in *Figure 7.6*, two styles have been created, one for the description, which will be paragraph one. A highlight style, which is applied to the second paragraph, is also created:

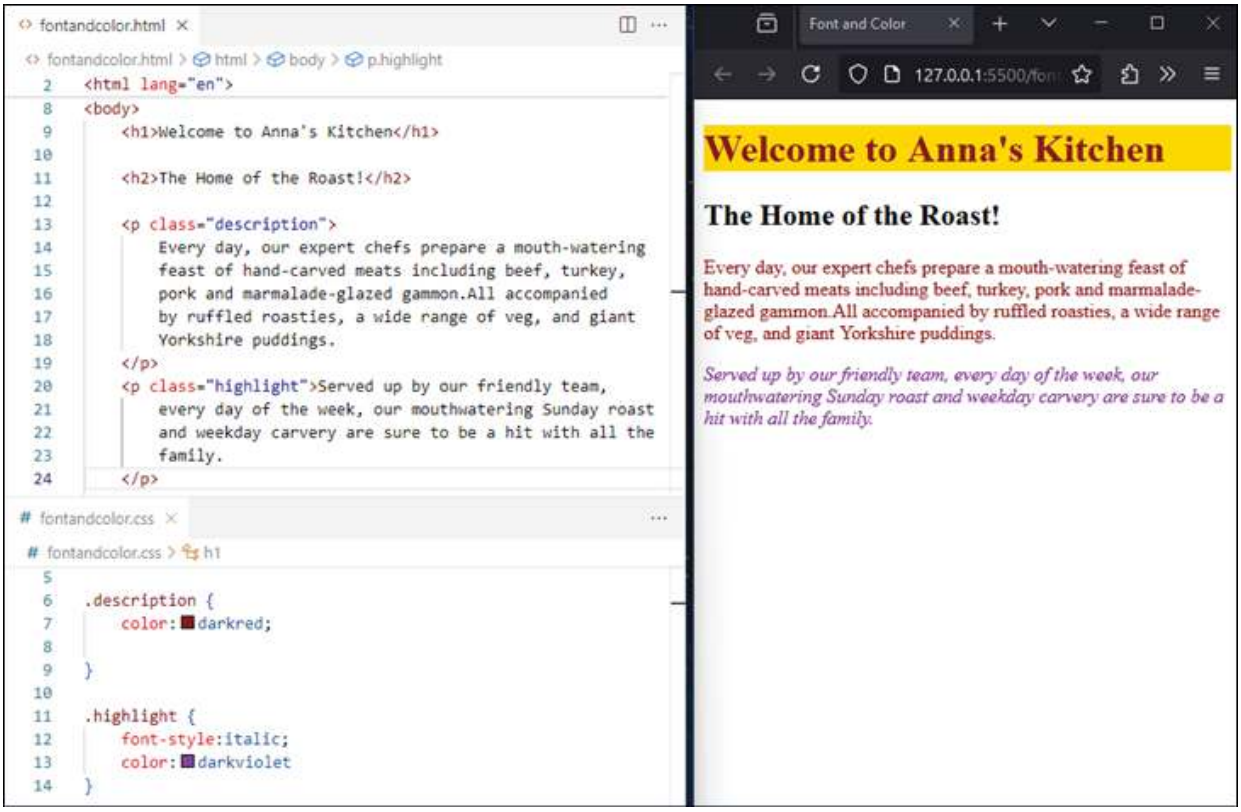


***Figure 7.6**: Changing font styles and colors*

To apply the **`class`** selector in the HTML, we add the **`class`** attribute to the opening tag of the element we are applying the style to, as:

```
    <p class="highlight"> Served up by our friendly team,
        every day of the week, our mouthwatering Sunday roast
        and weekday carvery are sure to be a hit with all the
        family.
    </p>
```

Hexadecimal colors

Hexadecimal (**Hex**) color codes are widely used because they are compact and supported by all browsers. A hex color is represented by a hash **#** symbol followed by either three or six hexadecimal digits, as:

- **Three-digit format**: **#RGB**

- **Six-digit format**: **`#RRGGBB`**

Hex numbers use a number system called base-16. Unlike the decimal system we use (which is base-10), hex uses sixteen symbols to represent numbers (the numbers 0-9 and the letters A-F), as shown in the following table:

Dec	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

***Table 7.1**: Hexadecimal numbers*

In a six-digit hex representation of a color, each pair of digits represents the intensity of the red, green, or blue channel, ranging from **`00`** (no color) to **`FF`** (full color intensity).

For example, **`#FFFFFF`** is white, where all three colors (**`FF FF FF`**) are set to full intensity, and black is **`#000000`**, where all colors (**`00 00 00`**) are at 0 intensity. Similarly, **`#FFA500`** is orange (FF **`A5 00`**), where red (**`FF`**) is set to full intensity, green is set to about medium intensity (**`A5`**), and blue (**`00`**) is set to 0 intensity.

You can shorten hex codes to use three digits instead of six. This can help increase performance in larger websites.

To do this, follow these steps:

1. Check each pair of digits in the 6-digit hex code.
2. If the pair is identical, shorten it to a single digit. If the pair is not identical, approximate the pair to the closest single digit.
3. Combine the shortened digits to form the 3-digit hex code.

   For example, **`#FFFFFF`** (white) can be shortened to **`#FFF`**.

If the pairs are not identical, such as **`FFA500`**, i.e., as follows:

- Red (**`FF`**) contains identical digits, so it can be shortened to **`F`**.
- Green (**`A5`**) is not identical, so it can be approximated to **`A`**.
- Blue (**`00`**) contains identical digits, so it can be shortened to **`0`**.

The following table shows some common colors and their corresponding hex codes:

Color	6-digit hex code	3-digit hex code
Black	#000000	#000

White	#FFFFFF	#FFF
Gray	#808080	N/A
Light gray	#D3D3D3	N/A
Dark gray	#A9A9A9	N/A
Silver	#C0C0C0	N/A
Dark blue	#00008B	N/A
Orange	#FFA500	#FA0
Gold	#FFD700	#FD7
Red	#FF0000	#F00
Green	#00FF00	#0F0
Blue	#0000FF	#00F

***Table 7.2**: Color codes*

Let us take a look at an example. Here in *Figure 7.7*, the background color of the heading **<h1>** has been changed to orange and the text color to red using hex codes. You will see that the first hex code uses the 6-digit format, and the second code uses the shorthand 3-digit format:

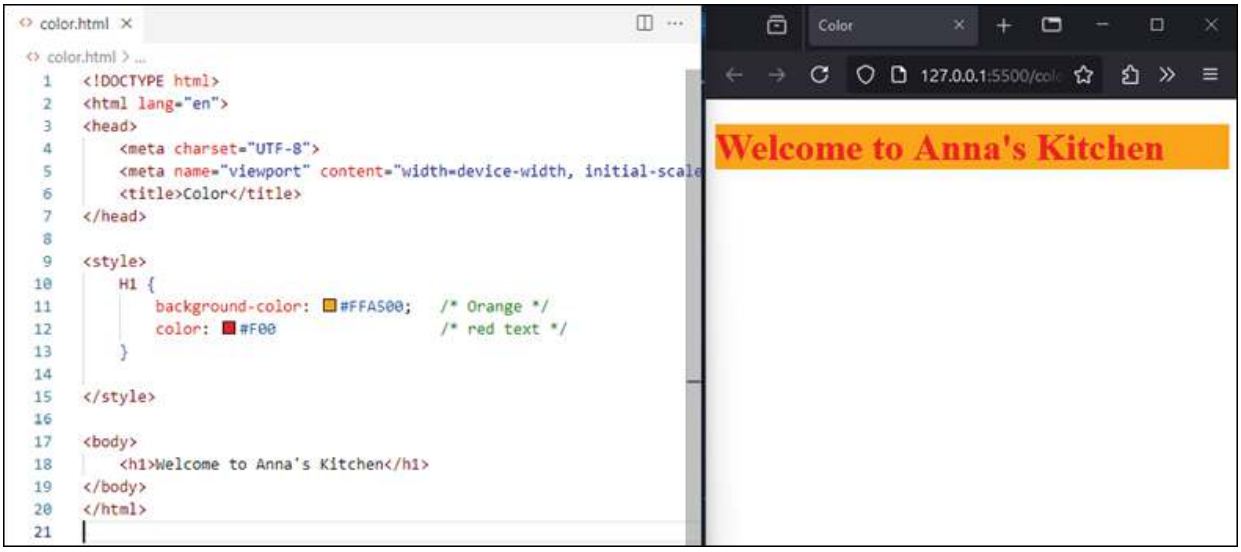


***Figure 7.7**: Adding colors using hex codes*

Fonts

CSS provides various properties to style the fonts used on web pages. These include setting the typeface, size, weight, and style.

There are various sources of fonts. Google Fonts is a free, web-based collection of open-source fonts that are hosted on Google's servers and can be easily integrated into web projects by linking to them in your HTML or CSS, as follows:

https://fonts.google.com/

Let us take a look at an example using the following steps. For the following web page, the **Roboto** font has been used.

1. Search for the font you want to use, such as **Roboto**, as shown in *Figure 7.8*:

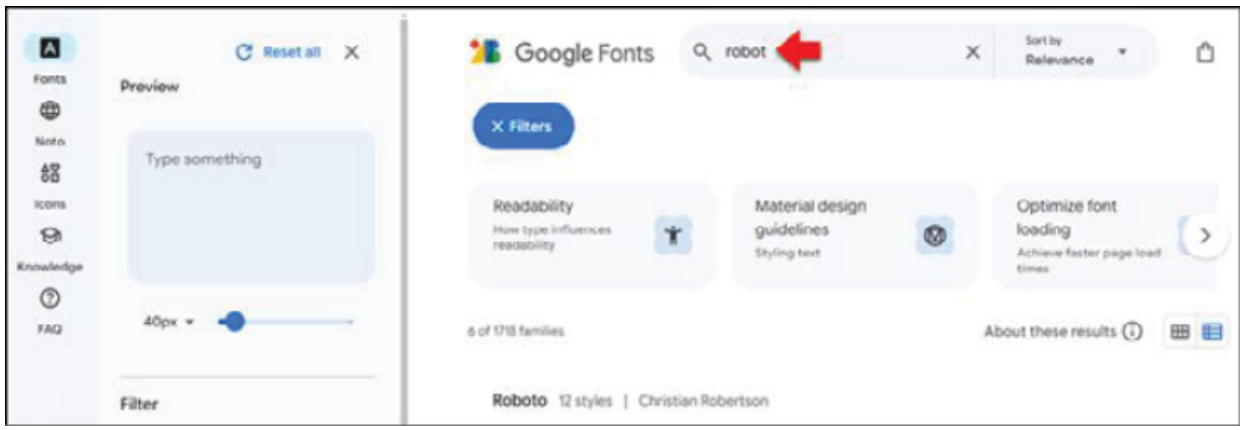


***Figure 7.8**: Searching Google Fonts*

2. Click on the font name to go to its details page, as shown in *Figure 7.9*:

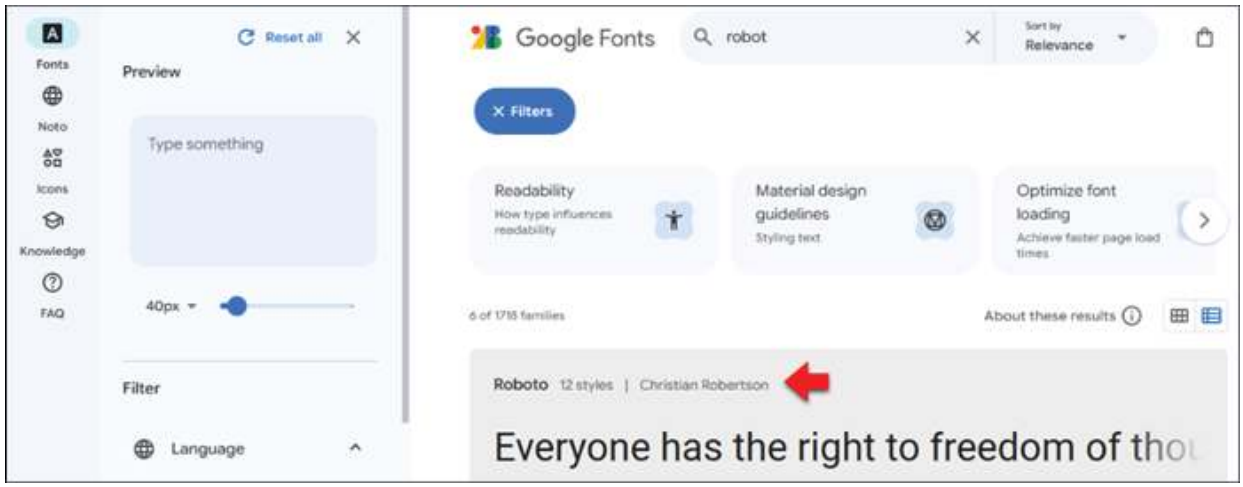


***Figure 7.9**: Selecting a font on Google Fonts*

3. Click **Get font** in the top right corner, as shown in *Figure 7.10*:



***Figure 7.10**: Getting font details*

4. Click **Get embed code**, as shown in *Figure 7.11*:



***Figure 7.11**: Getting the embed code from Google Fonts*

5. On the left side, click **Change styles**, as shown in *Figure 7.12*:

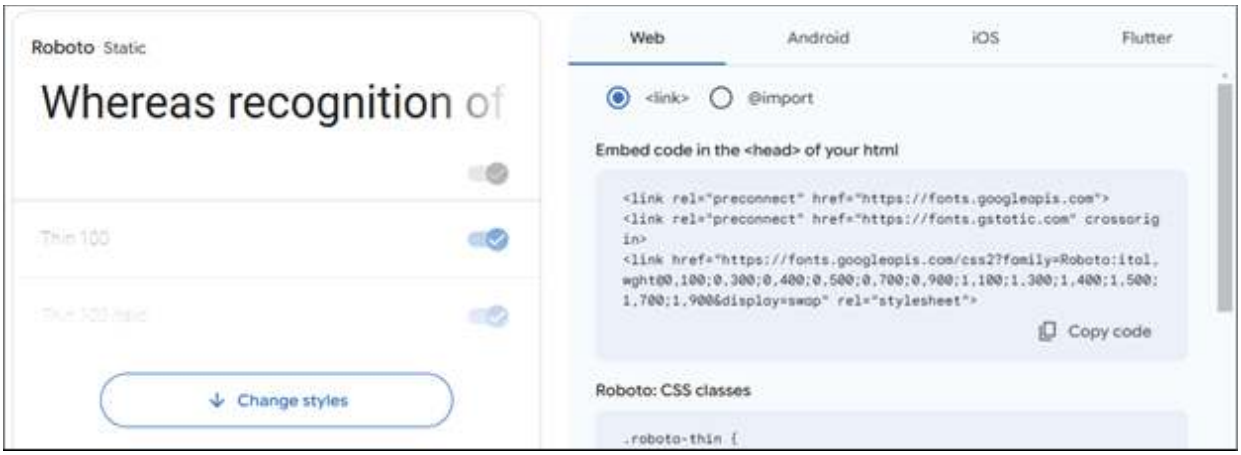


***Figure 7.12**: Copying the embed code from Google Fonts*

6. Select the styles you do not need by clicking the switches next to each weight to turn them off. Only leave the ones you want to use, such as **Regular 400** and **Bold 700**, as shown in *Figure 7.13*:

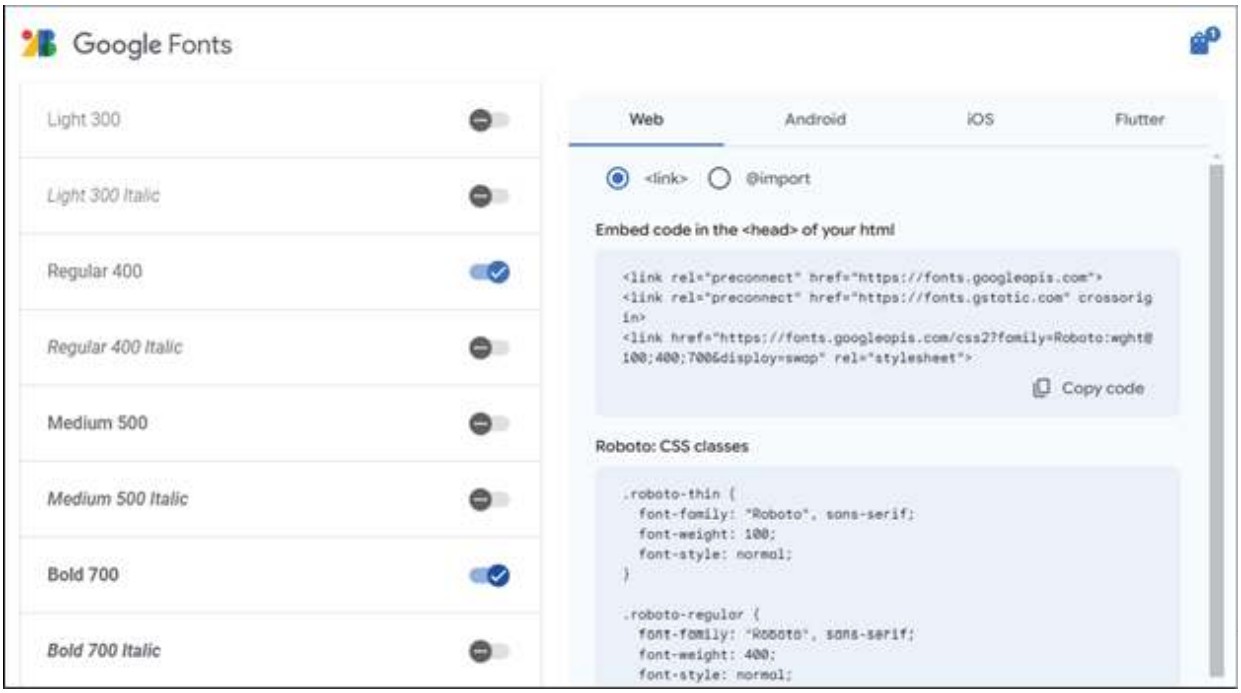


***Figure 7.13**: Selecting fonts to download*

7. On the right panel, as shown in *Figure 7.14*, you will see a section called Embed code in **`<head>`** of your HTML. Click on **Copy code**. This will copy the link to the clipboard.
8. Paste the following code into the head section of your HTML file:

```
<link rel="preconnect"
href="https://fonts.googleapis.com">
<link rel="preconnect"
href="https://fonts.gstatic.com" crossorigin>
<link href="https://fonts.googleapis.com/css2?
family=Roboto:wght@100;400;700&
display=swap" rel="stylesheet">
```

9. Let us add this to our web page. As shown in *Figure 7.14*, the code has been pasted into the **head** section of the HTML page. The heading **class** selector has also been styled to use the **Roboto** font, and a description class selector to style the paragraph of text:

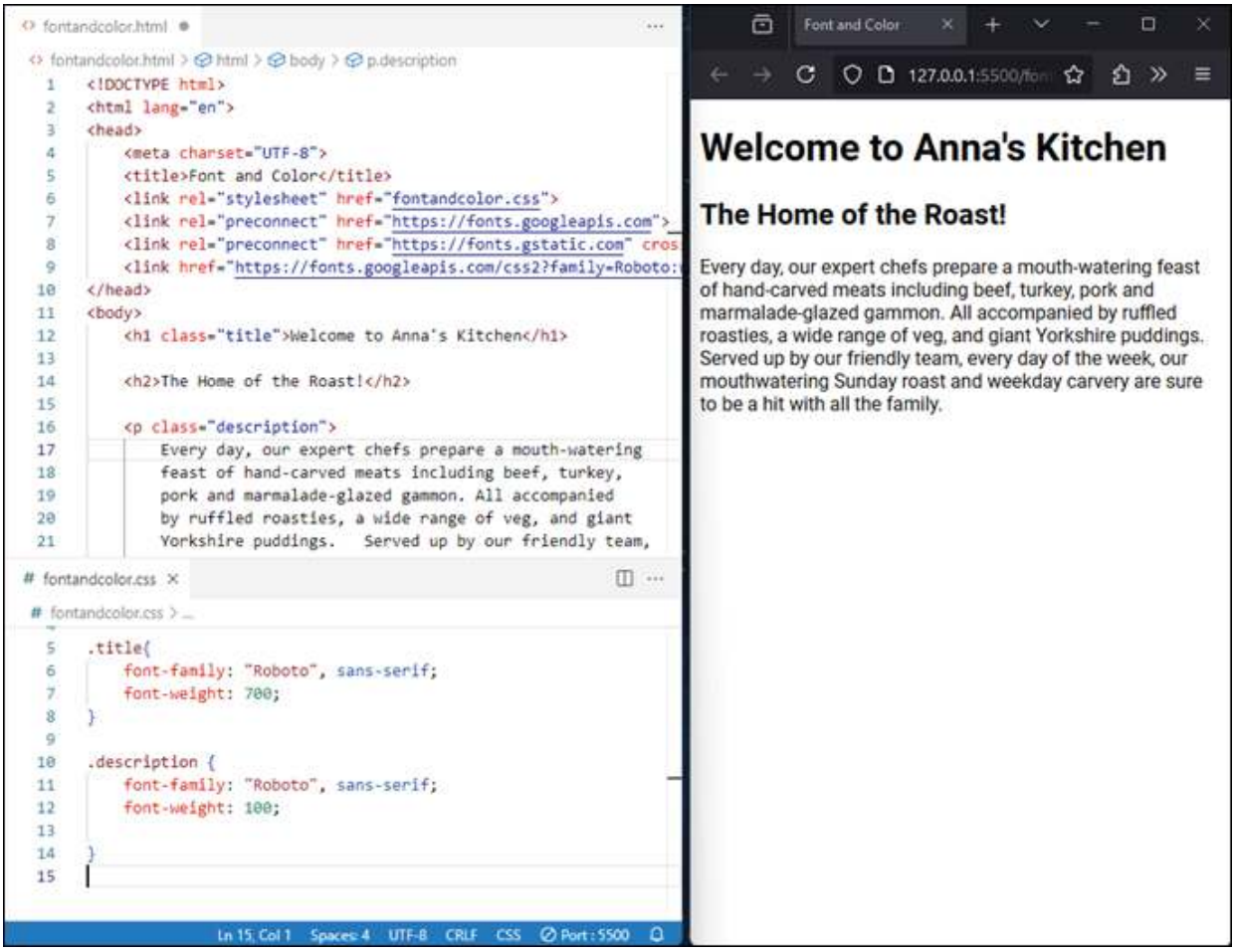


***Figure 7.14**: Adding Google Fonts to an HTML page*

Adding special effects

CSS provides a variety of techniques to create special effects and style elements on your web pages. You can use selectors and classes to target specific elements and then apply styles accordingly. If you want to apply the effects to specific elements, use a class, or if you want to target all elements of a specific type, use an element selector.

Text effects

CSS text effects can make your text stand out and enhance readability. For example, we can add shadows to text, creating a three-dimensional effect, as:

```
h1 {
    text-shadow: 2px 2px 5px lightgrey;
}
```

Or, you can add a color gradient to the text, as:

```
h1 {
    background: linear-gradient(90deg, red,
gold);
    -webkit-background-clip: text;
     color: transparent;
}
```

Pseudo-elements

Pseudo-elements are used to style specified parts of an element. The following table shows a list of the most commonly used pseudo-elements along with simple examples:

Pseudo-element	Description	Example
::before	Inserts content before an element's content.	`css p::before { content: "Start: "; }`
::after	Inserts content after an element's content.	`css p::after { content: " - End"; }`
::first-letter	Styles the first letter of an element.	`css p::first-letter { font-size: 2em; color: red; }`

Pseudo-element	Description	Example
::first-line	Styles the first line of an element.	`css p::first-line { font-weight: bold; }`
::selection	Styles the text selected by the user.	`css ::selection { background: yellow; color: black; }`
::placeholder	Styles the placeholder text in input fields.	`css input::placeholder { color: gray; font-style: italic; }`
::marker	Styles the marker (e.g., bullets) in lists.	`css li::marker { color: blue; font-size: 1.5em; }`

***Table 7.3**: Pseudo-elements*

Let us take a look at a simple example. Have a look at pseudoelement.html and **`pseudoelement.css`**. Here, pseudo-elements have been added to the paragraph tag to change the size, color, and positioning of the first letter, and the color and font weight of the first line. You will also notice that the bullet points (**`li::marker`**) are set to red, and the highlighted text (**`::selection`**) has been changed to yellow, as shown in *Figure 7.15*:

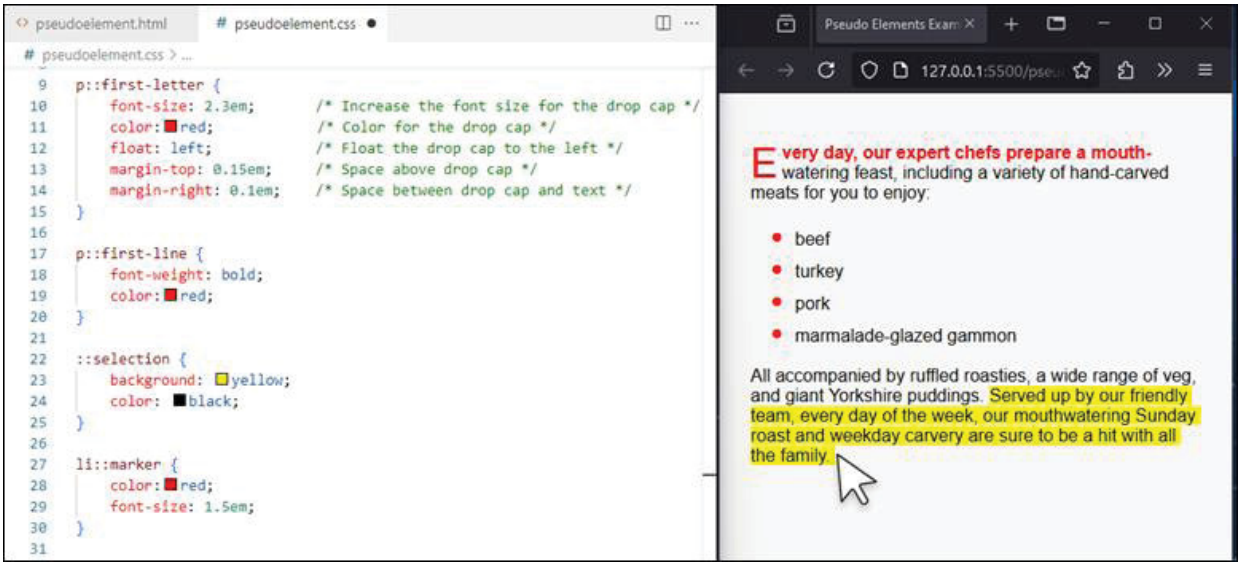


***Figure 7.15**: Using pseudo-elements*

Shadows

Shadows are used to create depth and add a layered effect to elements, making them look more visually appealing and realistic. To do this, we use

the **box-shadow** attribute, as:

```
box-shadow: 4px 4px 10px darkgrey;
```

Here, the first value is the horizontal offset. This moves the shadow 4 pixels to the right of the element. Positive values shift the shadow to the right, while negative values shift it to the left.

The next value is the vertical offset. This moves the shadow 4 pixels down from the element. Positive values move the shadow down, while negative values move it up.

The next value is the blur radius. This defines the blur effect of the shadow. A larger blur radius makes the shadow appear softer and more diffused. If the blur radius is set to 0px, the shadow will be a solid, sharp-edged shape.

The last value is the shadow color. This sets the shadow color to a predefined CSS color, darkgrey. You can also use other color formats, such as **rgba**, **hex**, or other named colors. Refer to the following figure:

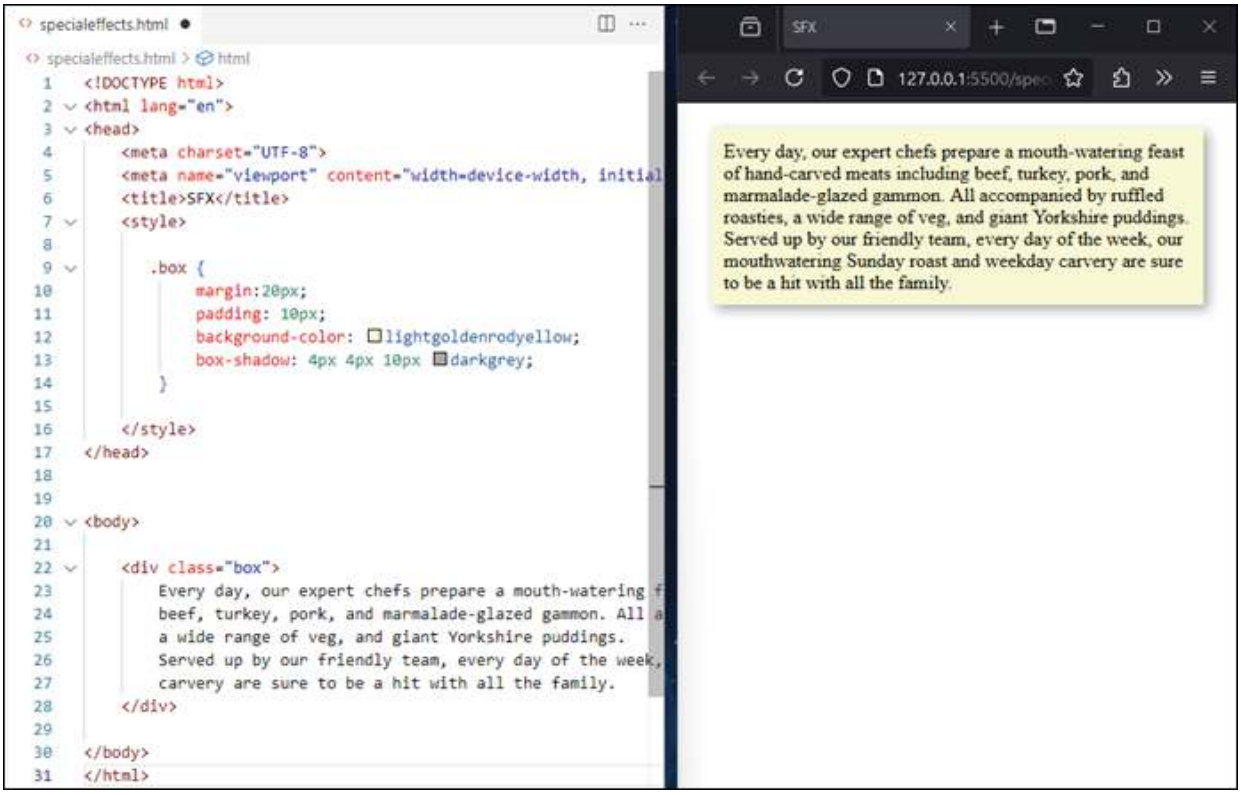


***Figure 7.16**: Adding shadows*

Rounded corners

Rounded corners make elements appear softer and more modern. You can do this using the **`border-radius`** property, as:

```
border-radius: 15px;
```

In *Figure 7.17*, we have added the **`border-radius`** property to the **`box`** class in the CSS.

You can also target specific corners using the following properties:

- **`border-top-left-radius`**
- **`border-top-right-radius`**
- **`border-bottom-left-radius`**
- **`border-bottom-right-radius`**

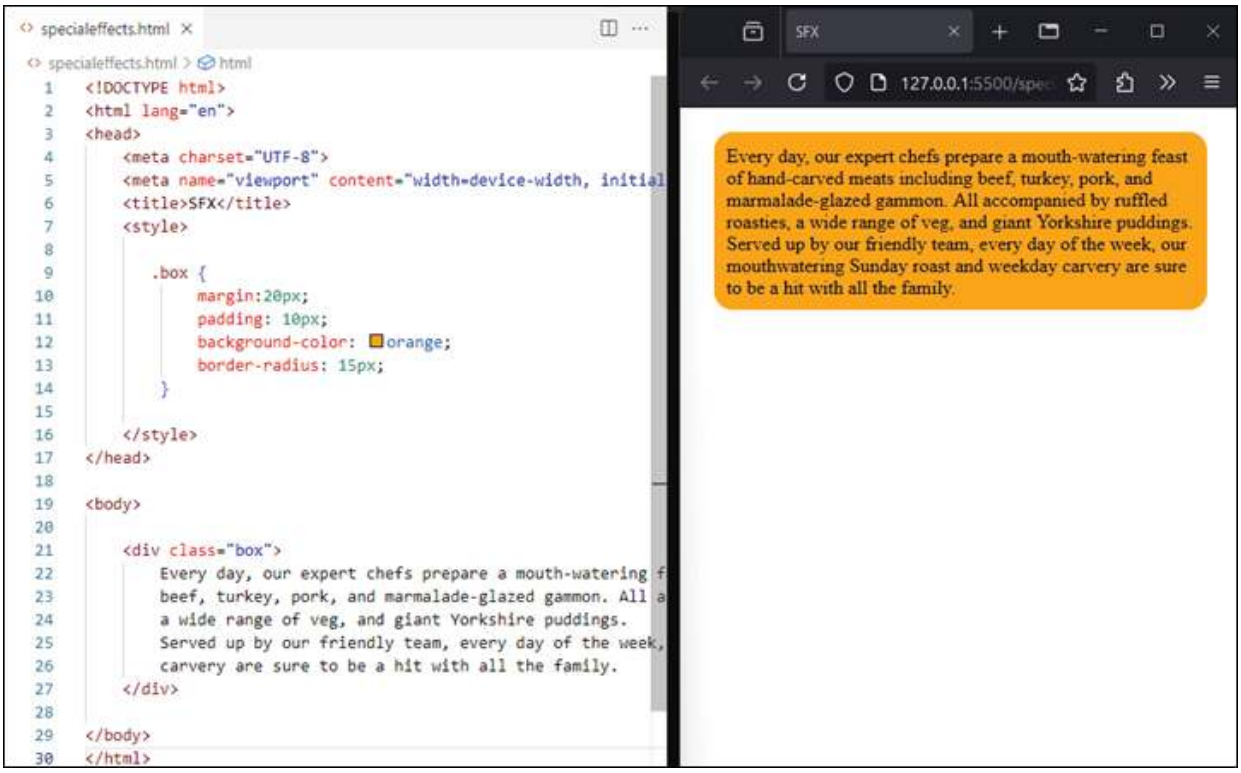


***Figure 7.17**: Rounding corners*

Gradients

Gradients are smooth transitions between two or more colors, used as backgrounds or to create effects like shadows or text fill. You do this using the **`linear-gradient`** function in the background property, as:

```
background: linear-gradient(to right,orange, red);
```

The function **`linear-gradient`** specifies that the gradient will be linear (in a straight line).

The first argument of the function **`to right`** specifies the direction of the gradient. In this case, the gradient will go from left to right.

The next two arguments, orange and red, are the colors used in the gradient. The gradient starts with orange on the left and transitions smoothly to red on the right.

In *Figure 7.18*, we have applied this to our box class:

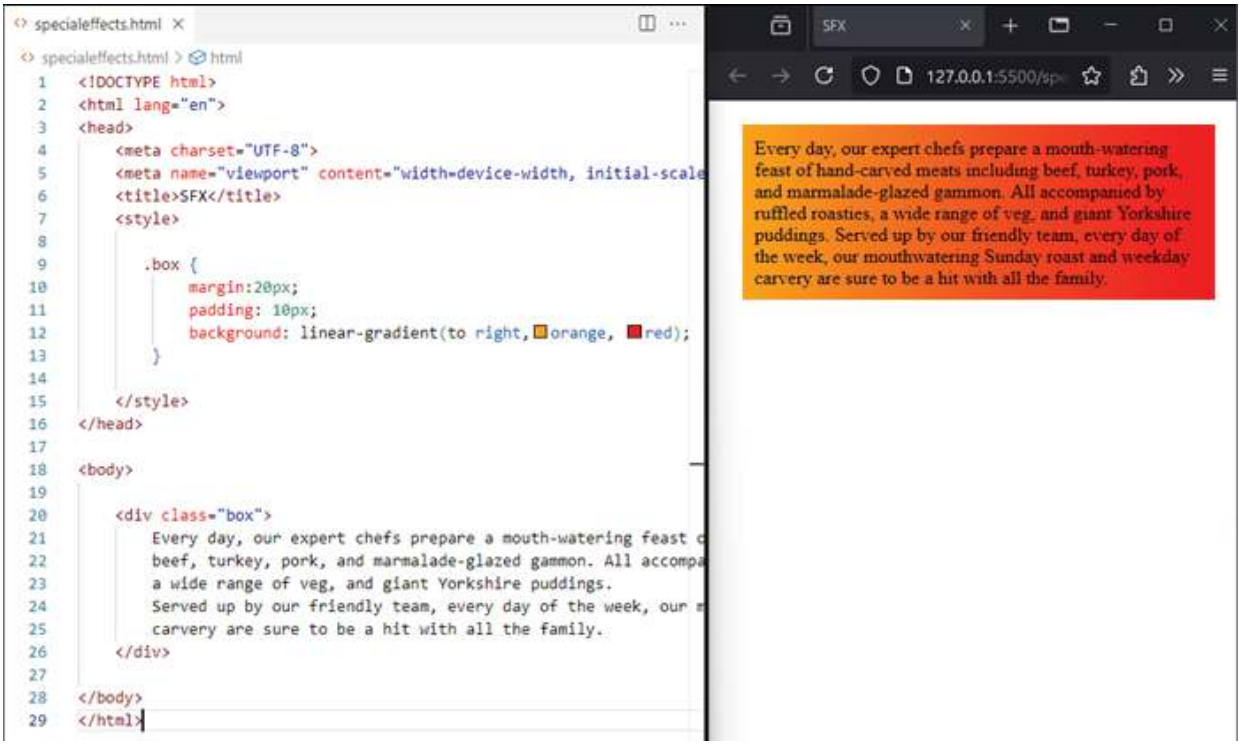


***Figure 7.18**: Adding a linear gradient to the box*

You can also use other gradient functions, like:

- **`linear-gradient()`**
- **`radial-gradient()`**
- **`conic-gradient()`**

Pseudo-classes

Pseudo-classes are used to target elements in a particular state, such as when a user interacts with them. *Table 7.4* shows the most commonly used CSS

pseudo-classes. These can be used to apply styles based on the state or characteristics of an element.

Pseudo-class	Description	Example
:hover	Applies when the user hovers over an element.	`a:hover { color: red; }`
:active	Applies when an element is being activated (e.g., clicked).	`button:active { background-color: blue; }`
:focus	Applies when an element has focus (e.g., input fields).	`input:focus { border-color: green; }`
:first-child	Selects the first child of its parent.	`p:first-child { font-weight: bold; }`
:last-child	Selects the last child of its parent.	`p:last-child { color: gray; }`
:nth-child(n)	Selects the nth child (n can be a number, even, or odd).	`li:nth-child(2n) { color: purple; }`
:nth-of-type(n)	Selects the nth element of a specific type among its siblings.	`p:nth-of-type(1) { font-size: 18px; }`
:not(selector)	Selects elements that do not match a given selector.	`p:not(.intro) { color: black; }`
:checked	Applies to elements that are checked (e.g., checkboxes, radio buttons).	`input:checked { background-color: yellow; }`
:disabled	Applies to elements that are disabled.	`button:disabled { opacity: 0.5; }`
:enabled	Applies to elements that are enabled.	`input:enabled { border-color: green; }`

***Table 7.4**: Pseudo-classes*

For example, if we wanted to change the color of the textbox when the user hovers the mouse pointer over it. We can add the hover pseudo-class to our box class, as:

```
.box {
    margin:20px;
    padding: 10px;
    background-color: lightgoldenrodyellow;
    box-shadow: 4px 4px 10px darkgrey;
```

```
}
.box:hover {
    background-color: lightcoral;
}
```

Here, the textbox will change to a light reddish coral color when the mouse pointer hovers over it, as shown in the following figure:

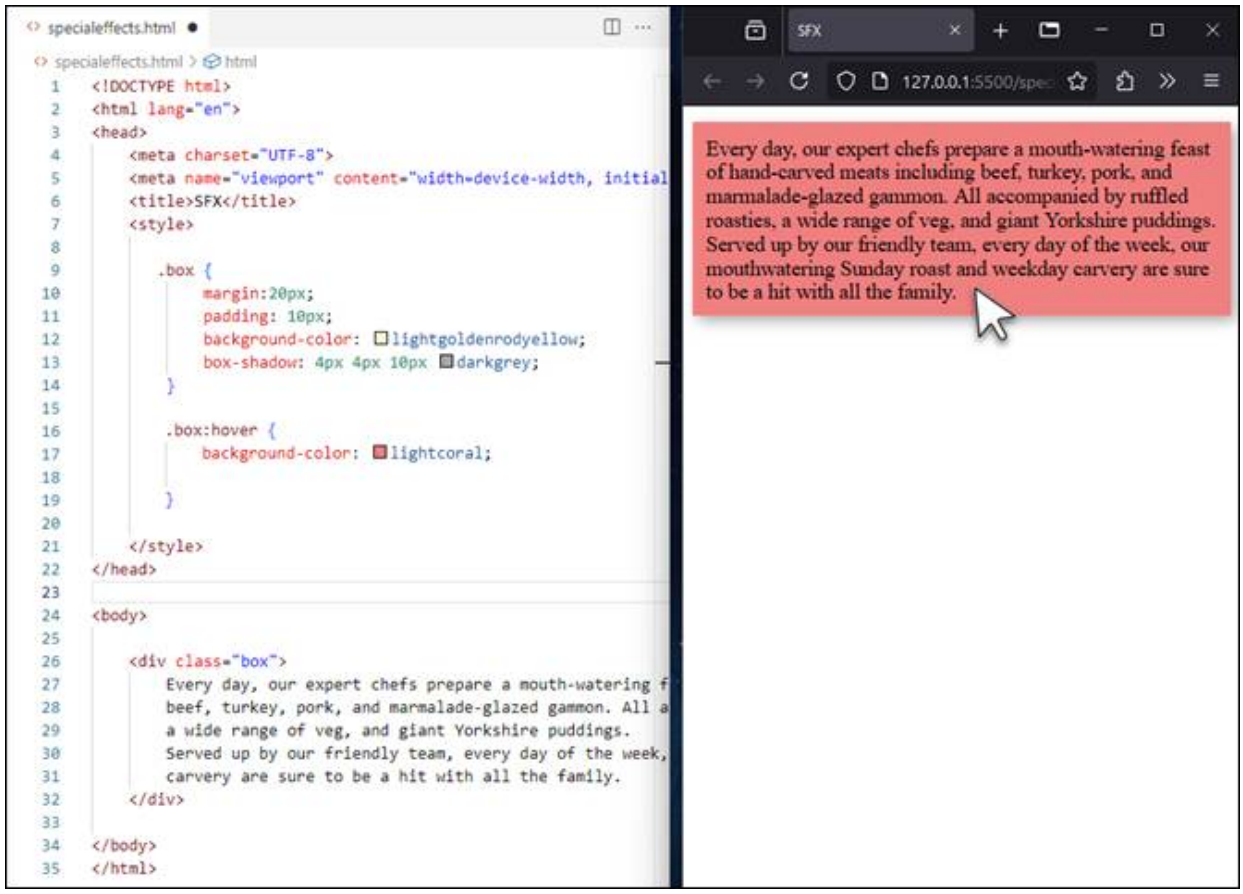


***Figure 7.19**: Using a pseudo-class*

Transitions

Transitions allow you to change property values smoothly. For example, if you notice in the previous example, the color changes instantly. We can add a transition to make the background color of the box fade in gradually. To do this, we can use the transition property, as:

```
transition: background-color 2s;
```

In *Figure 7.20*, the transition property has been added to the hover state of the box. Now, when you hover the mouse pointer over the box, the color will transition to lightcoral over 2 seconds:

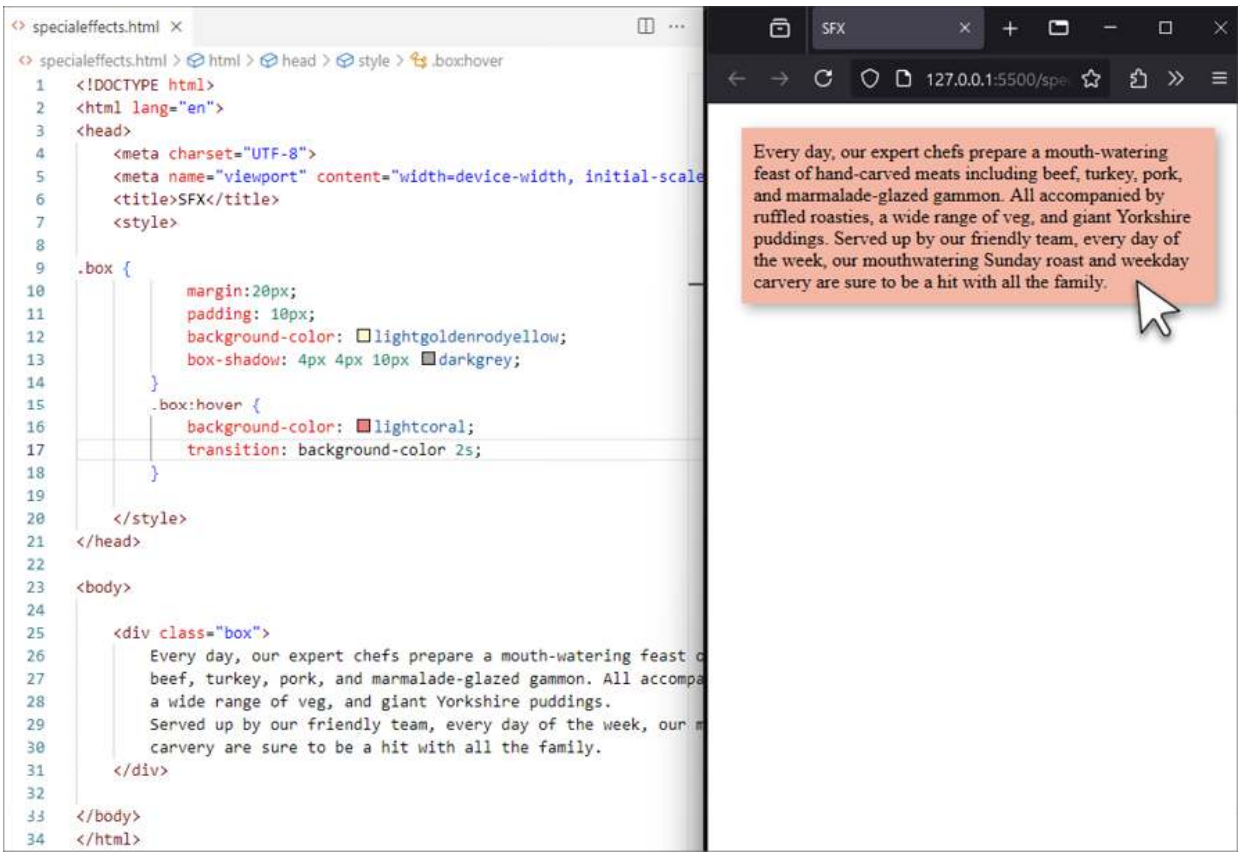


***Figure 7.20**: Creating a transition with pseudo-class*

Animations

Animation changes an element's style over time and uses the **`@keyframes`** rule to define the steps in the animation.

You can do this using two components: the **`@keyframes`** rule and the animation property.

The **`@keygrames`** rule defines the sequence of steps in the animation and specifies how the element's style changes. Each keyframe defines the state at a given percentage of the animation's duration (e.g., 0%, 50%, 100%).

The animation property is applied to the element that you want to animate. The following table shows the most commonly used animation properties

along with a brief description and example of how to implement them:

Property	Description	Example
animation-name	Specifies the name of the @keyframes animation to apply.	`animation-name: slide;`
animation-duration	Defines how long the animation takes to complete one cycle.	`animation-duration: 2s;`
animation-timing-function	Specifies the speed curve of the animation. Options include ease, linear, ease-in, ease-out, etc.	`animation-timing-function: ease;`
animation-delay	Delays the start of the animation by the specified time.	`animation-delay: 1s;`
animation-iteration-count	Specifies how many times the animation should be played. Can be a number or infinite for looping.	`animation-iteration-count: 3;`
animation-direction	Defines whether the animation runs forward, backward, or alternates. Options: normal, reverse, alternate, alternate-reverse.	`animation-direction: alternate;`
animation-fill-mode	Specifies how styles are applied before and after the animation. Options: none, forwards, backwards, both.	`animation-fill-mode: forwards;`
animation-play-state	Allows pausing and resuming the animation. Options: running, paused.	`animation-play-state: paused;`
animation	Shorthand property for all animation properties: name, duration, timing-function, delay, iteration-count, direction, fill-mode, play-state.	`animation: slide 2s ease-in-out infinite;`

***Table 7.5**: Most commonly used animation properties*

Let us take a look at an example. Have a look at **`ani.html`**. In *Figure 7.21*, the animation continuously changes the background color and position of the box over time.

In the CSS code at the top, the **`.box`** class defines the basic appearance and animation properties.

The key animation properties start with the animation-name, which specifies the name of the animation to apply, the animation-duration, which is how long the animation will take to complete one full cycle (4s in this case), and

the **`animation-iteration-count`**, which is how many times the animation will run (loop continuously in this case).

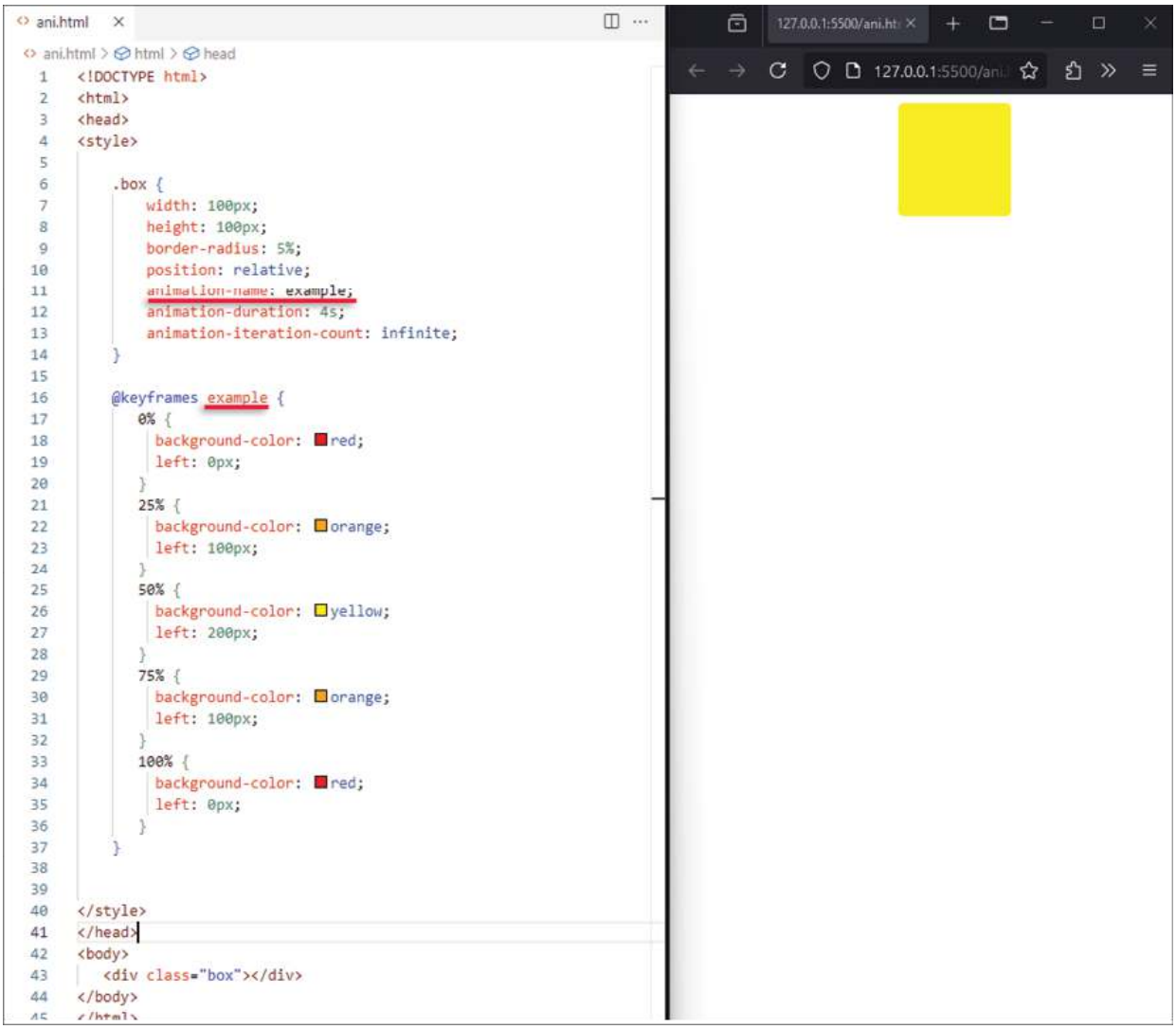


***Figure 7.21**: A simple animation*

The animation itself is defined by the **`@keyframes`** rule. This rule sets the key points in time where the animation changes. At the start of the animation (0%), the box has a red background color and is positioned at the leftmost side (**`left: 0px`**). As the animation progresses, at 25%, the background changes to orange, and the box moves 100 pixels to the right. At 50%, the box turns yellow and moves 200 pixels to the right. By 75%, the box shifts back to orange and moves back toward the left to 100 pixels. Finally, at 100%, the box returns to its starting position, both in terms of color (red) and

location (`left: 0px`).

When the animation runs, you will see the box move left and right across the screen, changing color as it moves.

Using the box model

The CSS box model is a concept in web design that defines how elements are structured and displayed on a web page. The model consists of four components: the content, padding, border, and margin, as shown in *Figure 7.22*:

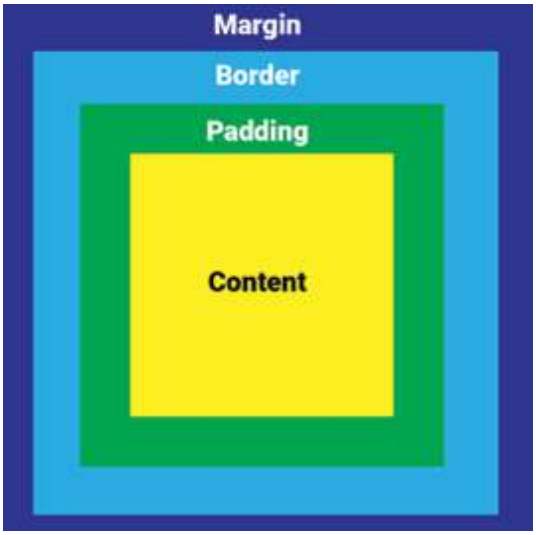


***Figure 7.22**: The box model*

The content is the innermost area of the box that holds text, images, and other content.

The padding is the space between the content and the element's border.

The border surrounds the padding and content.

The margin is the outermost layer that creates space between elements, as shown in the following figure:

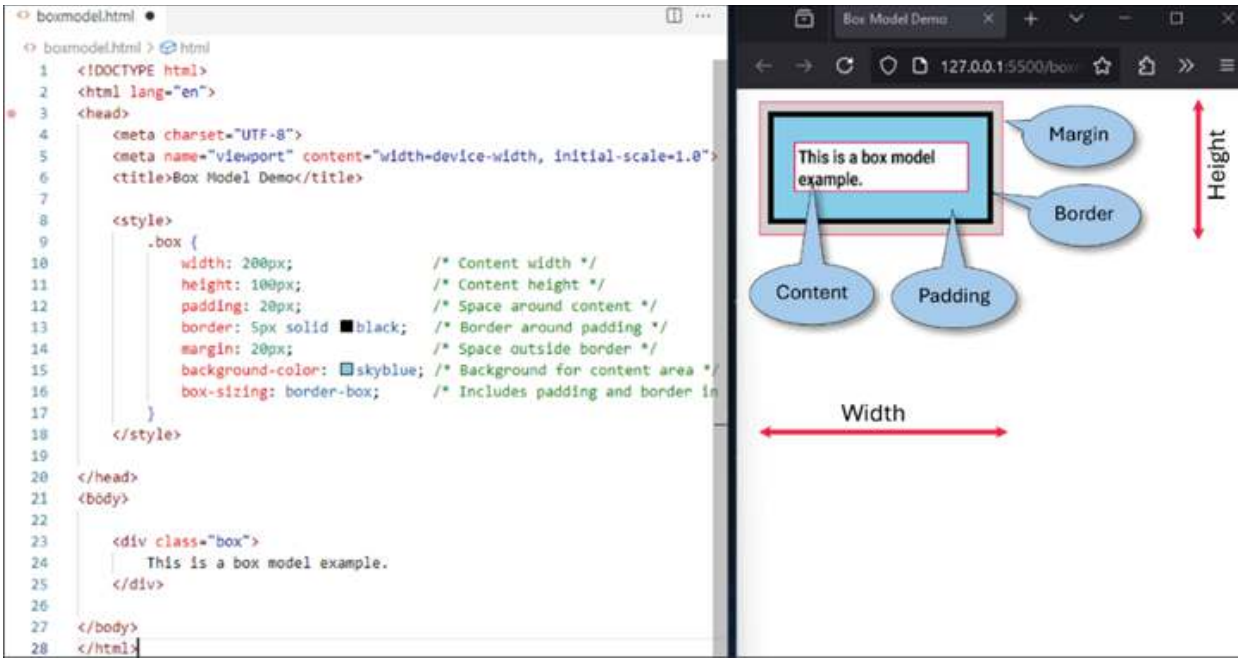


***Figure 7.23**: Box model in use*

Understanding display properties

The display property determines how an element is displayed on the page. It has several possible values, with the most common ones being block, inline, and inline-block.

A block-level element takes up the full width of its parent container, and each element starts on a new line and is stacked vertically. Elements, such as **`<div>`**, **`<h1>`**, and **`<p>`**, are block-level by default. In *Figure 7.24*, we see that each list item is displayed as a block, taking up the full width and stacked vertically:

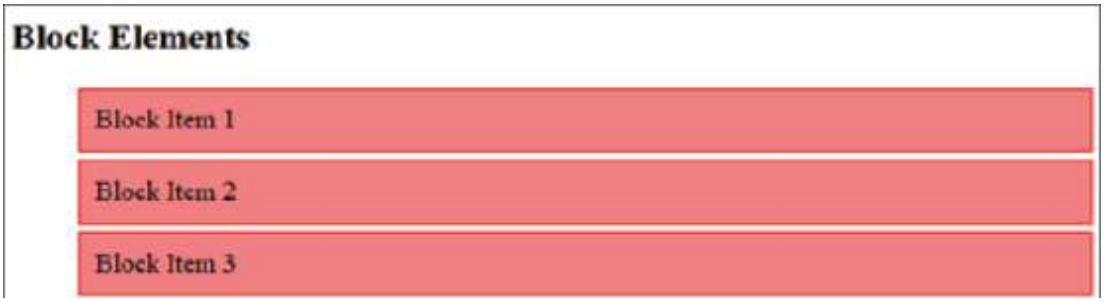


***Figure 7.24**: Block elements*

Inline elements take up only the space they need and appear on the same line, but do not accept width or height properties. Elements, such as `<span>`, `<strong>`, `<a>`, and `<em>`, are inline by default. In *Figure 7.25*, we see that each list item is displayed inline, taking only as much width as necessary and flowing horizontally:

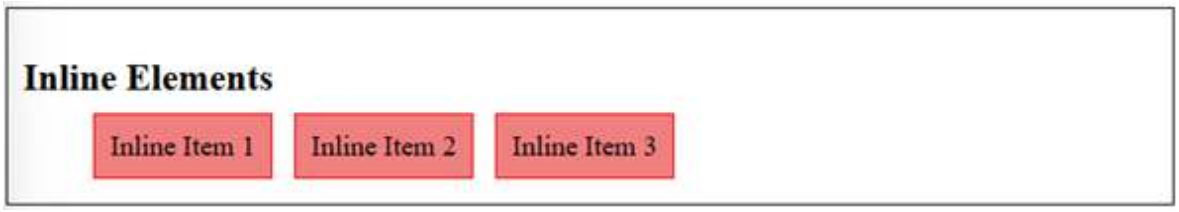


***Figure 7.25**: Inline elements*

Inline-block elements are similar to inline elements in that they appear on the same line. However, you can set width and height on inline-block elements. Elements, such as `<img>` and `<button>`, are inline-block by default. In *Figure 7.26*, we see that each list item is displayed inline-block, taking up a fixed width, displayed horizontally, but retaining block-level features:

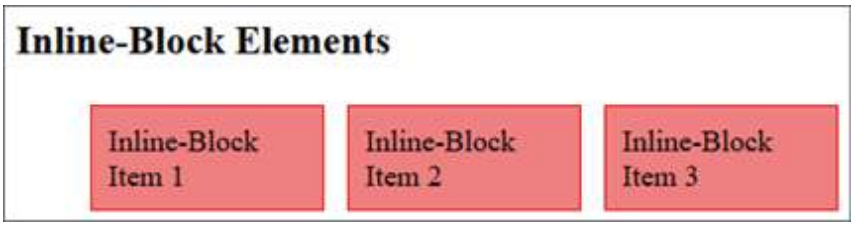


***Figure 7.26**: Inline-block elements*

One of the most common uses of inline-block is to display list items (`<li>`) horizontally instead of vertically, which is especially useful for navigation bars or menus.

Introduction to Sass

Syntactically Awesome Stylesheets (**Sass**) is a CSS preprocessor that introduces advanced features, such as variables, nesting, mixins, inheritance, and functions. By using Sass, developers can write cleaner, more organized,

and reusable stylesheets, which can drastically reduce the complexity of maintaining large projects.

With Sass, you can use variables to store values such as colors, fonts, or any CSS property, which makes updating styles easier.

You can write CSS code in a nested format, mirroring the HTML structure, and you can split the code into smaller, reusable files called partials. Sass allows you to create reusable blocks of code called **mixins** that can be included anywhere.

You can also create custom functions to generate dynamic CSS values.

To use Sass, you will need to install a few tools. Download and install Node.js (which includes npm). Then open your command prompt to install Sass, as:

```
npm install -g sass
```

Once installed in VS Code, create your Sass file with the file extension **`.scss`**. This is where you will write your Sass code. For example:

`sass.scss`

To compile the code, open the command prompt and type the following. You will need to be in the directory where these files are saved.

```
sass --watch sass.scss:sass.css
```

This command tells Sass to watch the **`Sass.scss`** file and compile it into **`sass.css`**. Every time you save the **`.scss`** file, the CSS file will automatically be updated, as shown in *Figure 7.27:*

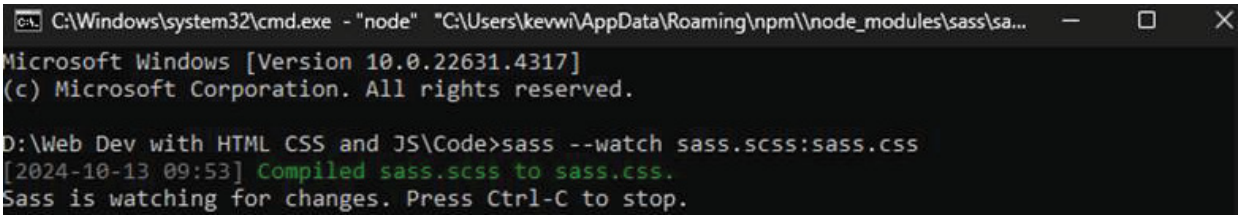


***Figure 7.27**: Sass compiler*

In the SCSS file, you can create your code. Have a look at **`sass.html`** and **`sass.scss`** as:

```
// Variables
$primary-color: #3498db;
```

```
$secondary-color: #2ecc71;
$font-stack: Arial, sans-serif;
// Mixin
@mixin button-style($bg-color) {
  background-color: $bg-color;
  padding: 10px 20px;
  border: none;
}
```

Conclusion

CSS plays a fundamental role in web development, allowing control over the visual presentation of HTML elements. It enables developers to create visually engaging and well-structured web pages through a combination of style rules. These rules can be applied from different sources, organized in a cascading manner that determines the priority of style applications. By using various selectors, properties, and values, CSS allows precise control over page layouts, colors, fonts, and spacing.

CSS allows for consistent styling across multiple web pages using external stylesheets, as well as styling using internal and inline methods. Understanding the hierarchy and specificity rules in CSS helps ensure styles are applied correctly.

The box model and display properties determine the spacing, positioning, and arrangement of elements.

In the next chapter, you will deepen your understanding of CSS layout techniques. You will learn how to position elements using CSS positioning schemes, create flexible designs with flexbox, and build advanced page structures using CSS grid layout. The chapter will also cover the use of floats and clear properties and introduce you to different CSS measurement units, providing you with the skills to build modern, responsive, and adaptive web layouts.

Points to remember

- CSS is used to define the styles and layout of HTML elements on a web page.
- Styles can be applied using external stylesheets, internal stylesheets, or inline styles.
- The term cascading refers to how styles from different sources are applied based on their specificity and importance.
- Specificity hierarchy: inline styles | ID selectors | class and attribute selectors | element selectors.
- The !important rule overrides other styles, regardless of specificity or source.
- The last rule applied principle states that the last defined style with the same specificity will take precedence.
- CSS rules consist of selectors (to target elements), properties (to define what to style), and values (to set the style).
- The universal selector (*) applies styles to all elements, while type, class, and ID selectors target specific elements.
- Inline styles apply directly to an element using the style attribute.
- Internal styles are defined within the <style> tag in the HTML <head>.
- External styles are stored in separate .css files and linked using the <link> element in the HTML <head>.
- The CSS box model consists of content, padding, border, and margin, which control the spacing and layout of elements.
- Common display properties include block, inline, and inline-block, which control element rendering behavior.

Exercises

1. How would you create a CSS rule to change the text color of all <p> elements?
2. What selector would you use to style elements with the class highlight?
3. How do you apply a background color only to elements with a specific ID?
4. What is the purpose of the !important rule in CSS?

5. Write a CSS rule to remove all default padding and margin from every element on a web page.
6. What is the specificity hierarchy for CSS selectors?
7. How can you link an external CSS file to an HTML document?
8. What is the difference between inline, internal, and external styles in CSS?
9. What is the box model?
10. What are the differences between block, inline, and inline-block display properties?
11. How can you center a block-level element horizontally using CSS?
12. What is the role of the position property in CSS?
13. Explain how CSS handles conflicting style rules for the same element.
14. What is the difference between margin and padding?
15. Write a CSS rule to apply a Google Font to all <h1> elements in a document.

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com



CHAPTER 8

CSS for Layouts

Introduction

This chapter begins with an exploration of CSS positioning techniques, including static, relative, absolute, and fixed positioning. Next, it covers flexbox for crafting flexible, adaptive layouts and CSS grid for managing complex grid-based designs. The chapter also covers traditional layout methods, such as float and clear, and concludes with strategies for aligning and centering elements to achieve precise design alignment.

Structure

The chapter covers the following topics:

- CSS positioning
- Flexbox
- CSS grid layout
- Floats and clear
- CSS measurement units

Objectives

By the end of this chapter, you will understand CSS positioning and how to use flexbox and CSS grid to create effective layouts for your website. You will also have an understanding of CSS measurement units and how to use them.

CSS positioning

Positioning refers to the way elements are arranged on a web page using the position property. The position property defines how an element is placed on a web page relative to other elements on the page. There are four commonly used values for the position property: static, relative, absolute, and fixed.

The default positioning of all elements is static. This means elements appear one after another in the order they are written in the HTML file. Static-positioned elements are not affected by the top, bottom, left, and right properties, as shown in the following figure:

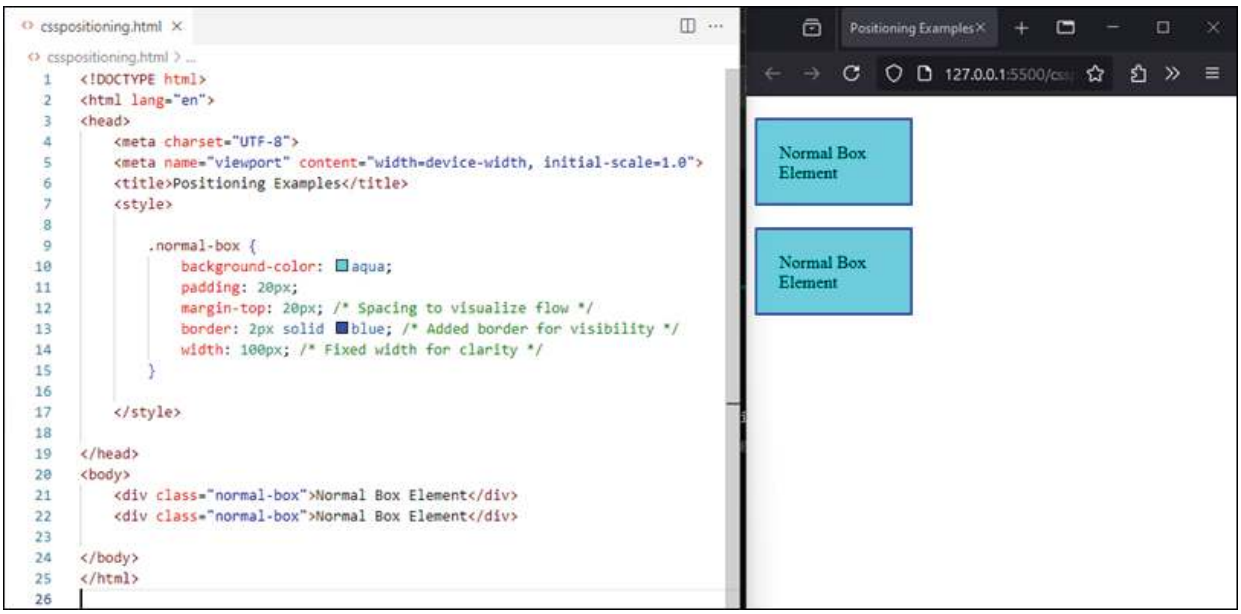


***Figure 8.1**: Static-positioned elements*

When an element is positioned relatively, it is placed relative to its original (static) position in the page flow. The original position is where it would be placed if it had static positioning. Setting the top, right, bottom, and left properties will move the element visually away from its original position, as shown in the following figure:

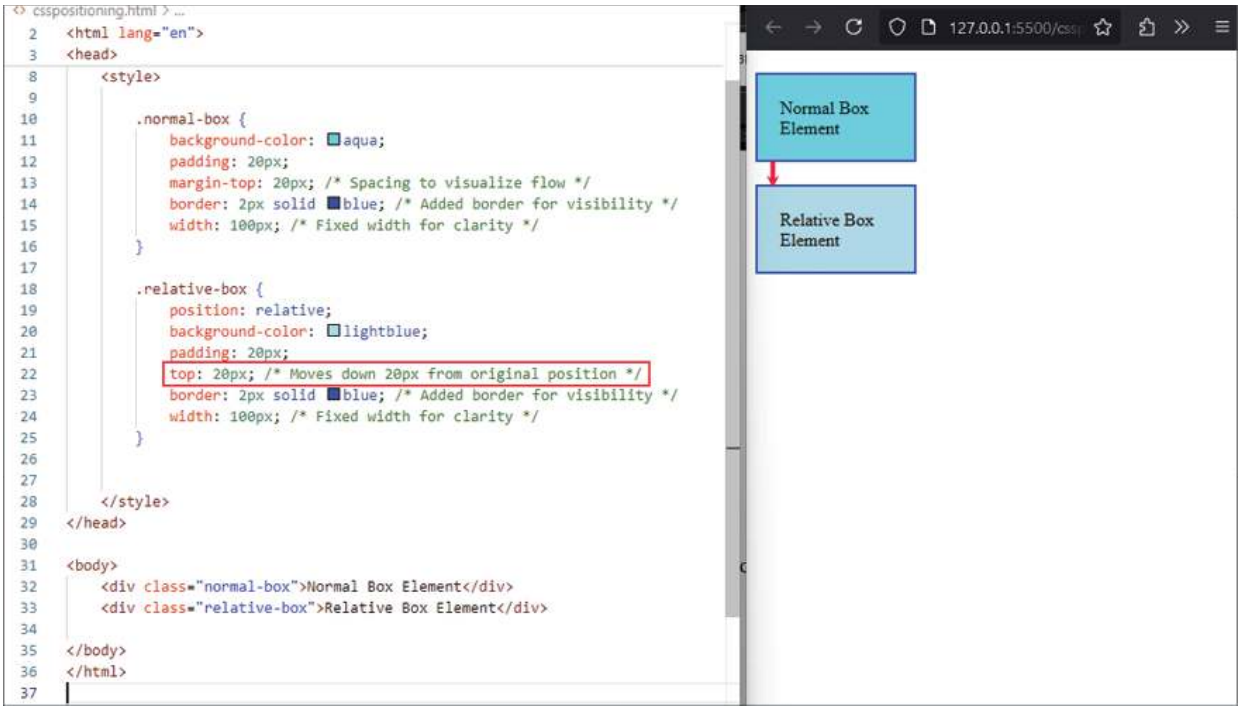


***Figure 8.2**: HTML*

For example, in *Figure 8.3*, if we move the **`Normal Box Element`** down, it will push the relatively positioned **`Relative Box Element`** down as well, because the **`Relative Box Element`** still occupies space in the flow:

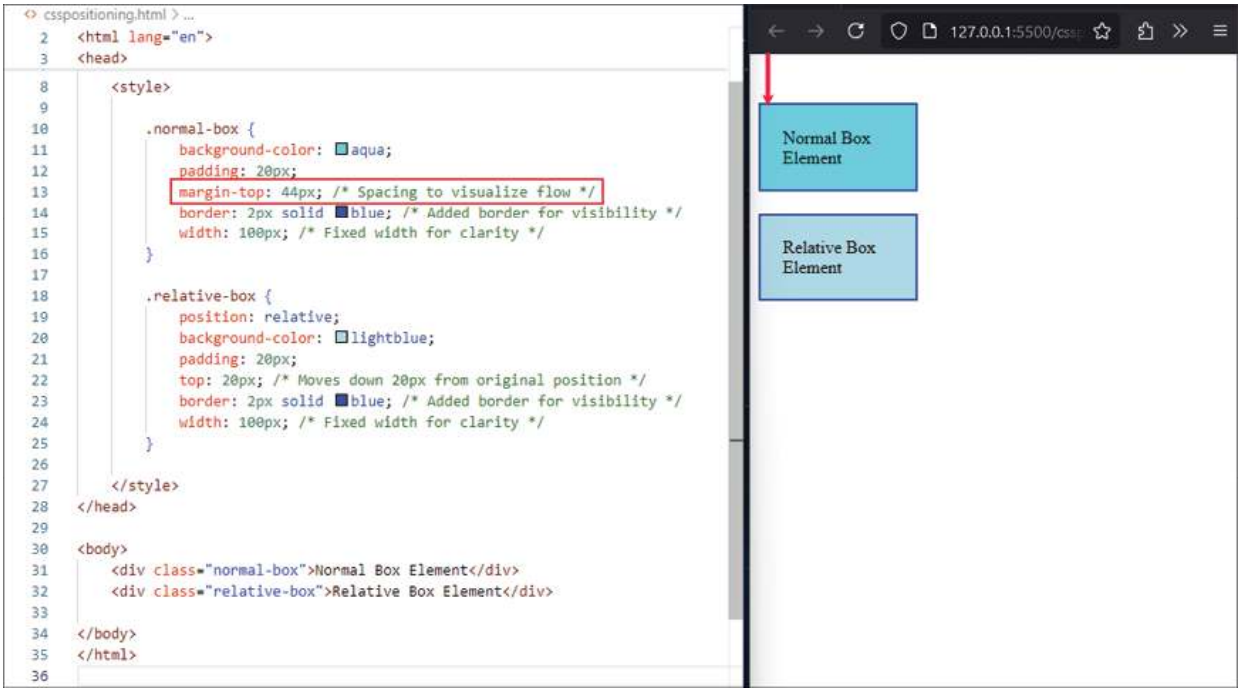


***Figure 8.3**: Relative Box Element*

When an element is positioned as absolute, it is removed from the normal document flow. This means it does not take up space in the layout anymore, and other elements will ignore its presence, behaving as if it is not there, as shown in the following figure:

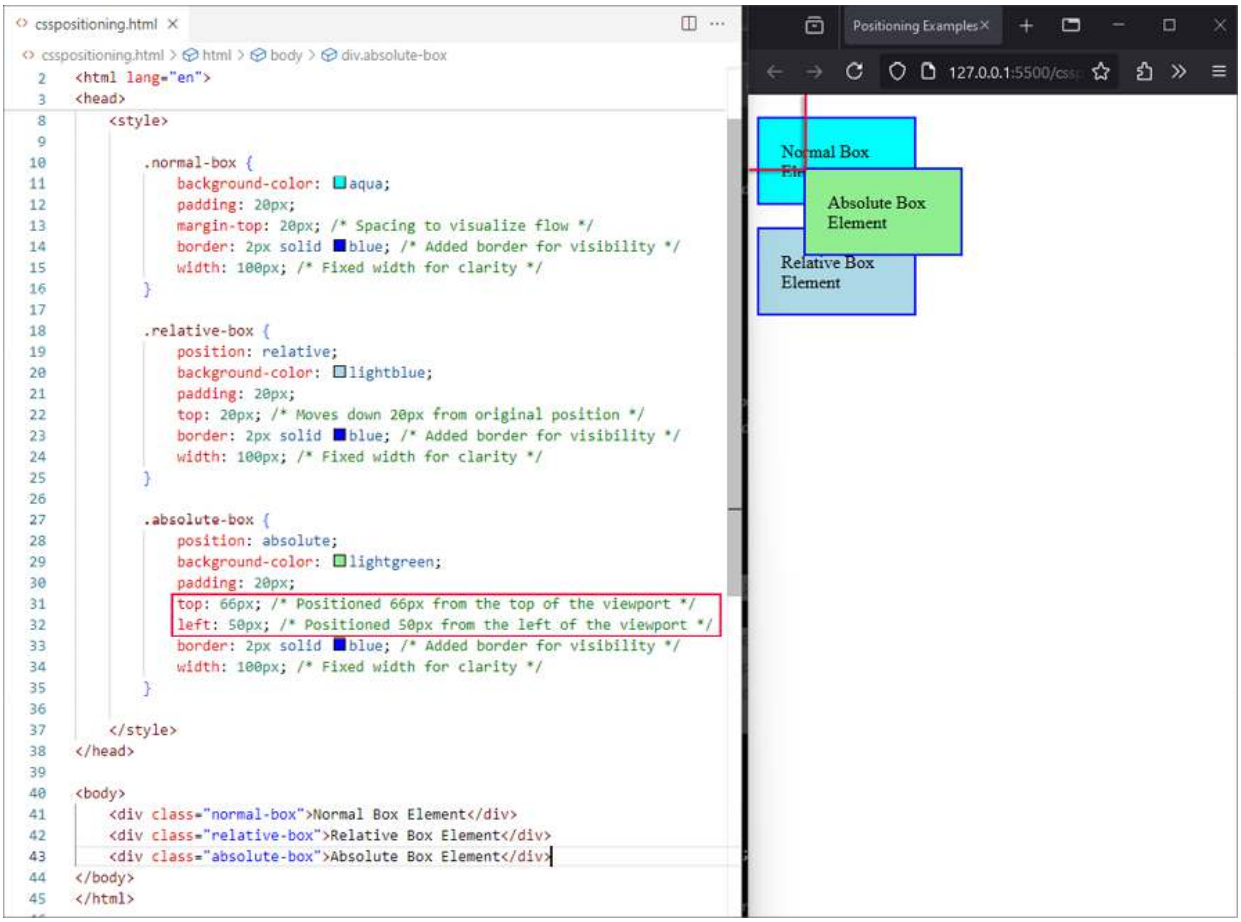


***Figure 8.4**: Element positioning*

An element with fixed positioning is removed from the normal document flow and positioned relative to the browser window. It stays fixed in its position even when the page is scrolled, as shown in the following figure:

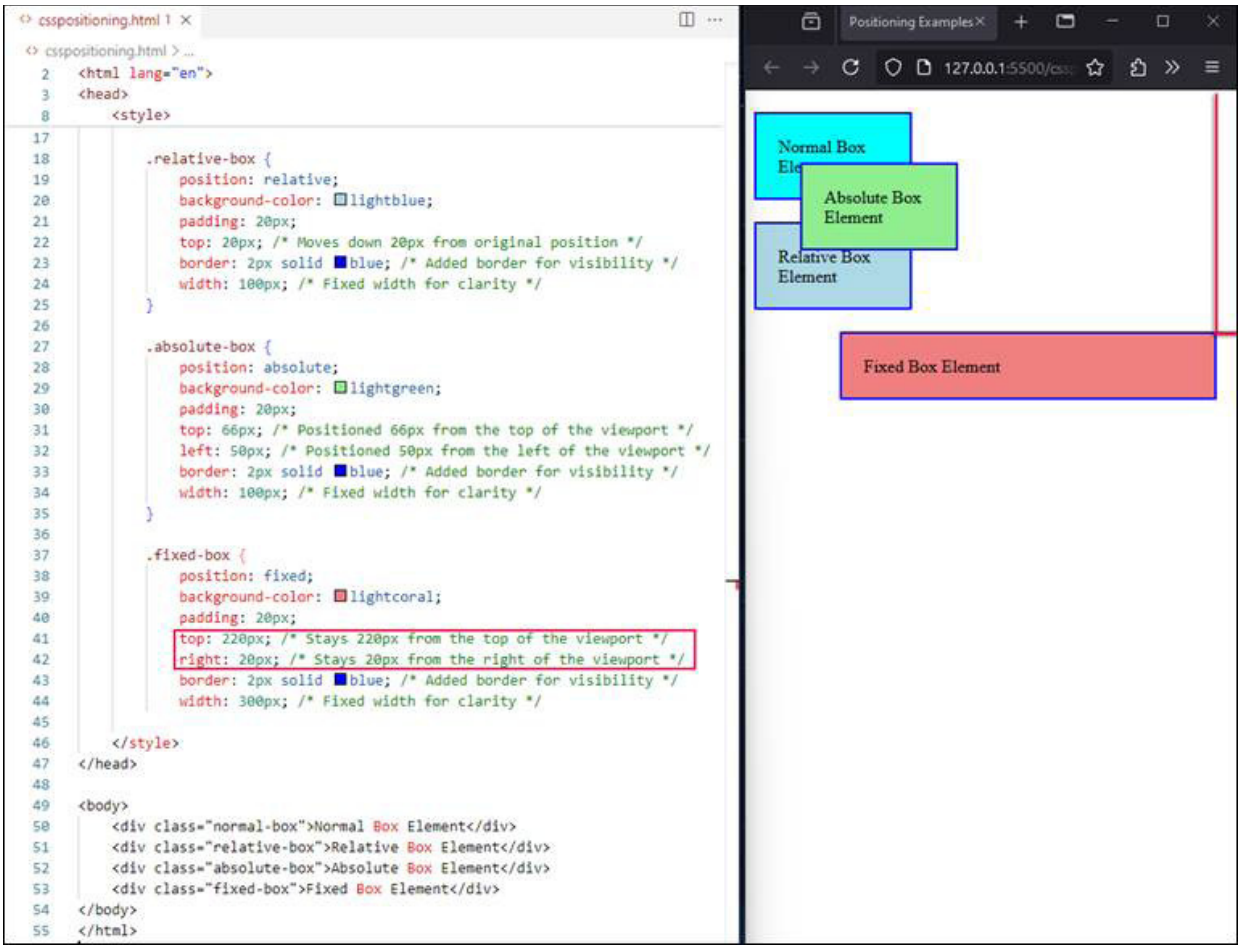


***Figure 8.5**: Fixed positioning*

Flexbox

A layout model that allows you to align and distribute elements within a container along a single axis when the size is unknown or dynamic. It simplifies creating responsive and flexible layouts, as shown in the following figure:

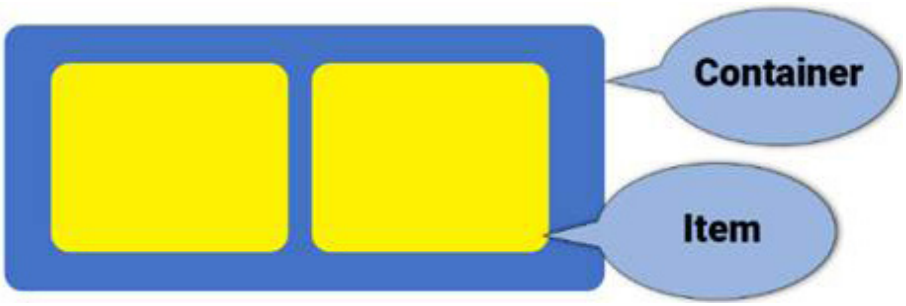


***Figure 8.6**: Layout model*

We can create a container in our CSS code like the following:

```
.flex-container {
  display: flex;
}
```

The flex property in the display element specifies how a flex item will grow or shrink to fit the space available in its container.

The elements within a flexbox container can be laid out in any direction and can dynamically adjust their sizes. We can change the container direction to span rows or columns, as shown in the following figure:

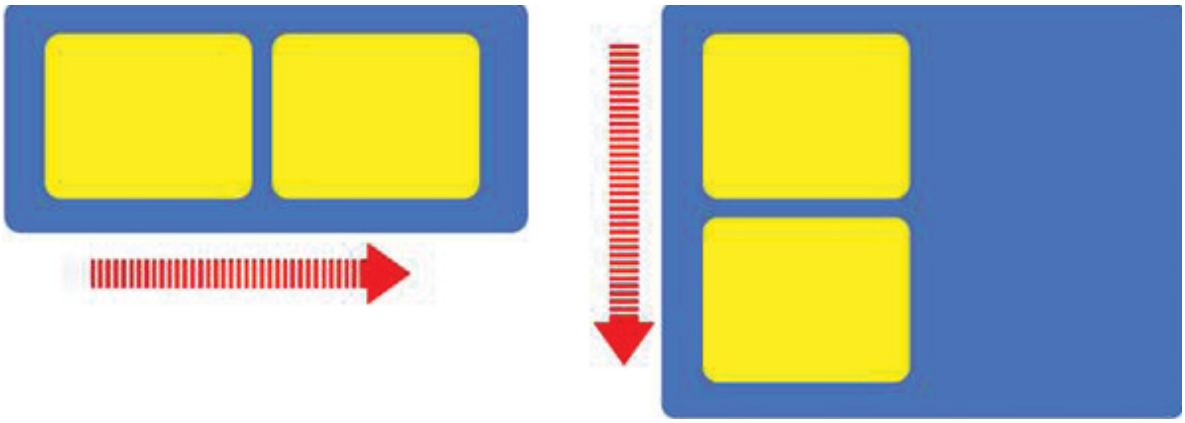


***Figure 8.7**: Changing container direction*

When items are arranged side by side in a horizontal row, this is the default layout behavior of a flexbox container, as shown in the following figure:



***Figure 8.8**: Default layout behavior*

To do this, we use **`flex-direction: rows`**, as:

```
.flex-container {
  display: flex;
```

```
  flex-direction: row;
}
```

Suppose we want the items to be stacked on top of each other in columns. In that case, we can change the layout direction by setting the **`flex-direction`** property to column, which arranges the flex items vertically within the container, as illustrated in *Figure 8.9*:



***Figure 8.9**: Vertical placement*

To do this, use **`flex-direction: columns`**, as:

```
.flex-container {
  display: flex;
  flex-direction: column;
}
```

We can specify whether we want the items to wrap or not. Here, the items will wrap to the next row if the screen size is smaller:

```
.flex-container {
  display: flex;
  flex-direction: row;
  flex-wrap: wrap;
}
```

This allows elements to shift position to fill space, grow to fill unused space, or shrink based on the container's available space and the properties applied to each element, as shown in the following figure:

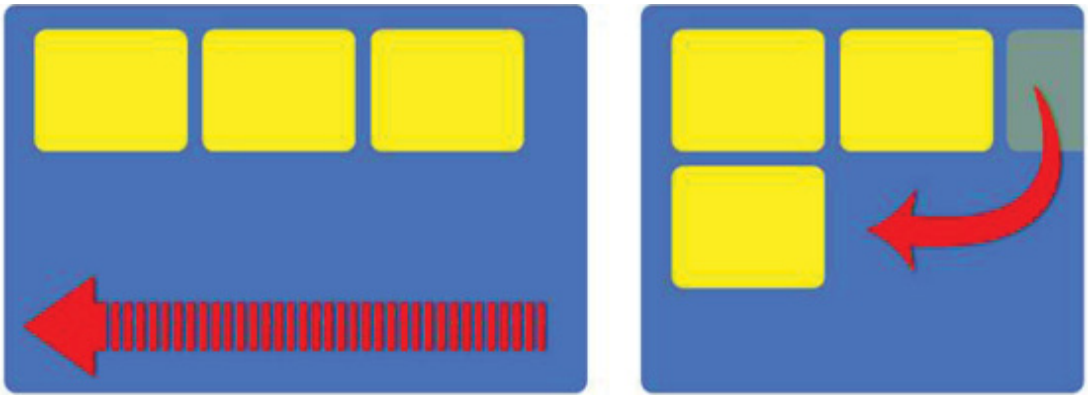


***Figure 8.10**: Elements shifting position*

Let us take a look at an example. Have a look at **`flexbox.css`** and **`flexbox.html`**. In *Figure 8.11*, a flexbox container has been created and three items: element 1, element 2, and element 3. The flow direction is set to rows, and you can see the items arranged in a row in the browser window:

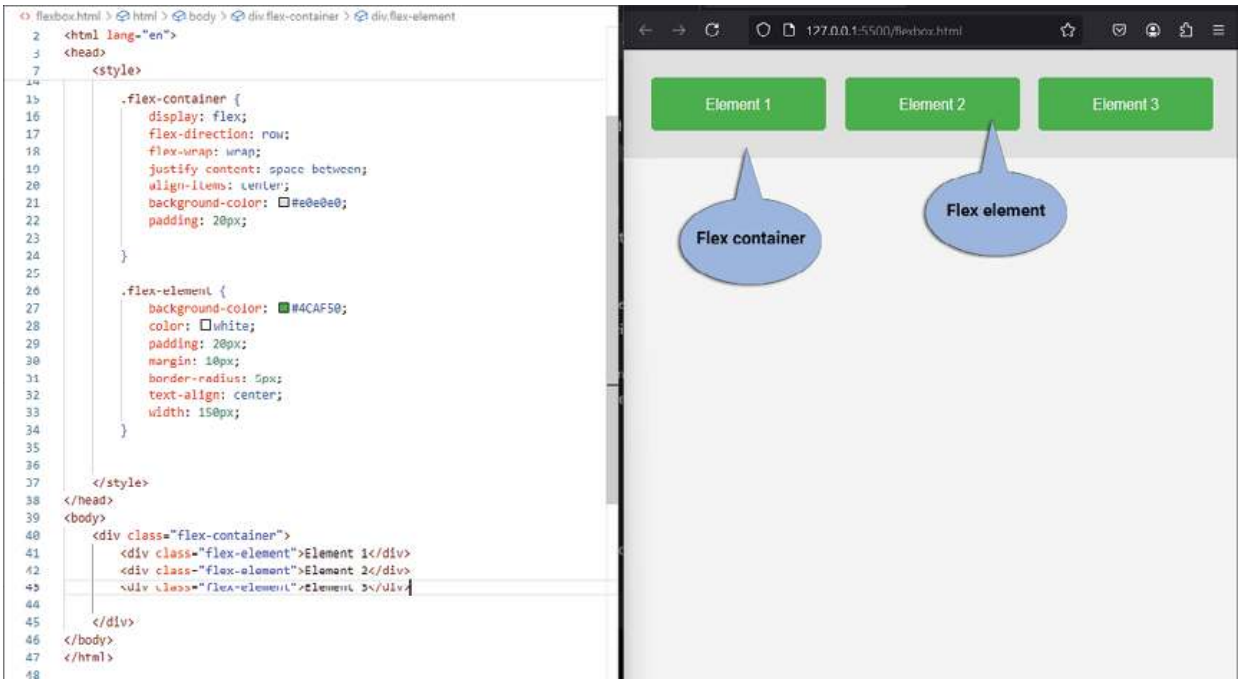


***Figure 8.11**: Flexbox container*

Now, when we resize the browser window, for example, make it smaller, you can see the flexbox items wrap, as shown in the following figure:

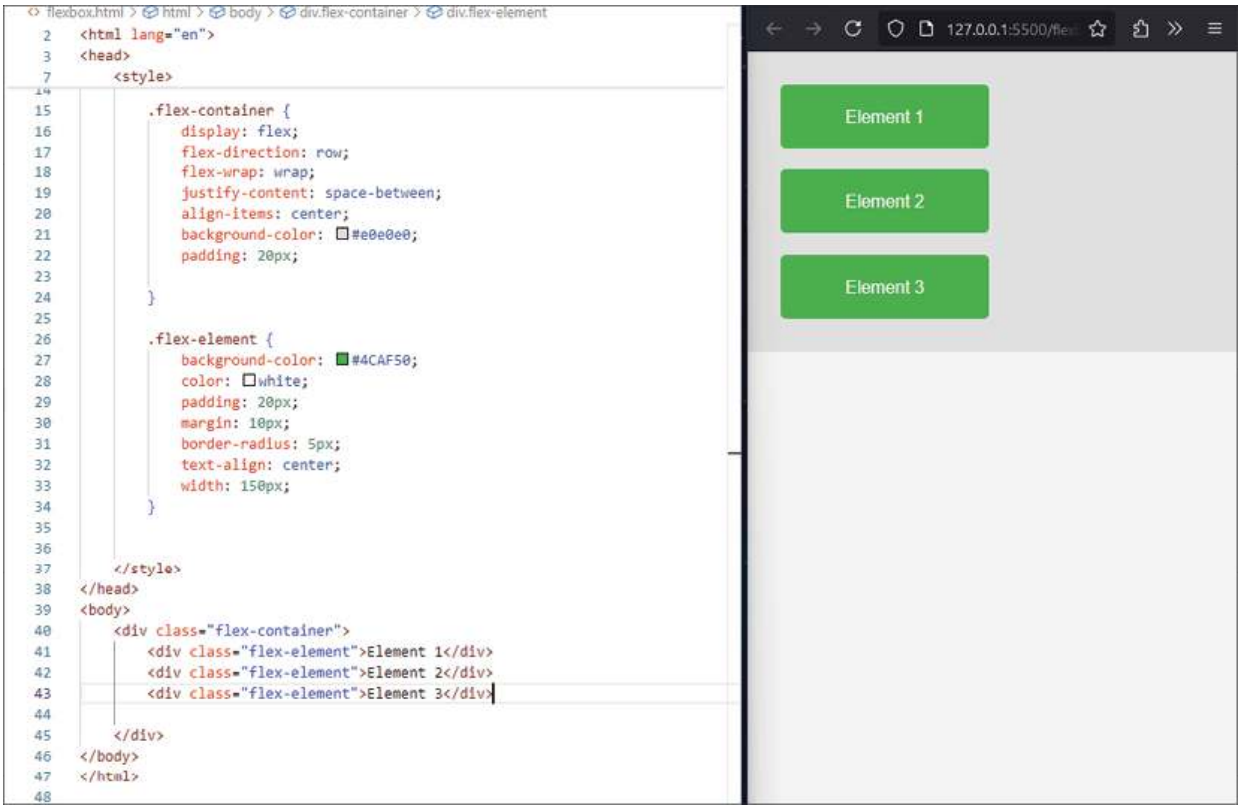


***Figure 8.12**: Flexbox items wrap*

CSS grid layout

CSS grid layout, often referred to as CSS grid, is a layout system that allows you to create complex, responsive web designs by defining rows and columns within a container.

The vertical lines are called **columns**, and the horizontal lines are called **rows**. The spaces between each column/row are called **row gaps**. Grid items, such as images or text, can be aligned along the rows and columns of the grid and are called **grid items**, as shown in the following figure:

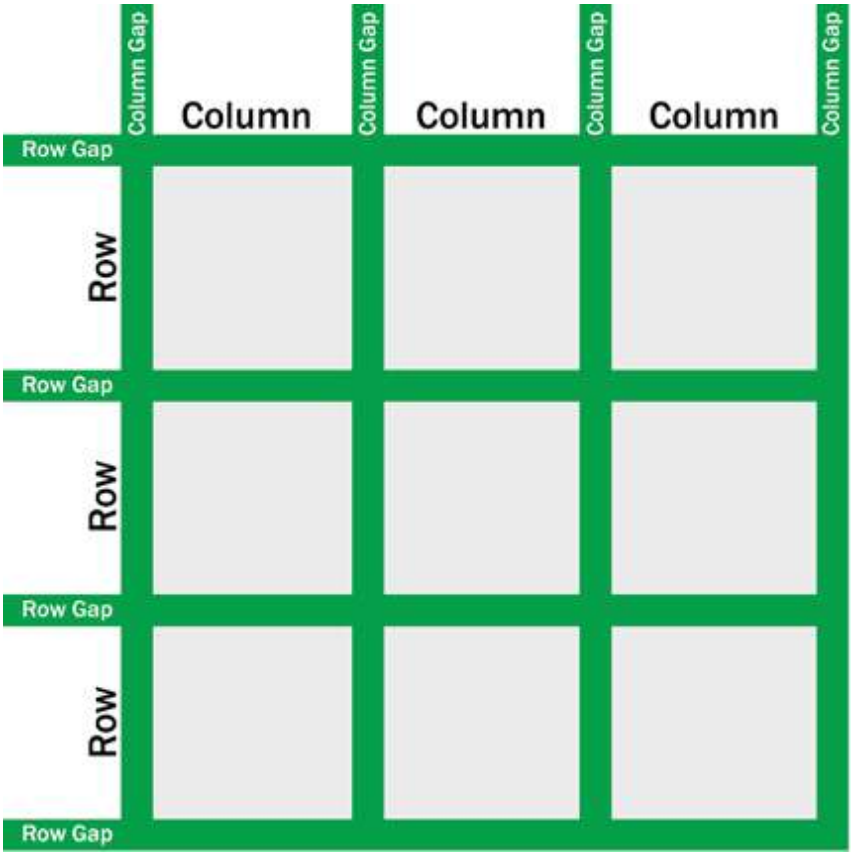


***Figure 8.13**: Grid layout*

Let us take a look at an example. Have a look at **`cssgrid.css`** and **`cssgrid.html`**. In this example, we want to lay out our website as follows. We want the width to span four columns. This means the header and the footer will span all four columns.

We also want the content to fill three columns (col 1, col 2, col 3).

The sidebar will take up the last column (col 4).

The site will also be split into four rows. The header on row 1, the nav bar on row 2, the content and sidebar on row 3, and the footer on row 4.

This layout can be seen in the following figure:

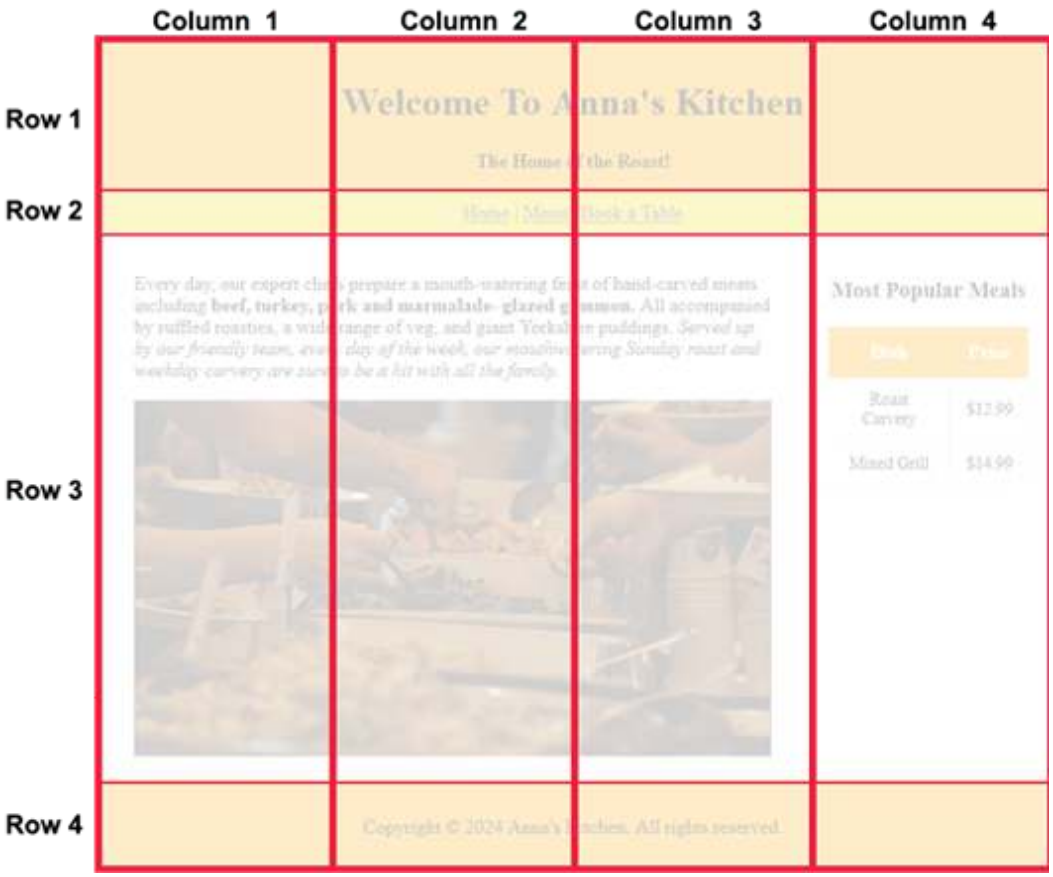


***Figure 8.14**: Layout*

First, to build the grid shown in the preceding figure, we need to create a grid container with four columns and four rows. The **`display: grid`** property sets the layout mode of the **`<body>`** element to a grid.

Next, we create the columns and rows. The property **`grid-template-columns: 1fr 1fr 1fr 1fr;`** creates four columns of equal width within the grid. The unit *fr* stands for *fractional unit* and is used to allocate a proportion of the available space within the grid container. Here, each column takes up **`1fr`** (1 fraction) of the total space.

The property **`grid-template-rows: auto auto auto auto;`** creates four rows of equal height within the grid. Here, we use the auto keyword to allow each row to automatically adjust its height based on the content, as:

```
body {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr 1fr; /* 4 columns
```

```
*/
    grid-template-rows: auto auto auto auto;    /* 4
rows */
  }
/* Header styling */
header {
    grid-column: 1 / 5; /* Span all 4 columns */
    grid-row: 1 / 2;    /* Occupy the first row */
    text-align: center;
    padding: 20px;
    background-color: orange;
}
/* Navigation styling */
nav {
    grid-column: 1 / 5; /* Span all 4 columns */
    grid-row: 2 / 3;    /* Occupy the second row */
    text-align: center;
    padding: 10px;
    background-color: gold;
}
/* Main content area styling */
main {
    grid-column: 1 / 4; /* Span columns 1 to 3 */
    grid-row: 3 / 4;    /* Occupy the third row */
    padding: 20px;
}
/* Sidebar styling */
aside {
    grid-column: 4 / 5; /* Occupy column 4 */
    grid-row: 3 / 4;    /* Occupy the third row */
    padding: 20px;
    background-color: lightgoldenrodyellow;
}
```

```
/* Footer styling */
footer {
    grid-column: 1 / 5; /* Span all 4 columns */
    grid-row: 4 / 5;    /* Occupy the fourth row */
    text-align: center;
    padding: 20px;
    background-color: orange;
}
```

Now, we can build the web page using HTML. To do this, all we have to do is use the **<header>**, **<nav>**, **<main>**, **<aside>**, and **<footer>** elements we defined in our CSS to mark the different parts of the web page, as:

```
    <!-- Header Section -->
    <header>
        <h1>Welcome To Anna's Kitchen</h1>
        <p>The Home of the Roast!</p>
    </header>
    <!-- Navigation Section -->
    <nav>
        <a href="index.html">Home</a> |
        <a href="menu.html">Menu</a> |
        <a href="book.html">Book a Table</a>
    </nav>
    <!-- Main content Section -->
    <main>
        <p>
            Every day our expert chefs...
        </p>
        <video>
            ...
        </video>
    </main>
    <!-- Sidebar Section -->
    <aside>
```

```
        <h3 style="text-align: center;">Most Popular
Meals</h3>
        <table>
            <tr>
                <th>Dish</th>
                <th>Price</th>
            </tr>
            <tr>
                <td>Roast Carvery</td>
                <td>$12.99</td>
            </tr>
            <tr>
                <td>Mixed Grill</td>
                <td>$14.99</td>
            </tr>
        </table>
    </aside>
    <!-- Footer Section -->
    <footer>
        <p>Copyright &copy; 2024 Anna's Kitchen. All
rights reserved.</p>
    </footer>
```

We will end up with something like the following:

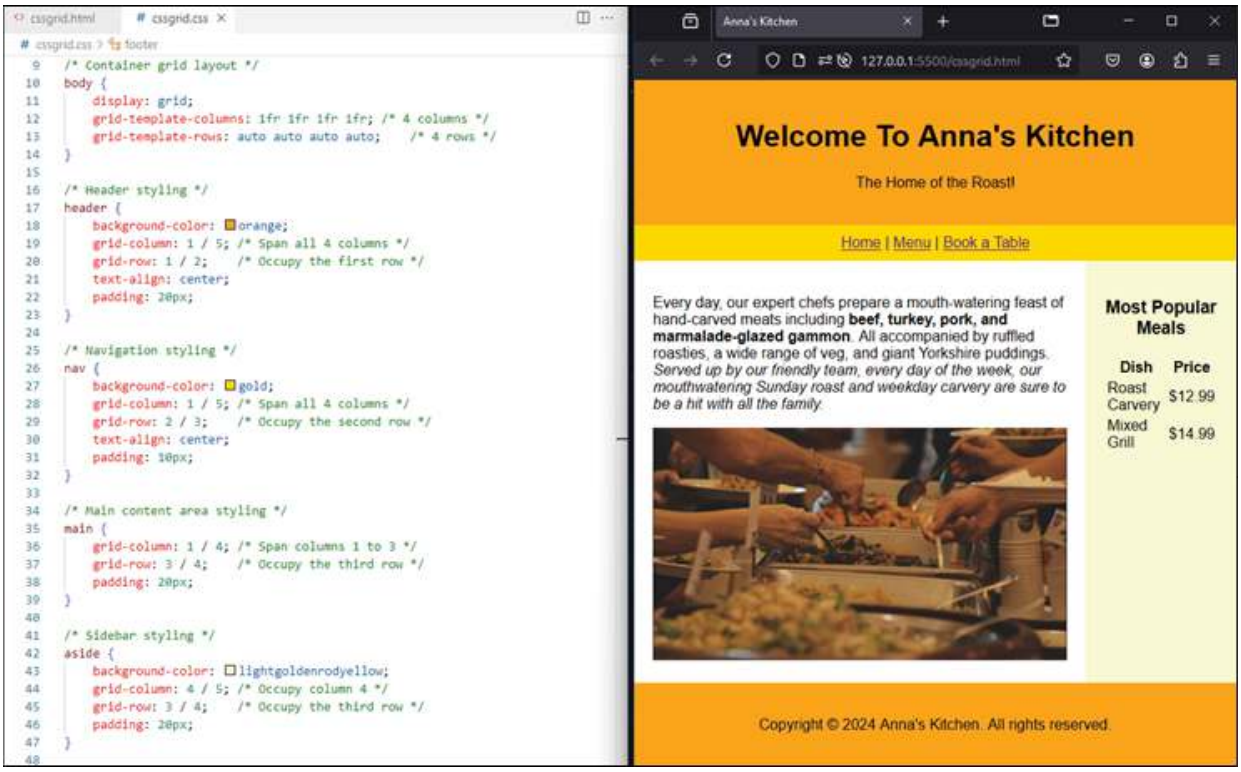


***Figure 8.15**: Property grid template*

Float and clear

The float property was originally used to wrap text around images, although it is used less frequently today thanks to flexbox and grid.

Floats take an element out of its normal document flow and allow other content to wrap around it. You can set the float attribute to left, right, none, or inherit.

In the following code, the image will float to the right of its container, and any adjacent text will wrap around it:

```
img {
  float: right;
}
```

The clear property prevents elements from floating around other floated elements, essentially forcing them below the floating element.

The float and clear properties are not commonly used for modern layouts due to the limitations and complexities they introduce. It is much better to

use flexbox or grid.

CSS measurement units

To control the size of certain objects, such as fonts or images, the sizes are specified using various units. There are two types: absolute and relative.

Units that are absolute are the same size regardless of the parent element or window size. The following table discusses some examples:

Unit	Meaning
in	Inch
mm	Millimeter
px	Short for pixels and is usually used to measure the dimensions of an image.'
pt	Short for points and is used to measure the size of a font.

***Table 8.1**: CSS measurement units*

Units that are relative scale in relation to the parent element or window size, depending on the unit used. The following table discusses some examples:

Unit	Meaning
em	Relative to the current font size of the element. If a font is set to 12pt, each em unit would be 12pt, so 2em would be 12 x 2, which is 24pt; similarly, 1.5em would be 18pt.
%	Relative to the parent element or window size.
fr	Stands for a fractional unit and is used for creating flexible and responsive layouts. 1fr means 1 fraction of the total space.

***Table 8.2**: Relative scale units*

Conclusion

A solid understanding of CSS positioning and layout techniques is essential for building well-structured, visually coherent, and responsive websites. In this chapter, you explored several critical layout models and methods used in modern web development. You began with an overview of CSS positioning strategies—static, relative, absolute, and fixed—each offering different ways to control the placement and behavior of elements within a page.

You then examined flexbox, a one-dimensional layout model ideal for aligning and distributing elements either in rows or columns, especially when the container's dimensions are flexible or unknown. This was followed by an introduction to CSS grid, a two-dimensional layout system that allows you to design complex grid structures with explicit control over both rows and columns.

The chapter also revisited traditional techniques like float and clear, illustrating their legacy use and limitations, and introduced CSS measurement units—both absolute (e.g., px, pt) and relative (e.g., %, em, fr)—which help define size, spacing, and proportions in adaptive layouts.

By mastering these layout systems, you can create web pages that are visually balanced, scalable across devices, and easier to maintain.

In the next chapter, we will extend these concepts by focusing on responsive design. You will learn how to adapt layouts for various screen sizes using media queries, implement mobile-first design strategies, and use advanced techniques such as calc(), min(), and max() functions to create fluid, adaptable layouts. You will also explore how to test and refine your designs to ensure they perform effectively across a wide range of devices.

Points to remember

- The default position for all elements is static, meaning elements follow the normal document flow.
- Relative positioning moves an element based on its original position without affecting the layout of other elements.
- Absolute positioning removes an element from the document flow and positions it relative to its nearest positioned ancestor.
- Fixed positioning removes an element from the flow and positions it relative to the browser window, remaining in place even when scrolling.
- Flexbox is used for aligning and distributing elements along a single axis (row or column).
    - Use flex-direction to control the direction of flexbox items (row or column).
    - Use flex-wrap to allow flexbox items to wrap onto the next line if

there is not enough space.
    - Flexbox is best suited for one-dimensional layouts, while CSS grid is more effective for two-dimensional layouts involving both rows and columns.
- CSS grid provides a structure for creating layouts using rows and columns.
    - grid-template-columns and grid-template-rows define the number and size of columns and rows in a grid.
- Floats take elements out of the normal flow and allow other content to wrap around them.
- Use the clear property to prevent elements from wrapping around floated items.
- em, %, and fr units are used to size elements relative to parent elements, container size, or available space.

Exercises

1. What is the default position value for all HTML elements?
2. How does the relative position value affect an element's placement compared to static?
3. Create a div element with a relative position. Set its top and left properties to move it 20px down and 30px to the right. What happens to its original position in the document flow?
4. Create a div with an absolute position inside another container with relative positioning. How does the absolute element's position change when you resize the browser window?
5. Use fixed positioning to place a navigation bar at the top of the page. How does this element behave when scrolling?
6. Create a simple web page using flexbox. Set up a container with three items and arrange them in a column. How does the layout change when you switch to row direction?
7. Modify the flexbox container to wrap its items when the browser width becomes too narrow. How does the wrapping affect the item layout?

8. Create a CSS grid with four columns and three rows. Place a header spanning all four columns at the top and a footer spanning all four columns at the bottom. Place a sidebar in the first column and content spanning the remaining columns in between. How does this structure behave when adding more rows?
9. Experiment with the grid-template-columns property. What happens when you use different units like fr, px, and %?
10. Use the float property to place an image on the left side of a block of text. How does the text wrap around the image?
11. Apply the clear property to another element below the floated image. What is the effect of setting clear: both;?
12. What are the main differences between using flexbox and grid for creating layouts?
13. Set up a container using flexbox and a similar layout using grid. Compare how each method handles responsiveness when changing the screen size.
14. Which positioning property would you use to keep an element in view even when the page is scrolled? Create a simple example to demonstrate this.
15. Experiment with em, %, and fr units to size elements in a flexbox container. How do these units affect the size and alignment of items in the container?

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com

CHAPTER 9

Responsive Design with CSS

Introduction

This chapter introduces the principles of responsive design, starting with media queries and their role in tailoring web experiences to various devices. It covers creating mobile-first designs to ensure optimal performance on smartphones and other mobile devices. You will learn how to adapt layouts effectively for different screen sizes and use viewport units and relative sizing to achieve fluid, scalable designs. Finally, the chapter provides guidance on testing responsive websites across different browsers to ensure consistency and functionality.

Structure

The chapter covers the following topics:

- Adapting layouts for different screen sizes
- Responsive layouts
- Introduction to media queries
- Creating mobile-first designs
- Testing responsive websites
- Calculating with CSS using calc(), min() and max()

Objectives

By the end of this chapter, you will understand how responsive layouts work and how they adapt to different screen sizes, as well as how to use media queries, mobile-first design, and various functions in CSS.

Adapting layouts for different screen sizes

Adapting a website to function and appear correctly across various screen sizes is a fundamental principle of modern web development known as responsive design. This approach ensures that web content is flexible, accessible, and visually consistent across a broad spectrum of devices, including smartphones, tablets, laptops, and large desktop monitors. Rather than designing multiple separate versions of a site, responsive design allows a single layout to adjust based on the user's screen dimensions dynamically.

A responsive design strategy uses flexible grids, scalable images, and CSS features such as media queries to accommodate different viewports. As a result, content may shift, resize, or reflow to provide an optimal viewing experience. For example, a multi-column desktop layout may collapse into a single-column vertical structure on a mobile device to ensure readability and usability without requiring zooming or horizontal scrolling.

As shown in *Figure 9.1*, the layout of a web page can change significantly depending on the screen size. Elements such as navigation menus, images, sidebars, and content areas reposition themselves to maintain logical structure and clarity, improving both functionality and aesthetic appeal.

Responsive layouts are not only about aesthetics—they are critical for usability, performance, and accessibility. A site that adapts properly to its environment will load faster on mobile networks, be easier to navigate with touch input, and provide a more inclusive experience to all users.

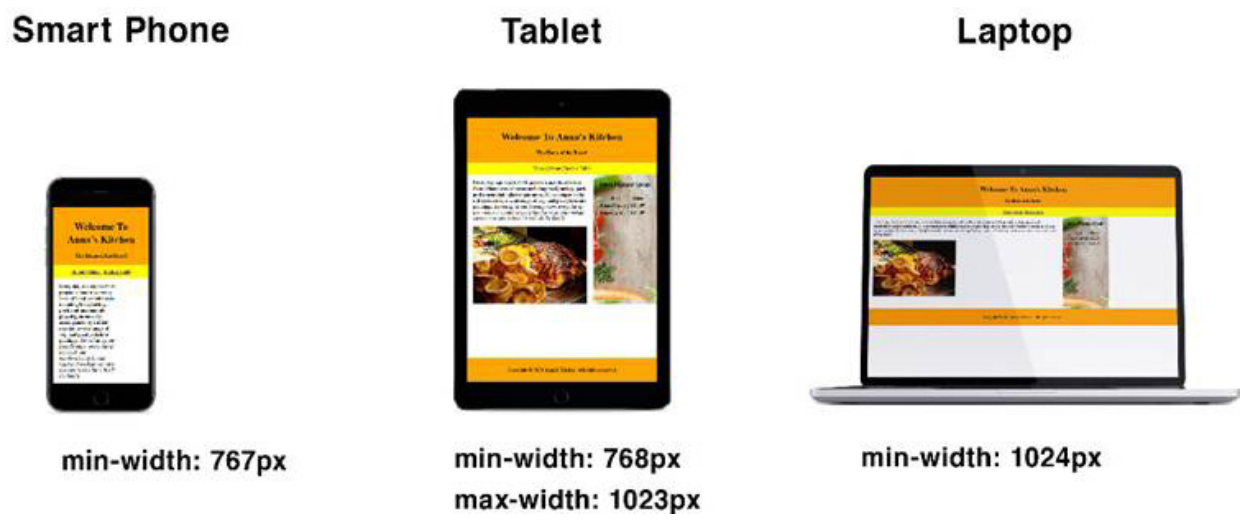


***Figure 9.1**: Responsive site on various screen sizes*

Responsive layouts

With the increasing use of smartphones, tablets, laptops, and PCs with different screen sizes, web pages utilize a responsive layout. This means web pages adapt to different screen sizes.

A responsive grid layout uses media queries in combination with CSS grid or, in some cases, flexbox. Flexbox is best for one-dimensional layouts where you need to align and distribute elements along a single axis (e.g., navigation bars). CSS grid is best for 2D layouts where you need precise control over rows and columns (e.g., complex dashboards or multi-row layouts).

Introduction to media queries

Media queries enable you to create responsive and adaptive layouts that allow your website or application to dynamically adjust its appearance based on screen size, resolution, and orientation. This ensures that content is displayed correctly on a wide range of devices and screen sizes, such as smartphones, tablets, laptops, and desktop computers.

Media queries make it possible to respond to a client's browser with a customized display for certain viewport sizes. The `@media` rule includes a block of CSS properties only if a certain condition is true.

You can set your breakpoints using min-width or max-width properties. *When should you use each one?* Well, if you are designing your layout for small smartphone screens first, then use min-width breakpoints and work your way up. If you have designed the website for a desktop display first, and you want to adapt the layout for smaller screens, then use max-width and work your way down to the smallest screen.

There are different screen sizes, as discussed in the following table:

Device type	Screen resolution	Description
Mobile	375x812	Older iPhone
	360x640	Most Android smartphones (Galaxy S9, Pixel 3)
	414x896	iPhone XS Max, iPhone 11 Pro Max
	375x667	iPhone 7, iPhone 8, iPhone SE (2nd Gen)
Tablet	768x1024	9.7-inch tablets
	800x1280	Samsung Galaxy Tab, 8-inch Android tablets
	834x1112	iPad Pro 10.5-inch
	1024x1366	iPad Pro 12.9-inch
Desktop	1366x768	Common laptop resolution
	1440x900	Larger 15-inch and 17-inch laptops
	1920x1080	Full HD resolution, typical 22-24 inch monitors
Large desktop	3840x2160	4K resolution, ultra-high-definition monitors

***Table 9.1**: Media queries*

In *Figure 9.2*, you can see the website displayed on different-sized screens. Note the content and the sidebar. In the desktop display, the image is aligned to the left of the content container, while the sidebar is aligned to the right. In the iPad, you can see the image is now the full width of the content container with the sidebar aligned to the right. If you look at the website on the phone, you will see that the image is full-width, and the sidebar has been shifted underneath the content container.



***Figure 9.2**: The website on different devices*

Let us take a look at an example. Have a look at **`cssmediaqueries.css`** and **`cssmediaqueries.html`**. To get our website to respond to the different-sized screens, we need to set a media query for each size. If we go back to our example using the CSS grid, you will notice that the content does not respond to different screen sizes; it simply scales to cover the available space, as shown in *Figure 9.3*. It would be better on the mobile device to stack the sidebar underneath the main content.



***Figure 9.3**: Tablet vs. phone view*

To do this, we need to add media queries for each screen size.

The first media query targets devices with a maximum width of 576 pixels. For this size screen, we want the header, navigation bar, content, sidebar, and footer to span the full width of a single column (grid-column: 1 / 2). Here, the **`body`** element has one column (**`grid-template-columns: 1fr`**) and five rows (**`grid-template-rows: auto auto auto auto auto`**):

```
@media only screen and (max-width: 576px) {
    body {
        grid-template-columns: 1fr;                    /*
Single column layout */
        grid-template-rows: auto auto auto auto auto; /*
5 rows */
    }
```

Each content section (header, navigation, main content, aside, and footer) is arranged to span a single column. The main content is set to be in the third

row, and the sidebar (aside) is positioned below it on the fourth row. The footer shifts to the fifth row. This layout ensures that on small screens, all elements are stacked vertically, as shown in the following code:

```
    /* Place all elements in a single column */
    header, nav, main, aside, footer {
        grid-column: 1 / 2; /* Occupy the single column
*/
    }
    /* Adjust rows for specific elements */
    main {
        grid-row: 3 / 4; /* Place main in the third row
*/
    }

    aside {
        grid-row: 4 / 5; /* Place aside sidebar below
main content */
    }

    footer {
        grid-row: 5 / 6; /* Place footer at the bottom
*/
    }
}
```

Figure 9.4 shows how this code will work:

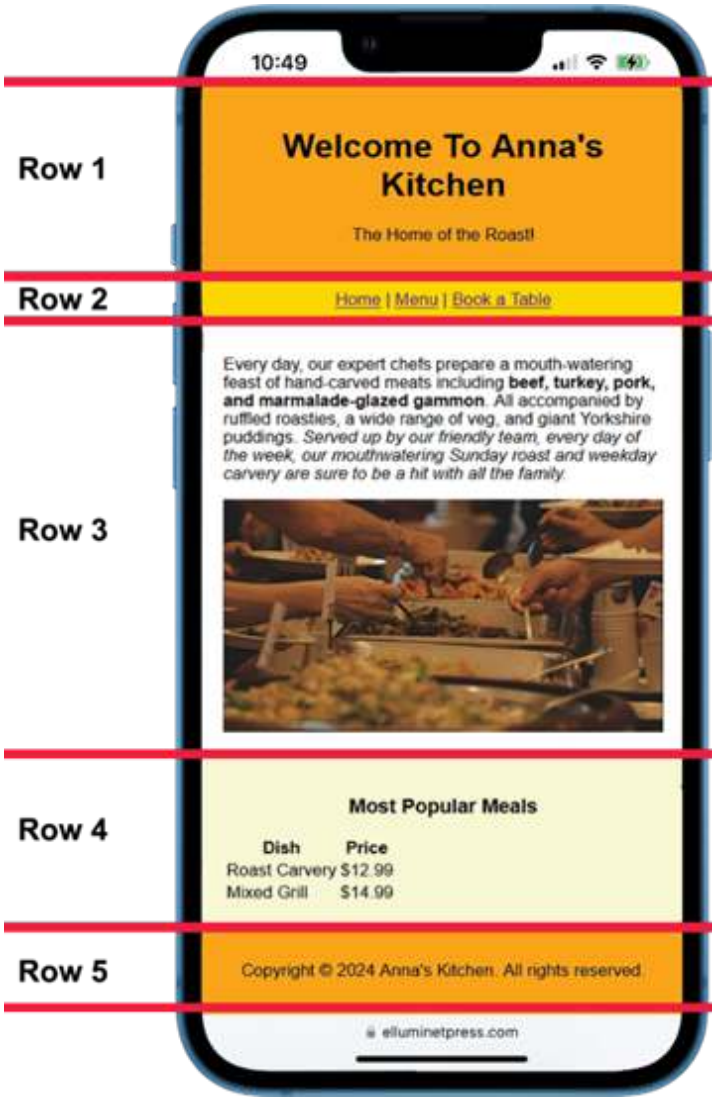


***Figure 9.4**: Row layout for mobile site view*

The second media query targets devices with screen widths between 577px and 991px, which are commonly used in tablets and some smaller laptops, as shown in *Figure 9.5*:

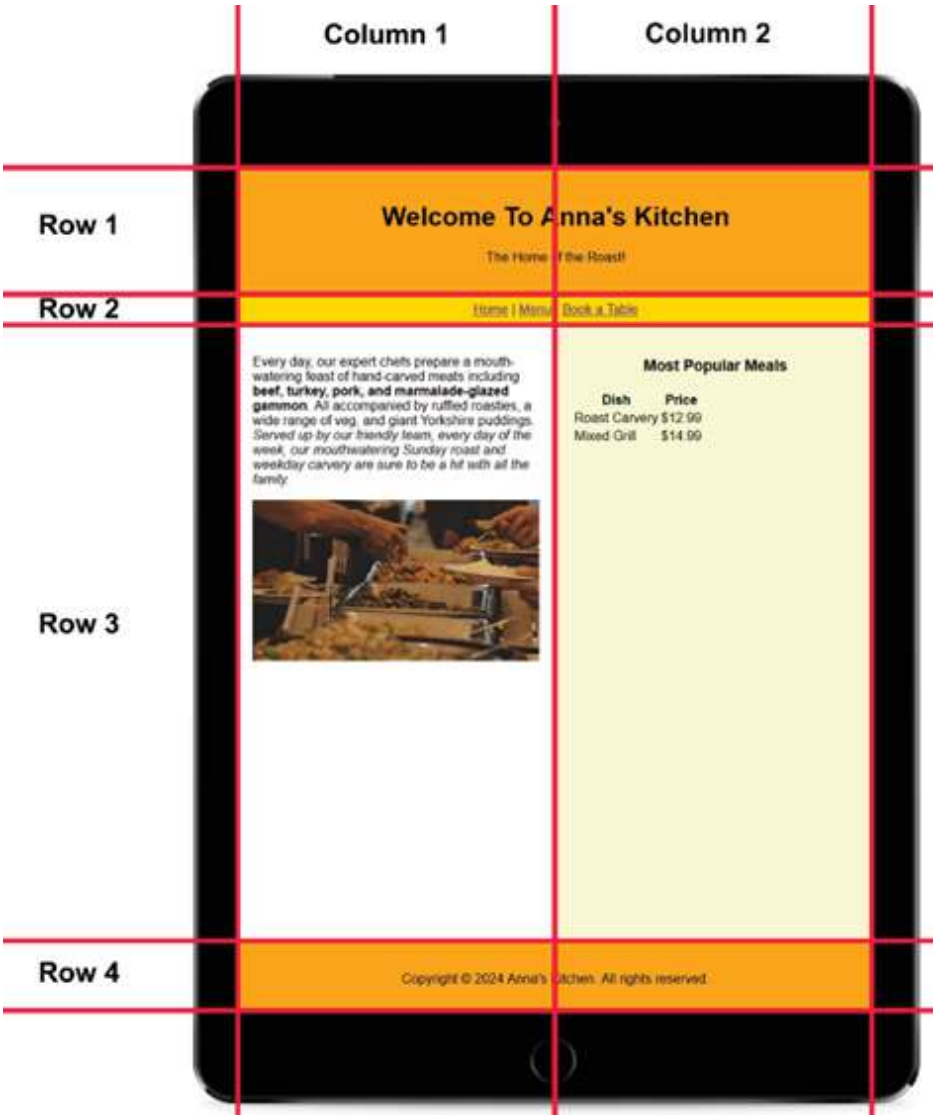


***Figure 9.5**: Row and column layout for tablet view*

This adjusts the layout to a grid with two equally sized columns, as follows:

```
@media only screen and (min-width: 577px) and (max-width: 991px) {
    body {
        grid-template-columns: 1fr 1fr;     /* 2-column layout */
    }
```

The **`header`**, **`nav`**, and **`footer`** elements span across both columns using the **`grid-column: 1 / 3;`** property:

```
    header, nav, footer {
```

```
        grid-column: 1 / 3; /* Span both columns */
    }
```

The **main** content and sidebar (**aside**) elements are positioned side-by-side within the two columns, with the main content in the first column (**grid-column: 1 / 2**), and the sidebar (**aside**) in the second column (**grid-column: 2 / 3**), as:

```
    main {
        grid-column: 1 / 2; /* Main occupies the first
column */
    }
    aside {
        grid-column: 2 / 3; /* Aside sidebar occupies
the second column */
    }
}
```

For any screen sizes over 992 pixels, we can use the default base styles defined at the top of the CSS file, as shown in the following figure:



***Figure 9.6**: Column and row layout for desktop view*

Creating mobile-first designs

Creating mobile-first designs is a design strategy that focuses on building websites specifically for mobile devices, then adapting them to larger screens, such as tablets and desktops. This ensures that websites and applications are fast, streamlined, and efficient on mobile devices, where download speeds and storage space are often limited.

Let us take a look at a simple example, as follows. Have a look at **`mobilefirst.css`** and **`mobilefirst.html`**:

1. Start by defining styles for the smallest viewports (e.g., mobile phones) as the default CSS rules. Refer to *Figure 9.7:*

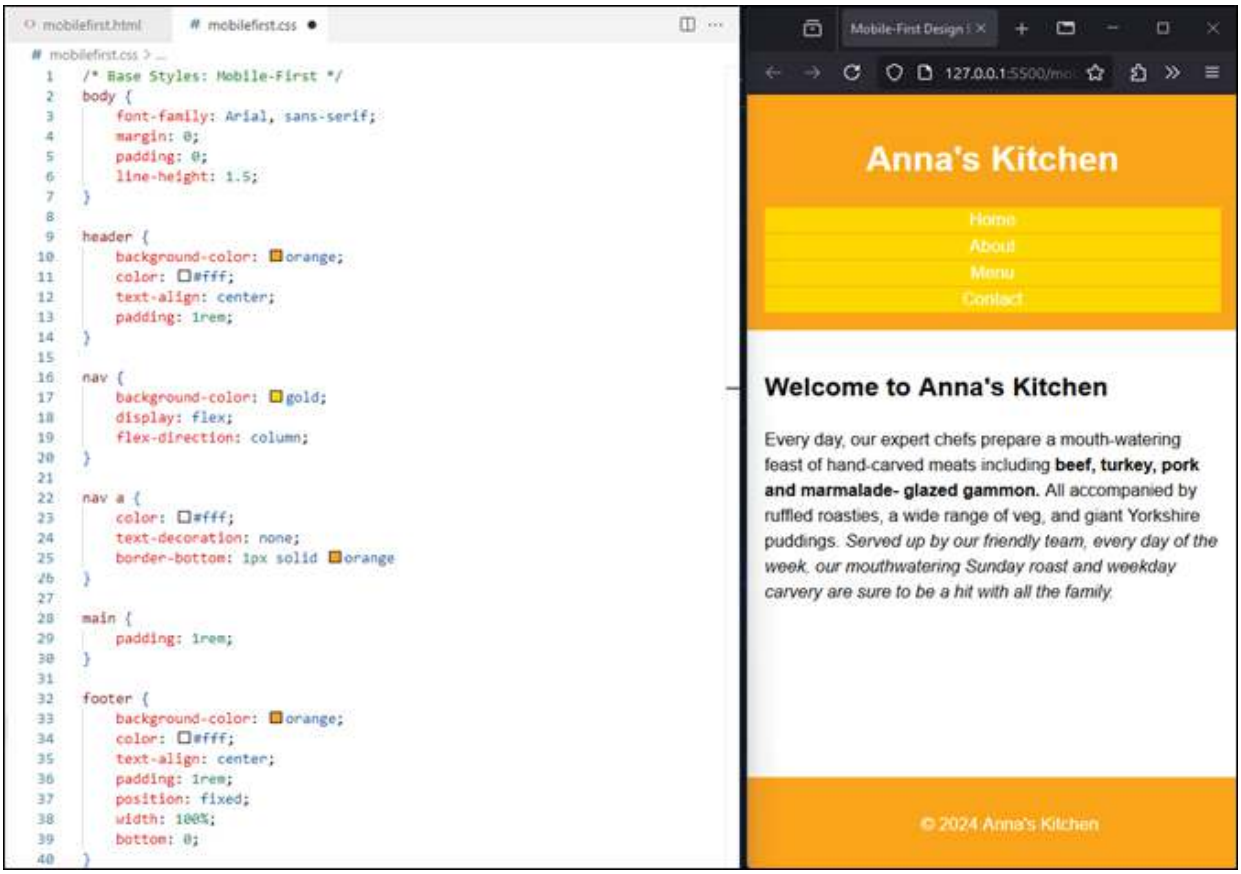


***Figure 9.7**: Mobile first styles*

2. Then use media queries to progressively scale and adapt the layout for a larger screen, such as a tablet. Refer to *Figure 9.8*:

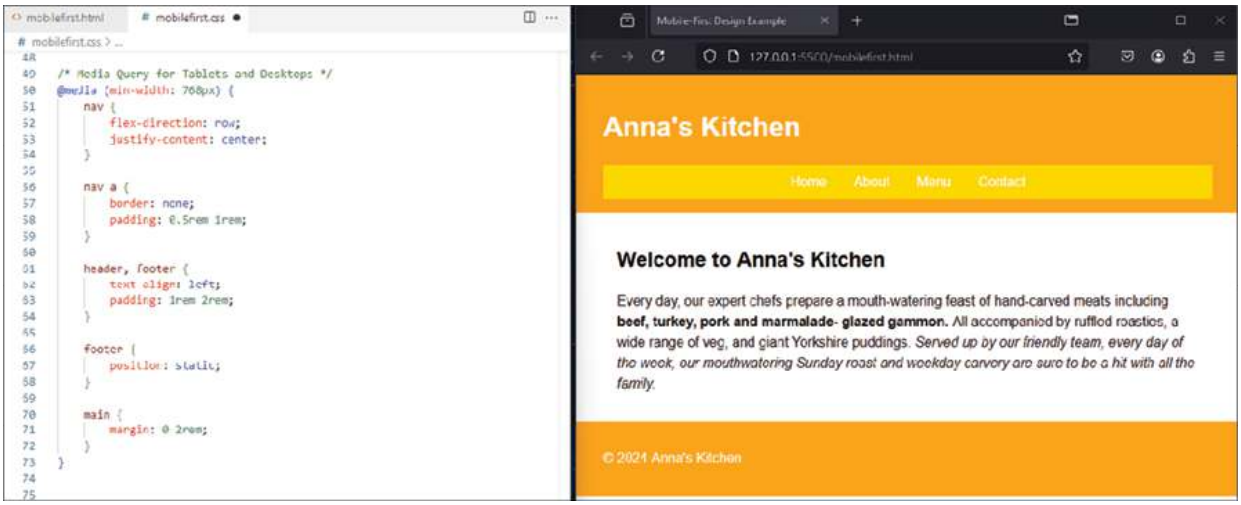


***Figure 9.8**: Tablet styles*

3. Then again, for a desktop screen, as:

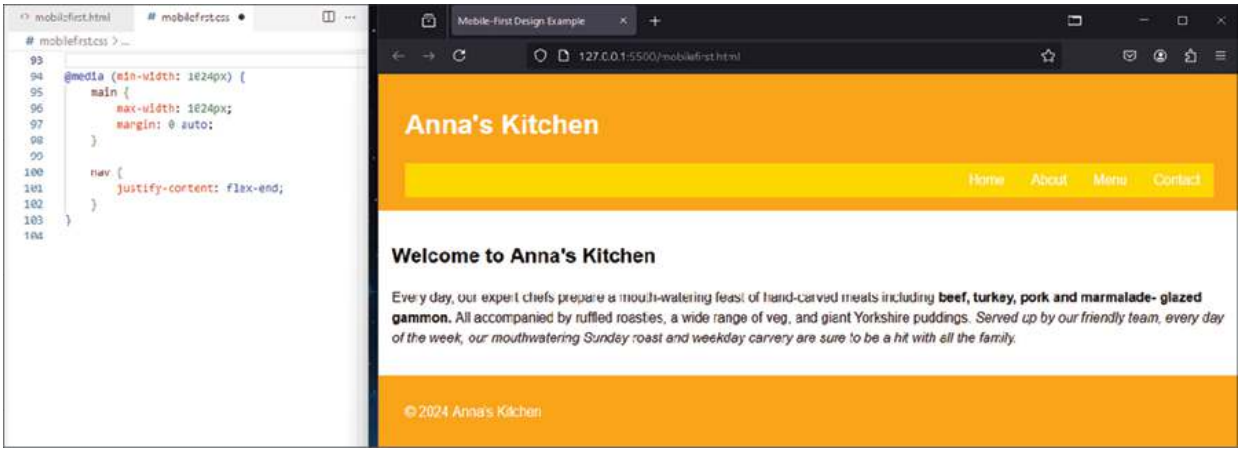


***Figure 9.9**: Desktop styles*

Try it out in the web browser. When the browser width is below 768 pixels, the mobile view is seen, as shown in *Figure 9.7*. When you increase the width above 768 pixels, you see the tablet layout, as shown in *Figure 9.8*. If you increase the width to over 1024 pixels, you see the desktop layout, as shown in *Figure 9.9*. This is why it is called **mobile first**, because we have built the site for a mobile screen first, then scaled it up for each larger screen size.

Testing responsive websites

Testing responsive websites involves ensuring that your web content adapts and performs well across a range of devices and screen sizes. You can do this using any browser, such as *Edge*, *Chrome*, *Firefox*, or *Safari*.

Open the website in a browser, then press *F12* to open the developer tools. Click on the toggle device toolbar button, which looks like a phone and tablet icon, as shown in *Figure 9.10*:

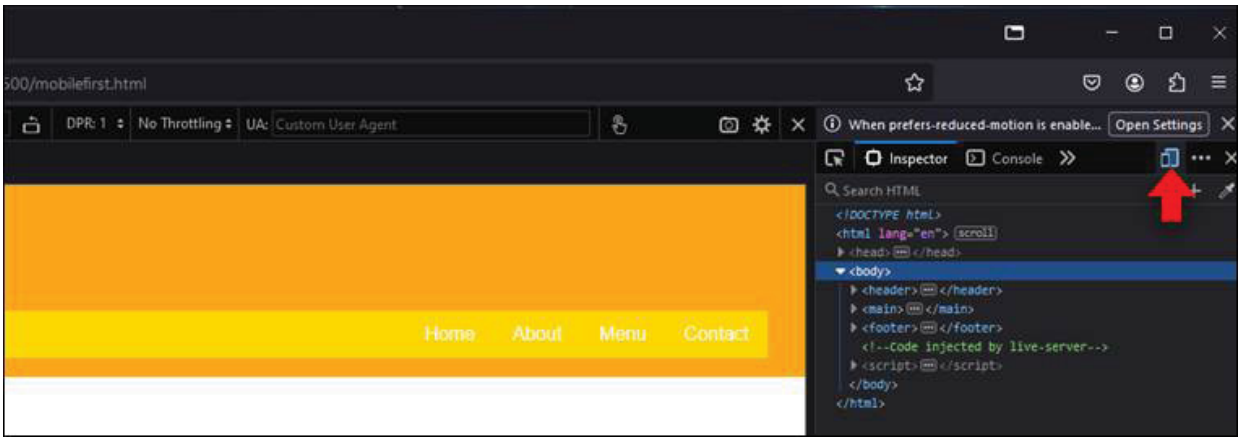


***Figure 9.10**: Opening responsive testing in browser developer tools*

From the bar across the top, choose from the pre-defined device options (like iPhone, iPad, Galaxy, etc.) or manually adjust the width and height to test custom dimensions, as shown in *Figure 9.11*:

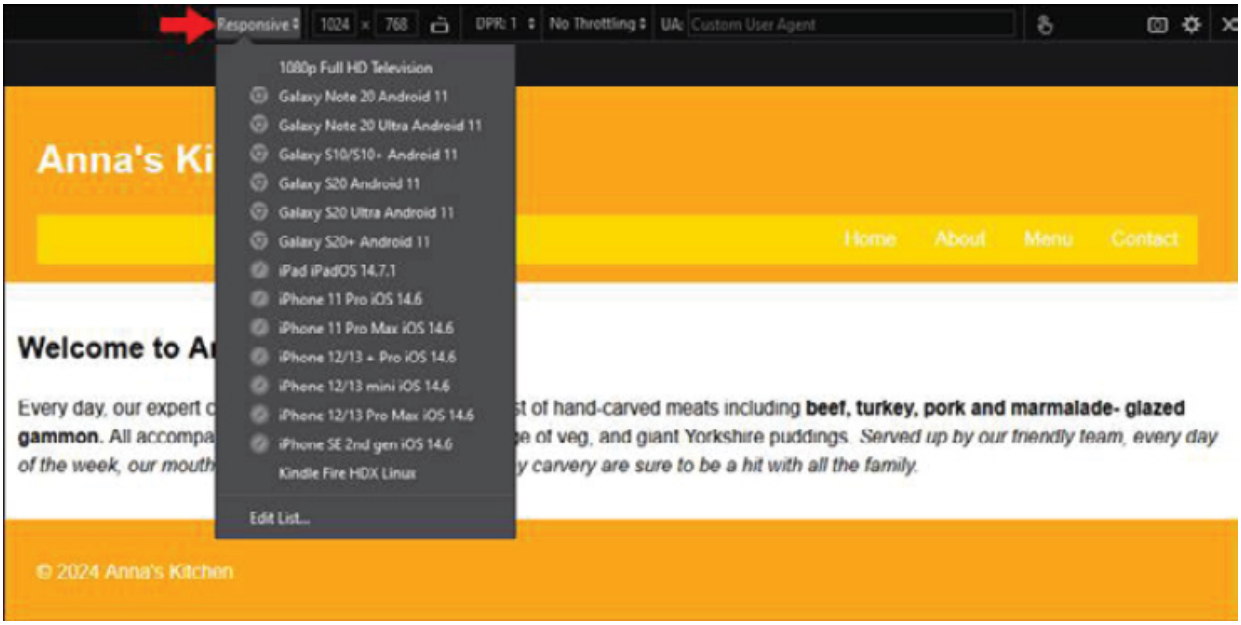


***Figure 9.11**: Changing to different devices and screen sizes*

Check how your website responds to different sizes. For mobile (up to 767px), this layout has a vertical navigation menu with centered text, and a fixed footer at the bottom. So, you can select the following devices from the drop-down menu:

- iPhone SE (375 x 667)
- iPhone 12/13 (390 x 844)

- Galaxy Note 20 (412 x 915)

In *Figure 9.12*, we can see the website as viewed on an iPhone 12/13:

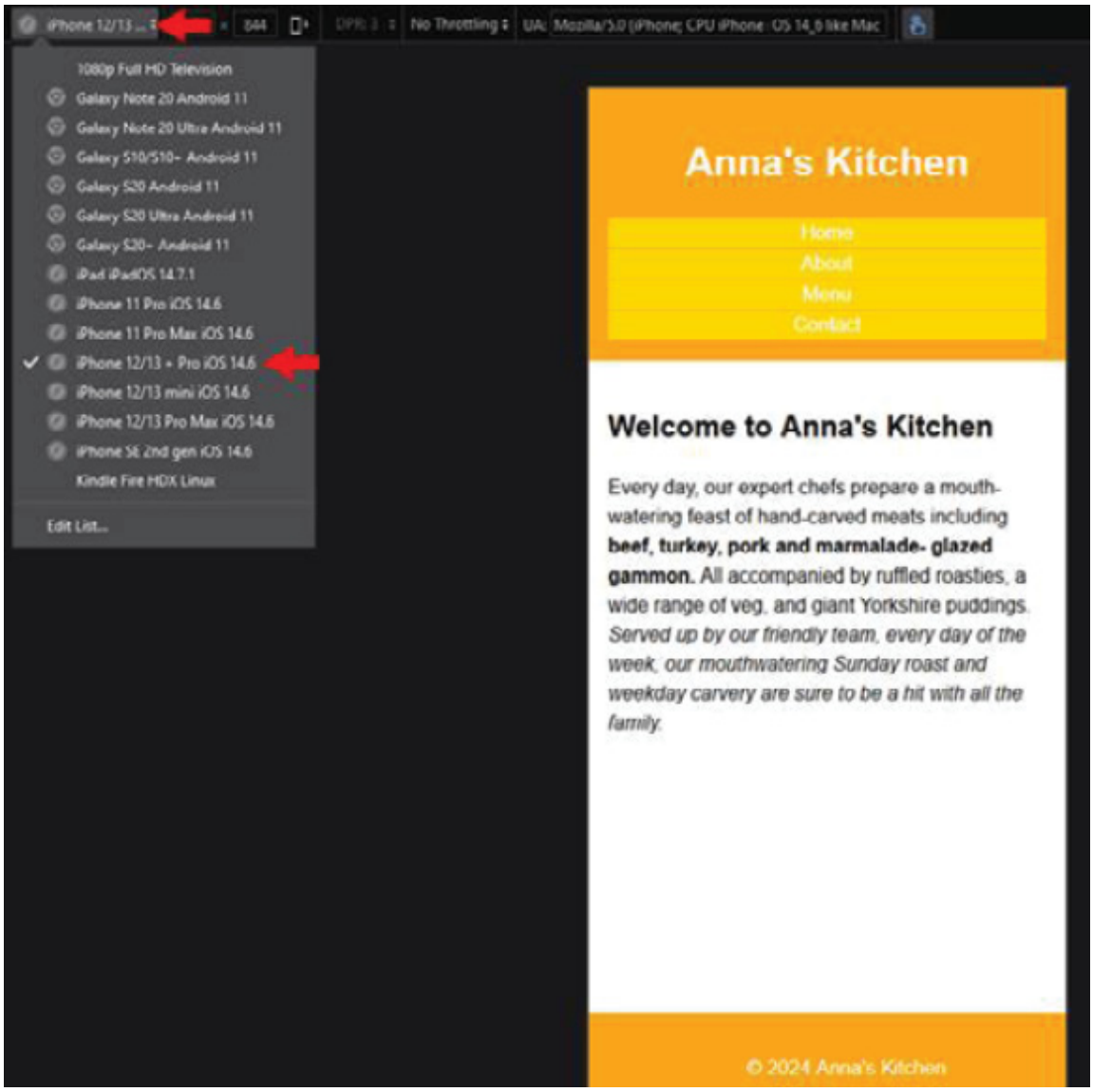


***Figure 9.12**: Selecting a different screen size and device*

For tablets (768px to 1023px), the navigation should change to a horizontal layout and be centered in the navigation bar. So, the following devices can be selected:

- iPad (768 x 1024)
- Galaxy S10/S10+ (360 x 760)

For desktop (1024px and above), the navigation bar should be aligned to the

right. The following can be selected:

- 1080p Full HD Television (1920 x 1080)
- Responsive and manually set the width to 1024px or higher using the height and width fields on the bar along the top of the screen.

Try various devices to make sure the site appears correctly on each one.

Calculating with CSS using calc(), min(), max()

There are various built-in functions that you can use to perform calculations, adapt layouts, and optimize designs dynamically.

Some of them are discussed in the following sections.

Calc

The function **`calc()`** performs calculations to determine CSS property values. In the following example, the element's width is set to the full width of its parent container (**`100%`**) minus 40 pixels:

```
width: calc(100% - 40px)
```

The **`calc()`** function allows you to combine different units, such as percentages and pixels, in a single calculation. This is useful when creating layouts that are flexible, as shown in the following figure:

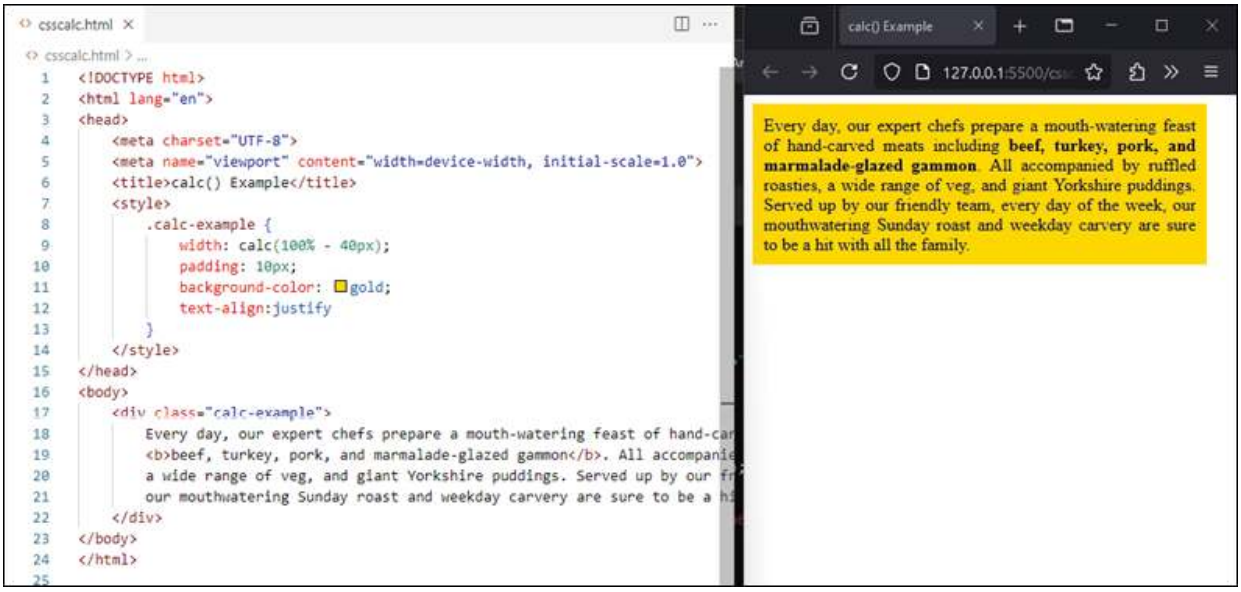


***Figure 9.13**: Using the calc function*

Min

The **`min()`** function is often used to ensure that a CSS property does not exceed a certain value. In the following example, the **`min()`** function takes two arguments, **`400px`** and **`80%`**. It will choose the smaller of the two values in the parentheses.

```
width: min(400px, 80%)
```

This means that if **`80%`** of the container or viewport width is smaller than **`400px`**, the element's width will be set to **`80%`** of the container or viewport. However, if **`80%`** of the container or viewport is larger than **`400px`**, it will restrict the element's width to a maximum of **`400px`**, as shown in *Figure 9.14*:

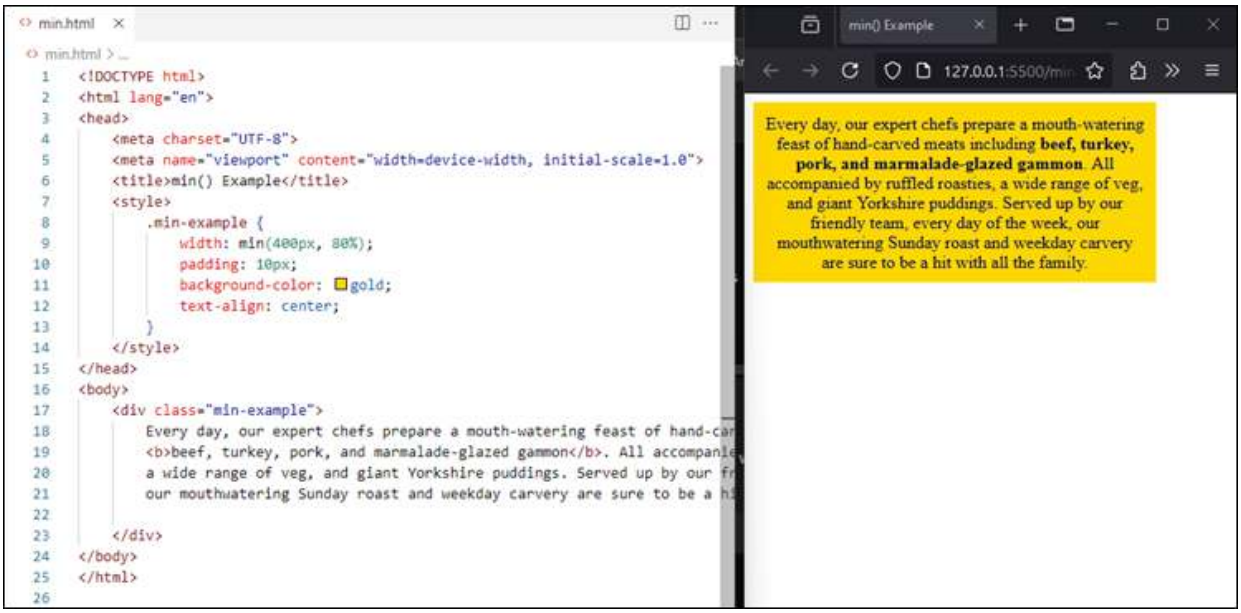


***Figure 9.14**: Using the min function*

Max

The **`max()`** function is used to ensure that a CSS property does not fall below a certain value. In the following example, we want to set the width of the element to the maximum value between two options: **`60%`** of its width or **`200px`**:

```
width: max(60%, 200px)
```

The **`max()`** function selects the greater value between the specified options

in the parentheses. In this case, if **`60%`** of the available container or the viewport width is smaller than **`200px`**, the element's width will be set to **`200px`**. If **`60%`** of the available container or the viewport width is larger than **`200px`**, the element's width will be **`60%`** of the container or the viewport width. This ensures that the element has a minimum width of **`200px`** but can expand based on the available space, as shown in *Figure 9.15*:

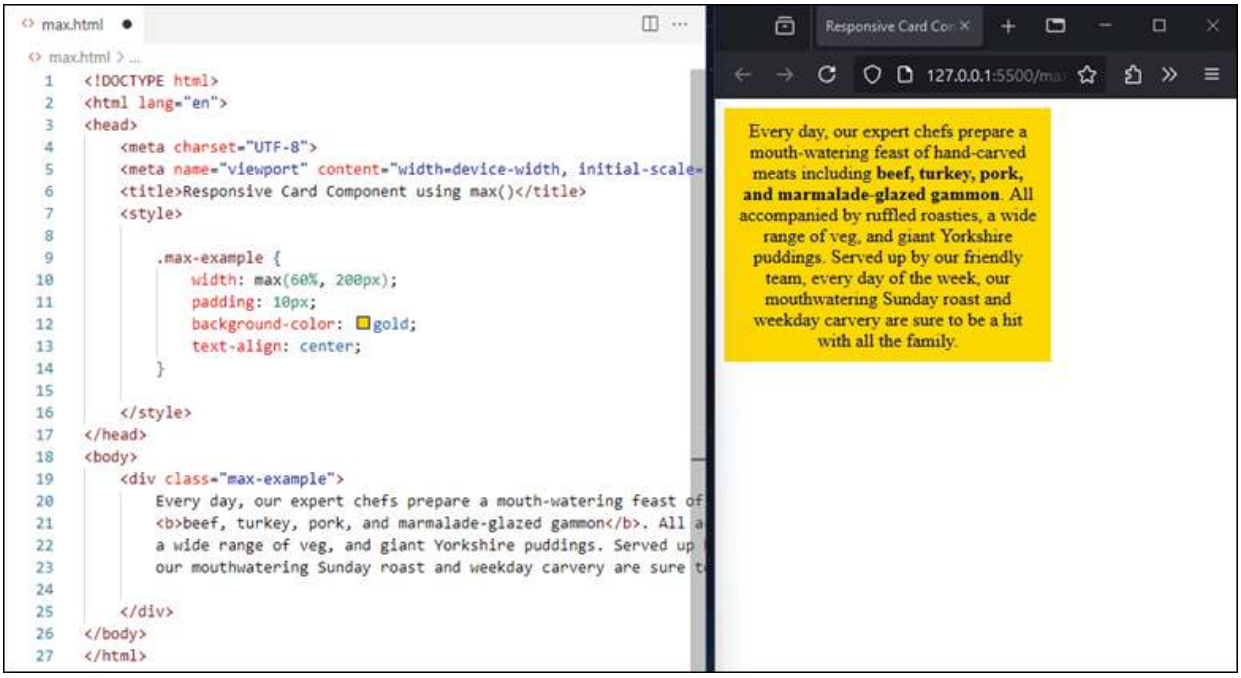


***Figure 9.15**: Using the max function*

Conclusion

Building flexible, adaptable websites is not just about scaling layouts; it is about delivering a good user experience that adapts to each screen type, from mobile phones to large desktop displays. By leveraging techniques such as CSS grid, flexbox, and media queries, we can fine-tune web layouts to different screen sizes.

The mobile-first approach ensures that design begins with the essentials, optimizing content for smaller screens and then expanding its utility and visual appeal for larger devices. This emphasis on responsiveness enhances both usability and performance, key pillars of modern, user-centric web development.

Finally, functions, such as calc(), min(), and max(), allow control over every element's size and placement.

In the next chapter, you will extend your CSS skills by learning how to customize the appearance of HTML forms. You will explore how to style form controls and design accessible, visually consistent buttons that align with your site's overall aesthetic. These styling techniques will enable you to transform basic forms into cohesive, user-friendly components that integrate seamlessly into modern, responsive designs.

Points to remember

- Responsive design enables web content to adapt seamlessly across various screen sizes and devices.
- Flexbox is ideal for one-dimensional layouts (e.g., navigation bars).
- CSS grid is best for complex two-dimensional layouts (e.g., dashboards).
- Media queries allow CSS to respond to different screen sizes, resolutions, and orientations, using min-width and max-width breakpoints.
- Mobile-first design starts with small screens as the default, scaling up for larger screens to ensure efficiency and simplicity.
- Breakpoints set specific layout changes at defined screen widths, helping content display optimally across devices.
- The calc() function combines units (e.g., %, px) for flexible layouts, adjusting width or spacing dynamically based on container size.
- The min() function ensures a maximum limit for element dimensions, while max() ensures a minimum limit, keeping layout control consistent.
- Testing in developer tools enables you to preview website responsiveness by simulating different device screen sizes and resolutions.
- Device resolutions like 375x667 (mobile), 768x1024 (tablet), and 1920x1080 (desktop) typically require tailored layout adjustments.
- A responsive website should look good and function properly on screens ranging from mobile phones to large desktop monitors.

Exercises

1. What is flexbox? Show an example.
2. What is CSS grid? Show an example.
3. Write a media query that targets devices with a screen width of 768 pixels or less. Use this media query to set the font size of paragraphs to 14px and the padding around images to 5px.
4. Outline the steps required to create a mobile-first layout for a simple web page with a header, navigation, main content, sidebar, and footer. Specify which CSS properties would be set in the base CSS.
5. What is the purpose of using min-width and max-width properties in media queries, and how does this affect the approach to responsive design?
6. Write an example CSS rule using the calc() function to set an element's width to 80% of the container's width minus 20px. Explain when you might use calc() in a responsive layout.
7. Create a media query for devices with a minimum width of 1024 pixels. Within this media query, style the main content to occupy two-thirds of the page width and a sidebar to occupy one-third of the page width.
8. Explain the difference between the min() and max() CSS functions, and give an example of how each might be used in a responsive design.
9. List three device types, their typical screen resolutions, and explain how a website layout might adjust differently on each.
10. Describe the process of testing a responsive website using the developer tools in a browser. What settings or tools would you use to simulate different screen sizes?

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com



CHAPTER 10

Styling Forms and Buttons with CSS

Introduction

As discussed in *Chapter 6, HTML Forms: Collecting User Input*, where we built our contact form, this chapter explores CSS techniques for enhancing and styling the appearance of web forms and buttons. It begins with customizing form inputs and text areas for a cohesive design. Next, you will learn to design visually appealing buttons and add interactive states like hover and active. The chapter also covers styling radio buttons and checkboxes to match the overall design theme. Additionally, the use of pseudo-classes and pseudo-elements to create dynamic and engaging form elements will be discussed.

Structure

The chapter covers the following topics:

- Customizing an HTML form
- Styling buttons

Objectives

By the end of this chapter, you will understand how to style and customize an HTML form, as well as buttons and text fields.

Customizing an HTML form

You can use CSS to customize and style form input fields and text areas. Have a look at **`book.html`** and **`book.css`**. You can add padding, borders, and background colors to make your form look more polished. We can do this by targeting the HTML tags and applying various CSS styles. For example, we can change the font family, background color, and margins of the booking page. We can do this using the body selector, which targets the body tag in the HTML, as:

```
body {
    font-family: Arial, sans-serif;
    background-color:orange;
    margin: 20px;
}
```

We can also style the **`fieldset`** element, which is the container used to group related form elements together, as:

```
fieldset {
    border: 0px solid ;
    padding: 15px;
    margin-bottom: 20px;
    background-color: gold;
    border-radius: 4px;
}
```

We can style the **`legend`** element, which is the caption or title that appears inside the **`fieldset`** element, as:

```
legend {
    background-color: gold;
    padding: 5px 10px;
```

```
    border-radius: 5px;
    font-weight: bold;
}
```

Once you have added the styles, you will end up with something like this, as shown in *Figure 10.1*:

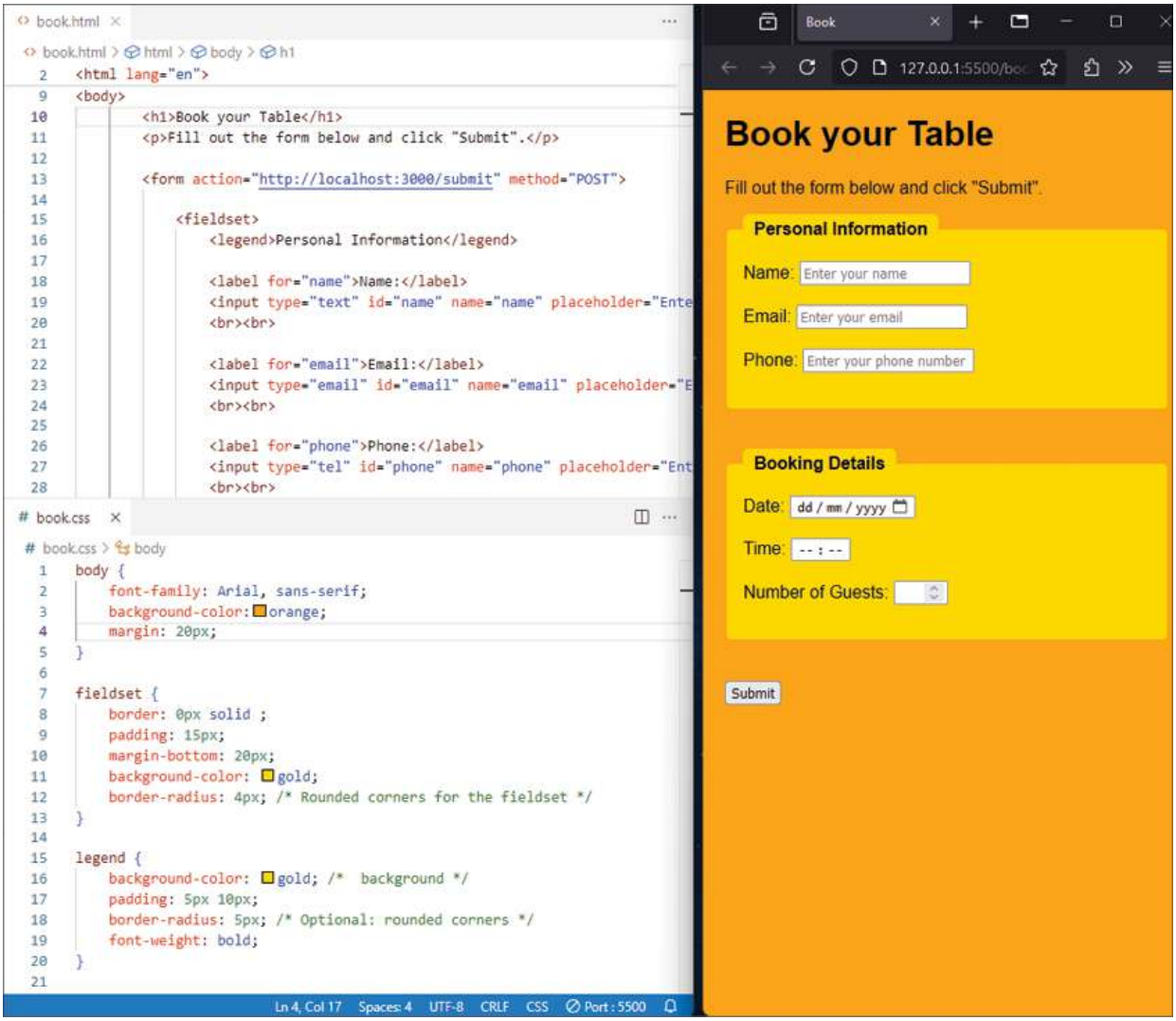


***Figure 10.1**: HTML form*

We can also style the **`input`** fields. In the following example, we are applying the styles to all the fields in the form:

```
input {
    width: 100%;
    padding: 4px;
```

```
    border: 1px solid darkorange;
    border-radius: 5px;
    box-sizing: border-box;
}
```

You can also target any other **`input`** type specifically:

`input[type="input type"]`

Just replace the **`input type`** with an input type from the following table:

Input type	Description
`text`	Standard single-line text input field.
`password`	Obscures the entered characters for password fields.
`email`	For entering a valid email address, often validated for proper format.
`tel`	For entering a telephone number.
`number`	For numeric values, with optional constraints like min, max, and step.
`checkbox`	For selecting multiple options independently.
`radio`	For selecting a single option from a set of mutually exclusive choices.
`button`	For a generic button that can trigger custom scripts or actions.
`submit`	For a button that submits the form data.
`reset`	For a button that resets all form fields to their initial values.

***Table 10.1**: Input fields*

For example, the **`date`** input type can be changed using the following code:

```
input[type="date"] {
    width: 100px;
    background-color: lightgoldenrodyellow;
}
```

We can see the size of the date field has changed, as well as the background color, as shown in *Figure 10.2*:

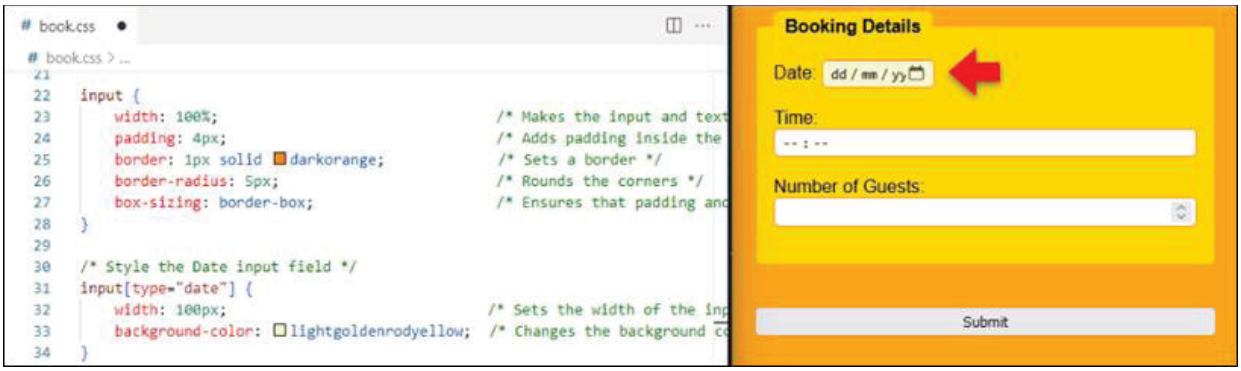


***Figure 10.2**: Data input type*

Styling buttons

Buttons allow users to take action, such as submitting a form or navigating through the site. To style the button, we add the field type to the **`input`** selector; in this case, it is a **`submit`** button. This selector targets the **`submit`** button within the form, as follows:

```
input[type="submit"] {
    background-color: gold;
    color: black;
    border: none;
    padding: 12px;
    border-radius: 5px;
    cursor: pointer;
    font-size: 16px;
    width: 100%;
    text-align: center;
}
```

Here, in the following figure, we have styled the **`submit`** button:

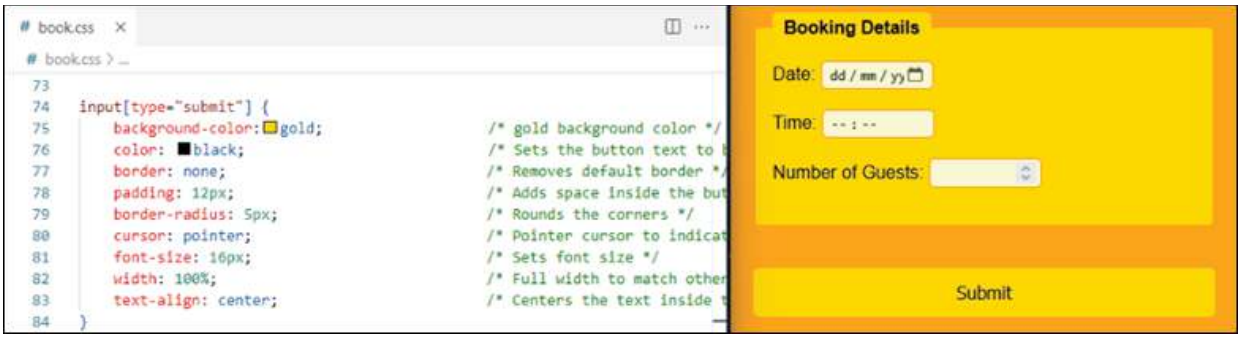


***Figure 10.3**: Submit button*

If we want to add a rollover effect so the button changes when the user hovers the mouse pointer over the button, we can add a pseudo-class.

For example, **`:hover`** changes the style of an element when a mouse pointer is hovering over it, **`:focus`** applies styles when an element gains focus, and **`:active`** applies styles while an element is clicked or activated. This provides useful visual feedback during user interactions. In the following figure, we can see the hover state of the submit button as the mouse pointer hovers over the button:



***Figure 10.4**: Hover state of the submit button*

Going back to our booking form, we can add hover and active states to provide visual feedback when users interact with the form. We can add a rollover effect when the user moves the mouse pointer over the button, or change the color of the button when it has been clicked.

We can do this by styling the different button states, as follows:

- A button can be in a normal state, which is the default state of a button when it appears on the page, and no interactions have occurred.
- The hover state is triggered when the user moves the mouse pointer over the button.
- The active state is triggered when the button is clicked or pressed.
- The focus state appears when the button gains focus, either by clicking on it or navigating to it using the keyboard.

To apply a style to a specific state, we use pseudo-classes. These are placed after the input type, separated by a colon. For example, the hover state is applied as follows:

```
input[type="submit"]:hover
```

So, to change the style of the hover state, we change the background color and the button text color, as:

```
input[type="submit"]:hover {
    background-color: darkorange
    color: white;
}
```

In the following figure, we have added a hover state to the submit button. Now, when you hover the mouse over the button, it changes to dark orange with white text:

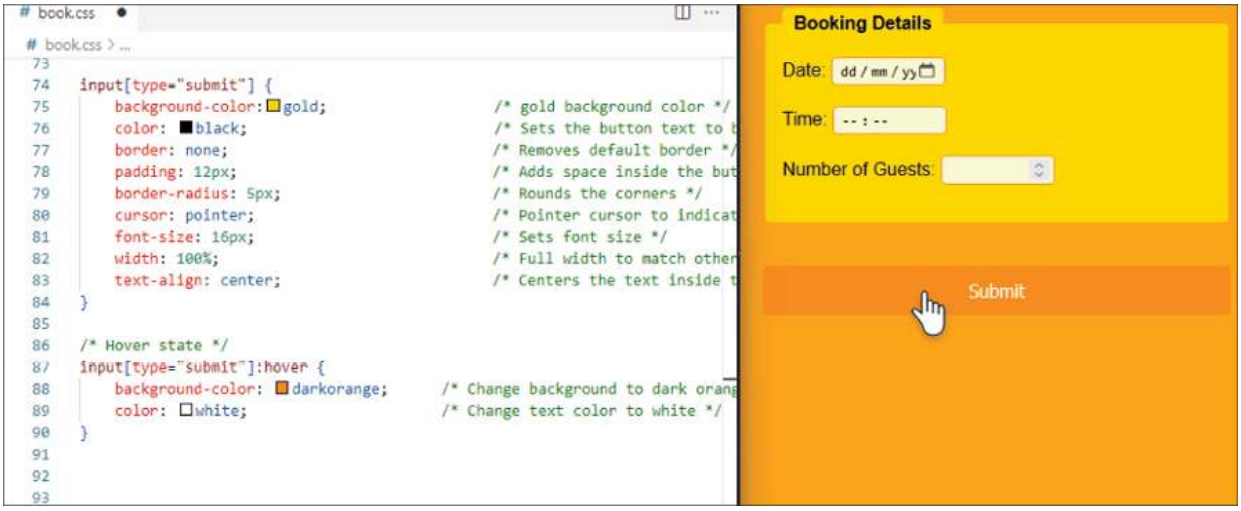


***Figure 10.5**: Submit button changes colour*

Conclusion

The visual styling of interactive forms plays a crucial role in enhancing both usability and accessibility. By applying CSS to customize form elements—such as input fields, labels, borders, fonts, and button states—developers can improve the overall user experience, making forms more intuitive and visually consistent with the rest of the website. Thoughtful design not only reinforces the structure of the form but also guides users through the data entry process, reducing errors and improving submission accuracy.

Effective form customization helps establish visual hierarchy, supports responsive design, and provides essential feedback through hover, focus, and active states. These refinements ensure that forms are not only functional but also accessible and aesthetically aligned with the site's design language. As developers gain confidence with CSS, further experimentation with form styling enables the creation of polished, professional user interfaces that meet both user expectations and design standards.

In the next chapter, the focus shifts to JavaScript, the scripting language that brings interactivity and logic to web pages. Readers will be introduced to JavaScript basics, including syntax, variables, constants, data types (such as strings, numbers, and Booleans), and output techniques. The chapter will also cover control structures, integration with HTML, and introduce basic error handling using try...catch, laying the groundwork for dynamic, interactive web applications.

Points to remember

- **Use CSS for enhanced visual appeal**: Styling forms with CSS makes them look polished and professional.
- **Add padding and borders**: Apply padding and borders to form fields for a cleaner, more organized look.
- **Group related elements with fieldset**: The fieldset and legend elements help group related form fields for better accessibility.
- **Apply styles with box-sizing: border-box;**: Ensures that padding and borders are included in the width/height of elements.

- **Customize button states with pseudo-classes**: Use :hover, :focus, and :active for responsive interactions on buttons.
- **Target specific input types**: Style particular input types, such as input[type="text"] or input[type="date"], for consistency.
- **Use border radius for rounded corners**: Adds a modern look to input fields and buttons, making them visually appealing.
- **Implement rollover effects on buttons**: Provide user feedback by changing button colors on hover or click states.
- **Ensure form responsiveness**: Use CSS to make form elements adapt to different screen sizes and devices.
- **Test interactivity and accessibility**: Ensure all form elements, especially interactive ones like buttons, are accessible and easy to use.

Exercises

1. How can using CSS enhance the user experience of a form on a web page?
2. Which CSS properties would you use to add spacing within input fields, and why is this important?
3. Explain how to style the fieldset element in a form. Why is it useful to group related form elements together?
4. How does the box-sizing: border-box; property affect the width of input fields in a form?
5. What is the purpose of using pseudo-classes like : hover, : focus, and: active on buttons, and how do they improve user interaction?
6. Describe the steps needed to apply a custom background color to a submit button in a form.
7. How would you target a specific input type, such as input[type="date"], in CSS to apply unique styling?
8. Why might it be beneficial to use a reset button (input[type="reset"]) within a form, and how could you style it to stand out?
9. What are the advantages of adding rounded corners (using border-radius) to form elements like buttons and input fields?
10. How can CSS be used to ensure that form elements are responsive and

adapt to different screen sizes?

CHAPTER 11

Introduction to JavaScript

Introduction

This chapter provides a comprehensive introduction to JavaScript, covering fundamental concepts essential for beginners. It starts with JavaScript basics, including syntax and data types, then progresses to variables and constants. You will explore operations with numbers, strings, and booleans, and learn to manage control flow using if/else and switch statements. The chapter concludes with a look at functions and their parameters, equipping you with the foundational skills to start coding in JavaScript.

Structure

The chapter covers the following topics:

- JavaScript basics
- Basic syntax
- Reserved words
- Variables and constants
- Working with numbers, strings, and Booleans
- Output
- Integration with HTML

- Flow control
- Basic error handling with try...catch

Objectives

By the end of this chapter, you will be able to understand the role of JavaScript in web development and its execution environment, including its use in both client-side and server-side applications. You will be able to write JavaScript programs using correct syntax and variable declarations while ensuring proper structuring of code. Additionally, you will learn how to use control structures, such as conditional statements and loops, to manage the flow of execution in a program.

You will be able to define and call functions, including those with parameters and return values, to create reusable and efficient code. Working with arrays will also be covered, allowing you to perform common operations such as adding, removing, and extracting elements. Furthermore, you will learn how to create and manipulate objects, define properties and methods, and use classes with constructors to create structured and scalable code.

The chapter will also explain hoisting and how it affects JavaScript code execution, ensuring a deeper understanding of how the language processes variable and function declarations. You will also learn how to handle errors effectively using try...catch blocks to prevent runtime failures. Additionally, logical operators will be explored, enabling you to evaluate conditions efficiently within programs. Lastly, you will be able to integrate JavaScript with HTML and display output using various methods to create interactive and dynamic web applications.

JavaScript basics

JavaScript is a high-level, dynamic, multi-paradigm programming language primarily used for creating interactive and dynamic content on web pages. It supports object-oriented, functional, or imperative styles of programming, and code is typically executed in a web browser on the client. With the

advent of Node.js, JavaScript can also be used for server-side programming.

Node.js is a JavaScript runtime environment that allows JavaScript to be executed on a server. It is primarily used for building server-side applications, including web servers, RESTful APIs, real-time applications (like chat apps), and other backend services. Node.js operates with a non-blocking, asynchronous model, meaning it can handle multiple operations simultaneously without waiting for one to finish before starting another, making it especially efficient for I/O-intensive tasks.

The JavaScript ecosystem contains numerous frameworks and libraries, such as React, Angular, and Vue.js, that are used to build complex websites and applications. These frameworks provide reusable components, pre-defined functions, standardized conventions, and best practices so developers can focus on implementing core features and program logic rather than coding everything from scratch.

React is a JavaScript library developed by Facebook, primarily used for building user interfaces. It follows a component-based architecture, which means developers can break down the UI into reusable and independent pieces. React also utilizes a Virtual DOM, which optimizes rendering by updating only the necessary parts of the actual DOM, resulting in better performance. Since React is primarily a library, it provides flexibility, allowing developers to integrate it with other tools and frameworks to meet specific project needs.

Angular, developed by *Google*, is a full-fledged framework that comes with a comprehensive set of built-in features, including two-way data binding, dependency injection, and a powerful **command line interface** (**CLI**) for automating tasks. Unlike React, Angular provides a complete solution for building large-scale applications, offering tools for routing, form validation, state management, and more. It follows the **Model-View-Controller** (**MVC**) architecture and uses TypeScript, making it a structured and maintainable choice for enterprise applications.

Vue.js is a progressive framework that combines some of the best features of React and Angular while maintaining a simpler learning curve. It provides reactive data binding, meaning the UI automatically updates when the underlying data changes. Like React, Vue is also component-based, allowing developers to create modular and reusable code. However, Vue is

more lightweight than Angular and easier to integrate into existing projects, making it a popular choice for developers who want flexibility without the overhead of a full framework.

Key features of JavaScript are as follows:

- JavaScript is interpreted and does not require compilation.
- Runs on both client-side and server-side. While JavaScript traditionally runs in the web browser, environments such as Node.js enable JavaScript to run on the server as well.
- Dynamically typed meaning variables are not bound to specific data types.
- Can be integrated seamlessly with HTML and CSS.

Basic syntax

JavaScript follows a structured syntax similar to other programming languages. It is case-sensitive. This means, for example, the variable **`Greeting`** is not the same as the variable **`greeting`**.

Statements are terminated with a semicolon (**`;`**), as shown in the following code. While using semicolons is not strictly required, it is generally considered a good practice in professional development, particularly in large projects or teams that follow stricter linting rules. If you are writing for a personal project or a team that follows a no-semicolon style (like StandardJS), omitting them is acceptable, as:

```
let firstName = "Alice";
```

Code blocks are enclosed in curly braces (**`{}`**), allowing for structured and organized execution of code, as shown in the following code. This is commonly used when defining functions, classes, if statements, and loops:

```
if (num >= 10) {
    …
}
```

Comments can be written as single-line using a double slash **`//`** or as multi-line comments enclosed within **`/*`** and **`*/`**, as follows:

```
// This is a single line comment
```

```
/* This is a multiple line comment
as it takes up more than one line */
```

Reserved words

Keywords are reserved words that have a predefined meaning and are used to define control structures, declare variables, create functions, and perform other essential operations. Since these words are part of the language syntax, they cannot be used as variable names, function names, or identifiers. The following is a table of reserved keywords in **ECMAScript 6 (ES6)** along with some examples of use.

First are control statements. These control the flow of program execution, such as conditional branching and multi-way decision structures.

Category	Keyword	Description	Example
Control statements	`if`	Executes a block of code if a condition is true.	`if (x > 10) { message = "x greater than 10"; }`
	`else`	Executes a block of code if the if condition is false.	`if (x > 10) { message = "x greater than 10"; } else { message = "x is 10 or less"; }`
	`switch`	Executes different blocks of code based on different cases.	`switch (day) { case "Monday": task = "Start work"; break; default: task = "Rest day"; }`
	`case`	Defines a case in a switch statement.	`case "Monday": task = "Start work"; break;`
	`default`	Defines the default case in a switch statement.	`default: task = "Rest day";`

***Table 11.1**: Control statements*

The following table shows keywords associated with loop constructs in JavaScript, enabling repeated execution of code blocks based on specified conditions or iterations:

Category	Keyword	Description	Example
Loops	`for`	Creates a loop that runs a specified number of times.	`for (let i = 0; i < 5; i++) { count += i; }`
	`while`	Creates a loop that runs while a condition is true.	`while (x < 10) { x++; }`
	`do`	Executes a loop at least once before checking the condition.	`do { x++; } while (x < 10);`
	`break`	Exits the current loop or switch statement.	`for (let i = 0; i < 10; i++) { if (i === 5) break; }`
	`continue`	Skips the current loop iteration and moves to the next one.	`for (let i = 0; i < 10; i++) { if (i === 5) continue; sum += i; }`

***Table 11.2**: Keywords*

The keywords shown in the following table are used to define and manage functions, return values, and implement generator functions that can be paused and resumed:

Category	Keyword	Description	Example
Functions	`function`	Declares a function.	`function greet(name) { return "Hello " + name; }`
	`return`	Exits from a function and returns a value.	`function sum(a, b) { return a + b; }`
	`yield`	Pauses and resumes a generator function.	`function* generator() { yield "Hello"; yield "World"; }`

***Table 11.3**: Keywords to define and manage functions*

The following table outlines the reserved words used for declaring variables in JavaScript, including block-scoped declarations and constants introduced in ES6:

Category	Keyword	Description	Example
Variable declarations	`var`	Declares a variable (old syntax).	`var x = 5;`
	`let`	Declares a block-scoped variable.	`let x = 10;`
	`const`	Declares a constant variable.	`const PI = 3.14;`

***Table 11.4**: Reserved words*

The following table lists keywords used in defining and working with classes, object-oriented inheritance, and object references within class structures:

Category	Keyword	Description	Example
Classes and objects	`class`	Defines a class.	`class Person { constructor(name) { this.name = name; } }`
	`extends`	Inherits from another class.	`class Employee extends Person { constructor(name, role) { super(name); this.role = role; } }`
	`super`	Calls the constructor of the parent class.	`super(name);`
	`this`	Refers to the current object.	`this.name = "John";`

***Table 11.5**: Keywords used in defining and working classes*

The following table introduces keywords related to asynchronous programming in JavaScript, particularly for working with promises and async functions:

Category	Keyword	Description	Example
Promises and async	`async`	Declares an asynchronous function.	`async function fetchData() { return "data"; }`
	`await`	Pauses execution until a promise is resolved.	`let data = await fetchData();`

***Table 11.6**: Keywords related to asynchronous programming*

The following table includes a selection of additional reserved keywords

used for object creation, type checking, exception handling, and other core language operations:

Category	Keyword	Description	Example
Other reserved keywords	`new`	Creates an instance of an object.	`let obj = new Object();`
	`delete`	Removes a property from an object.	`delete obj.property;`
	`typeof`	Returns the data type of a variable.	`let type = typeof 10;`
	`instanceof`	Checks if an object is an instance of a class.	`console.log(obj instanceof Object);`
	`void`	Evaluates an expression and returns undefined.	`void(0);`
	`try`	Defines a block to handle exceptions.	`try { riskyFunction(); } catch (error) { handle(error); }`
	`catch`	Catches and handles an error.	`catch (error) { message = "An error occurred"; }`
	`finally`	Executes code after try and catch, regardless of outcome.	`finally { cleanup(); }`
	`throw`	Throws an error.	`throw new Error("Something went wrong");`

***Table 11.7**: Additional reserved keywords*

Variables and constants

In JavaScript, variables are used to store data that can be manipulated and referenced later in the program.

Each variable must be declared before it is used. Traditionally, this was done using the **`var`** keyword. However, with the introduction of ES6, the **`let`** keyword is now preferred for variable declarations due to its block-scoping feature, as:

```
let firstName = "Alice";
```

Also introduced in ES6, **`const`** is used to declare constants, which means their values cannot be changed after the initial assignment, as:

```
const pi = 3.14
```

Working with numbers, strings, and Booleans

JavaScript supports different data types that allow you to perform various operations. You can use numbers, strings, and Booleans.

Numbers

Numbers are used for mathematical calculations. JavaScript has a single Number type that includes both integer and floating-point values, and is used for all numerical operations, including arithmetic and advanced calculations, as follows:

```
let num1 = 10;      // Integer
let num2 = 3.5;     // Float
```

Basic operations

JavaScript supports standard mathematical operations, which are used to perform arithmetic on numbers. These numbers can be integers or floating-point:

Operation	Description	Example
Increment (++)	Increases a variable's value by 1.	`let x = 5; x++;`
Decrement (--)	Decreases a variable's value by 1.	`let y = 5; y--;`
Multiplication (*)	Multiplies two numbers.	`let prod = firstnum * secondnum;`
Division (/)	Divides one number by another.	`let quo = firstnum / secondnum;`
Modulus (%)	Returns the remainder of a division.	`let remainder = firstnum % secondnum;`

Operation	Description	Example
Addition (+)	Adds two numbers together.	`let sum = firstnum + secondnum;`
Subtraction (-)	Subtracts one number from another.	`let diff = firstnum - secondnum;`

***Table 11.8**: Standard mathematical operations*

Mathematical functions

JavaScript provides built-in mathematical functions through the **`Math`** object and the **`Number`** class. The following functions are used to perform complex calculations, rounding operations, and number validations:

Function	Description	Example
`Math.sqrt(x)`	Returns the square root of x.	`Math.sqrt(25);`
`Math.pow(x, y)`	Returns x raised to the power of y.	`Math.pow(2, 3);`
`Math.round(x)`	Rounds x to the nearest integer.	`Math.round(4.6);`
`Math.floor(x)`	Rounds x down to the nearest whole number.	`Math.floor(4.9);`
`Math.ceil(x)`	Rounds x up to the nearest whole number.	`Math.ceil(4.1);`
`Math.random()`	Generates a random number between 0 and 1.	`Math.random();`
`Number.isInteger(x)`	Checks if x is an integer.	`Number.isInteger(10);`
`Number.isNaN(x)`	Checks if x is **not-a-number** (**NaN**).	`Number.isNaN(10 / 'hello');`

***Table 11.9**: Mathematical functions*

Strings

Strings are used for storing and manipulating text. They are immutable, meaning once a string is created, it cannot be changed, but methods can be used to generate new strings based on modifications. Strings are enclosed in double quotes (""), single quotes ('), or backticks (`). Double and single quotes work the same way, but backticks allow embedded expressions and

multi-line strings, as follows:

```
let greeting = "Hello there";
let greeting = 'Hello there';
let greeting = `Hello there, Welcome to JavaScript`;
```

String operations

String operations allow for various manipulations, such as concatenation, formatting, and handling escape sequences.

Concatenation joins strings using the **+** operator, as:

```
let message = "Hello" + " " + "World";
```

Template literal allows embedding expressions inside backticks using **${}**, with the variable you want to embed inside the braces, as:

```
let name = "John";
let message = `Hello, ${name}!`;
```

Escape sequences are special characters that can be used with a backslash (****). These are used to represent characters that are difficult or impossible to type directly in a string. They are useful for formatting text, inserting special characters, or encoding non-printable characters, as follows:

- Formatting text (e.g., **\n** for a new line, **\t** for a tab).
- Inserting special characters (e.g., **\'** for a single quote in a single-quoted string).
- Encoding non-printable characters (e.g., **\0** for a null character, **\b** for a backspace).
- Representing Unicode characters (e.g., **\u2764\uFE0F** for ♡).

The following table shows some examples of various string operations:

Code	Output
`let greeting = "Hello\nWorld";`	`Hello` `World`
`let quote = "She said, \"This is fun!\"";`	`She said, "This is fun!"`
`let contraction = 'It\'s a great day!';`	`It's a great day!`
`let filePath = "C:\\Users\\John";`	`C:\Users\John`

`let heartEmoji = "\u2764\uFE0F";`	♡

***Table 11.10**: String operations*

You can use any of the following escape characters:

Escape sequence	Description	Example usage
`\'`	Single quote	`'It\'s a nice day'`
`\"`	Double quote	`"He said, \"Hello!\""`
`\\`	Backslash	`"C:\\Users\\John"`
`\n`	New line	`"Hello\nWorld"`
`\r`	Carriage return	`"Hello\rWorld"`
`\t`	Tab	`"Hello\tWorld"`
`\b`	Backspace	`"Hello\bWorld"`
`\f`	Form feed	`"Hello\fWorld"`
`\v`	Vertical tab	`"Hello\vWorld"`
`\0`	Null character	
`\uXXXX`	Unicode character, eg, "©"	`"\u00A9"`
`\xXX`	Hexadecimal character, eg: "©"	`"\xA9"`
`\u{XXXXX}`	Unicode code point (useful for emojis and symbols, eg, 😀)	`"\u{1F600}"`

***Table 11.11**: Escape characters*

Useful string methods

Strings are one of the most frequently used data types in JavaScript and are used for handling user input, formatting text, and processing data. JavaScript provides built-in string methods that allow you to manipulate, extract, and transform text. These methods modify string content without altering the original value since strings in JavaScript are immutable.

Indexing a string

You can access individual characters in a string using square brackets `[]`.

For example:

```
let email = "user@example.com";
let thirdChar = email[2];
```

We can index this as shown in *Table 11.12*. The characters are indexed starting from **0**. So, the character in **email[2]** is **e**.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
u	s	e	r	@	e	x	a	m	p	l	e	.	c	o	m

***Table 11.12**: Indexing a string*

Getting the length of a string

Knowing the length of a string is essential when validating input, processing text, or ensuring content fits within a certain limit. You can use the **.length** property to determine the total number of characters in a string, including spaces and punctuation, as:

```
let message = "This is Fun!";
let length = message.length;
```

Changing case

Sometimes, text needs to be converted to all uppercase or lowercase for consistency, such as when normalizing user input or making case-insensitive comparisons. JavaScript provides two methods for changing case, as follows:

- **.toUpperCase()**: Converts all characters in a string to uppercase.
- **.toLowerCase()**: Converts all characters in a string to lowercase.

For example:

```
let text = "This is Fun!";
let upperText = text.toUpperCase();
let lowerText = text.toLowerCase();
```

This declares a variable text and assigns it the string **This is Fun!**.

The method **.toUpperCase()** converts all the characters in text to uppercase, and the output for which is **THIS IS FUN!**.

The method **.toLowerCase()** converts all the characters in text to lowercase, which outputs **this is fun!**.

Extracting a portion of a string

Sometimes, only part of a string is needed, such as extracting a username from an email address or getting a specific section of text. The **`.slice(start, end)`** method allows us to extract a portion of a string based on index positions, as:

```
let email = "user@example.com";
let username = email.slice(0, email.indexOf("@"));
```

This declares a variable email and assigns it the string **`user@example.com`**.

The method **`email.indexOf("@")`** finds the index position of the **`@`** symbol in the email string. In the email string, **`@`** is at index 4. This method returns 4. The following table shows how this is indexed; **`@`** is at index 4:

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
u	s	e	r	@	e	x	a	m	p	l	e	.	c	o	m

***Table 11.13** : Indexing*

The method **`.slice(0, email.indexOf("@"))`** extracts the portion of the string from index **`0`** up to but not including the **`@`**.

Replacing part of a string

When working with dynamic content, we often need to replace certain words or characters within a string. The **`.replace(search, replace)`** method allows us to replace the first occurrence of a specified value with a new one, as:

```
let sentence = "This is Java!";
let newSentence = sentence.replace("Java", "JavaScript");
```

Splitting a string into an array

Breaking a string into smaller parts is useful when parsing CSV data, breaking up sentences, or processing user input. The **`.split(separator)`** method divides a string into an array based on a specified delimiter, as:

```
let csvData = "apple,banana,grape,orange";
let fruits = csvData.split(",");
```

Booleans

Booleans represent logical values that are either true or false. They are commonly used in control flow statements and logical operations, as follows:

- The AND (**&&**) operator returns true only if both operands are true. If either operand is false, it returns false.
- The OR (**||**) operator returns true if at least one operand is true. Returns false only if both operands are false.
- The NOT (**!**) operator negates a Boolean value. In other words, it converts true to false and false to true.

Operator	Name	Description
&&	AND	Returns true if both operands are true.
||	OR	Returns true if at least one operand is true.
!	NOT	Negates a Boolean value.

***Table 11.14**: Booleans*

Logical operators can be combined to form more complex expressions. When combining logical operators, the operators are evaluated in this order:

! (NOT)

&& (AND)

|| (OR)

For example:

```
let a = true;
let b = false;
let c = true;
let result = a || b && c;
```

Here, **`b && c`** is evaluated first, as AND has higher precedence than OR.

Since **&&** has a higher precedence than **||**, sometimes we can use parentheses to control how expressions are evaluated, as:

```
let result = (a || b) && c;
```

Now, **`a || b`** will be evaluated first.

Output

The **`document.write()`** method is used to write content directly into an HTML document. It is commonly used for testing and debugging purposes to display output on the web page. However, it should be used cautiously, as calling **`document.write()`** after the document has loaded can overwrite the entire page content.

The **`console.log()`** method is another method commonly used for outputting data in JavaScript. Unlike **`document.write()`**, which writes directly to the web page, **`console.log()`** sends messages to the browser's developer console. This is useful for debugging and testing JavaScript code without modifying the web page's content. It is often used to check variable values, track errors, and understand how code executes.

Integration with HTML

JavaScript can be integrated with HTML to create dynamic and interactive web pages and can be included inside a **`<script>`** tag within an HTML file.

Place **`<script>`** at the end of **`<body>`** for best performance and to avoid errors. If the script must be in **`<head>`**, use the defer keyword, as shown:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>JavaScript Example</title>
</head>
<body>
</body>
    <script>
        ...
    </script>
</html>
```

JavaScript can also be placed in a separate **`.js`** file and linked to the HTML

document using a **<script>** tag, as:

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

Flow control

Flow control refers to the order in which statements are executed in a program. By default, JavaScript executes code line by line, from top to bottom, but flow control allows you to change this execution order based on conditions, loops, or function calls.

Conditional statements

Conditional statements allow a program to execute different code blocks based on certain conditions. Remember, a code block is code defined inside braces { }.

If statement

The if statement executes a block of code only if the given condition evaluates to true. You can define if statements like the following:

```
if (condition) {
    // Code to execute if condition is true
}
```

For example:

```
let number = 5;
if (number > 0) {
    console.log("The number is positive.");
}
```

If-else statement

The if-else statement runs one block if the condition is true and another block if it is false. You can define if-else statements like the following:

```
if (condition) {
    // Code to execute if condition is true
} else {
```

```
    // Code to execute if condition is false
}
```

For example:

```
let number = 10;
if (number % 2 == 0) {
    console.log("The number is even.");
} else {
    console.log("The number is odd.");
}
```

Switch statement

The switch statement is used when a variable is compared against multiple values. The break statement stops execution once a match is found, and the default case runs if no match is found. You can define a switch statement like the following:

```
switch (expression) {
    case value1:
        // Code to execute if expression === value1
        break;
    case value2:
        // Code to execute if expression === value2
        break;
    default:
        // Code to execute if no match is found
}
```

For example:

```
let day = "Monday";
switch (day) {
    case "Monday":
        console.log("Start of the workweek.");
        break;
    case "Friday":
        console.log("Weekend is coming!");
```

```
        break;
    case "Saturday":
    case "Sunday":
        console.log("It's the weekend!");
        break;
    default:
        console.log("It's a regular weekday.");
}
```

Loops

Loops allow a program to execute a block of code multiple times. Instead of manually writing the same code over and over, loops allow you to execute the same lines of code multiple times. They are particularly useful for iterating over arrays, performing calculations, or processing data structures such as objects and lists. By using loops, a program can efficiently execute operations on multiple data elements without redundancy.

Loops consist of three main components: initialization, condition, and iteration/update. The initialization step sets up a loop control variable, which acts as a counter or tracker for the loop.

The condition determines whether the loop should continue running or not. If the condition evaluates to true, the loop executes the code inside its block. If it evaluates to false, the loop stops.

The iteration/update step modifies the loop control variable or counter. In a for loop, this step occurs after each iteration of the loop block, before the next condition check. In a while or do-while loop, this step is typically placed inside the loop block and executes before the next condition check. This moves the loop control variable or counter closer to a value that will eventually make the condition false, ensuring that the loop terminates.

For loop

A for loop is used when the number of iterations is known. You can define a for loop like the following:

```
for (initialization; condition; counter++) {
    // Loop block: Code to execute
```

```
}
```

For example, print numbers 1 to 5, as:

```
for (let i = 1; i <= 5; i++) {
    console.log(i);
}
```

While loop

A while loop allows you to repeat code while a condition is true. You can define a while loop like the following:

```
while (condition) {
    // Loop block: Code to execute
    counter++
}
```

For example, count down from 5, as:

```
let count = 5;
while (count > 0) {
    console.log(`Countdown: ${count}`);
    count--;
}
console.log("Blast off! 🚀");
```

do-while loop

A do-while loop always runs at least once, regardless of the condition, and will continue while a condition is true. You can define a do-while loop like the following:

```
do {
    // Loop block: Code to execute
    counter++
} while (condition);
```

For example, print numbers from 1 to 5, as:

```
let i = 1;
do {
```

```
    console.log(i);
    i++;
} while (i <= 5);
```

Functions, parameters, and arguments

A function is a block of reusable code that performs a specific task. It allows you to execute the same code multiple times with different inputs. The inputs defined in the function are called parameters, and the actual values passed to the function are called arguments. Functions help make the program more modular, readable, and maintainable.

Defining a function

You can define functions using the following syntax:

```
function functionName(parameters) {
  // code to be executed
  return result;
}
```

For example, we can define a simple function as follows, with any parameters in parentheses:

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

We can call the function by name with the arguments in parentheses, as:

```
let result = add(5, 3);
```

Types of function parameters

There are different types of parameters you can use to handle various input scenarios, as follows:

- **Required parameters**: These are the parameters where arguments must be provided when the function is called, as:

  ```
  function multiply(x, y) {
    return x * y;
  }
  ```

```
let product1 = multiply(4, 5);  // product1 is 20
let product2 = multiply(4);     // product2 is NaN
(y is undefined)
```

- **Default parameters (ES6)**: Default parameters assign a fallback value to a parameter if no argument or an undefined value is passed, as:

```
function greet(name = "Guest") {
  return `Hello, ${name}!`;
}
let greeting1 = greet("Alice"); // greeting1 is
"Hello, Alice!"
let greeting2 = greet();        // greeting2 is
"Hello, Guest!" (default used)
```

- **Rest parameters**: Rest parameters allow a function to accept an indefinite number of arguments as an array, as:

```
function sum(...numbers) {
  return numbers.reduce((total, num) => total + num,
0);
}
let total1 = sum(1, 2, 3, 4, 5); // total1 is 15
let total2 = sum(10, 20);        // total2 is 30
```

Return statement

The return statement is used to return a value to the part of the program where the function was called. This allows functions to provide results that can be used elsewhere in your program.

When a function reaches a return statement, it stops executing immediately and outputs the value specified after return. If a function does not have a return statement, it returns undefined by default.

For example, the following code defines a function called **`add`** that takes two parameters, **`a`** and **`b`**. The function returns the sum of **`a`** and **`b`** using the **`return`** statement. The **`return`** statement stops the function's execution and sends the result back to the function call. Here, we define a function to add two numbers:

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

We can call the function like:

```
let result = add(5, 3);
```

When the function is called, we pass 5 and 3 as arguments. The function calculates 5 + 3 and returns the result (8 in this case).

The returned value 8 is assigned to the variable result using the statement let result **= add(5, 3);** as:

```
let result = add(5, 3);
```

Function scope

Function scope refers to the accessibility of variables within different parts of a program. Variables declared inside a function have local scope, meaning they are only accessible within that function. These variables cannot be accessed or modified outside the function in which they are declared, as follows:

```
function exampleFunction() {
    let localVar = "I am local"; // Declared inside the
function
    return localVar;
}
let result = exampleFunction(); // result is "I am
local"
```

Variables declared outside a function have global scope, meaning they can be accessed from anywhere in the program. If a function declares a variable using **let**, **const**, or **var**, that variable exists only within that function, preventing unintended modifications from external code. For example, if a function defines a variable and attempts to access it outside that function, an error will occur because the variable is not recognized in the global scope:

```
let globalVar = "I am global"; // Declared outside a
function
function showGlobal() {
```

```
    return globalVar; // Accessible inside the function
}
let result = showGlobal(); // result is "I am global"
```

Hoisting

Hoisting is a behavior in JavaScript where function and variable declarations are moved to the top of their scope before the code is executed. This means you can use a function or variable before it appears in the code, but how it behaves depends on how it is declared. JavaScript internally moves the function declaration to the top. So, for example, when you write this, with the function call above the declaration:

```
hello(); // Function call
function hello() {
    return "Hello, world!";
}
```

JavaScript internally moves the function declaration to the top, like:

```
function hello() {
    return "Hello, world!";
}
hello(); // Function call
```

Keep in mind that only function declarations are fully hoisted. This means the entire function (not just the name) is available anywhere in the scope. You can safely call the function before declaring it. Hoisting does not apply the same way to function expressions. If you assign a function to a variable (using **`var`**, **`let`**, or **`const`**), only the variable is hoisted, not the function itself.

Introduction to arrays

Arrays are data structures that allow storing multiple values in a single variable. They allow you to organize and manipulate collections of data, such as lists of numbers, names, or objects. Arrays are indexed, meaning each element has a specific position that allows for easy retrieval and modification.

Defining arrays

You can define an array by specifying the array elements in square brackets, as:

```
let fruits = ["Apple", "Banana", "Cherry"];
```

Accessing array elements

Array elements are accessed using zero-based indexing, meaning the first element has an index of 0. So, the array defined previously would be indexed as follows:

0	1	2
Apple	Banana	Cherry

***Table 11.15**: Accessing array elements*

We can access any of the elements in the array by specifying the index in square brackets, as:

```
let myfruit = fruits[1]
```

Modifying array elements

You can update array elements by assigning new values using their index, as:

```
fruits[1] = "Blueberry";
```

This will replace the element at index 1 with **`Blueberry`**, as shown:

0	1	2
Apple	Blueberry	Cherry

***Table 11.16**: Modifying array elements*

Adding and removing elements

There are various methods you can use to add or remove elements from an array. You can use **`push()`** to add more elements to the end of the array, as:

```
fruits.push("Mango");
```

This will add **`Mango`** to the end of the array, as shown:

0	1	2	3

Apple	Blueberry	Cherry	Mango

***Table 11.17**: Adding and removing elements*

You can use **`unshift()`** to add elements to the beginning of the array, as:

```
fruits.unshift("Strawberry");
```

This will add **`Strawberry`** to the beginning of the array, as shown:

0	1	2	3	4
Strawberry	Apple	Blueberry	Cherry	Mango

***Table 11.18**: Adding strawberry*

You can use **`pop()`** to remove the last element of the array, as:

```
let removedFruit = fruits.pop();
```

This will remove **`Mango`** from the end of the array, as shown:

0	1	2	3	4
Strawberry	Apple	Blueberry	Cherry	Mango

***Table 11.19**: Removing mango*

You can use **`shift()`** to remove the first element of the array (**`Strawberry`** in this case), as:

```
let firstRemovedFruit = fruits.shift();
```

The result is shown in the following table:

0	1	2
Apple	Blueberry	Cherry

***Table 11.20**: Removing strawberry*

Length of an array

You can use the length property to find the length of the array, meaning it returns the number of elements in the array, as:

```
let numberOfFruits = fruits.length;
```

Iterating over an array

You can loop through an array using a for loop, as:

```
for (let i = 0; i < fruits.length; i++) {
```

```
    let fruit = fruits[i];
    ...
}
```

Here, the loop starts with **`i = 0`** (the first index). It runs as long as **`i < fruits.length`**, meaning the loop continues iterating over the array until **`i`** reaches the last valid index, which is contained in the length property of the array. Inside the loop, **`fruits[i]`** accesses each element by its index represented by the loop counter **`i`**.

Splice and slice

If you want to extract elements from an array, you can use the splice and slice methods.

The **`splice()`** method allows you to remove, add, or replace elements in an array, directly modifying the original array, as:

```
let fruits = ["Apple", "Banana", "Cherry", "Mango"];
let removed = fruits.splice(1, 2);
```

Here, the first argument, 1, specifies the starting index where the modification begins, which is **`"Banana"`**. The second argument, 2, indicates the number of elements to remove, meaning both **`"Banana"`** and **`"Cherry"`** are removed from the original array. The removed elements are stored in the removed variable, which now contains **`["Banana", "Cherry"]`**, as shown in the following table:

0	1	2	3
Apple	Banana	Cherry	Mango

***Table 11.21**: Removed variables*

Meanwhile, the original fruit array is modified, and its new value becomes **`["Apple", "Mango"]`**, as the two middle elements have been removed, as shown:

0	1
Apple	Mango

***Table 11.22**: Modified fruit array*

The **slice()** method is used to extract a portion of an array without altering the original, returning a new array with the selected elements, as:

```
let fruits = ["Apple", "Banana", "Cherry", "Mango"];
let extracted = fruits.slice(1, 3);
```

Here, the first argument, 1, indicates the starting index, meaning extraction begins at **"Banana"**. The second argument, 3, specifies the ending index but is not included in the result, so the extraction stops before **"Mango"**. As a result, the extracted variable stores **["Banana", "Cherry"]**, which are the elements at indexes 1 and 2, as shown:

0	1	2	3
Apple	Banana	Cherry	Mango

Table 11.23: *Extracted variables*

The original fruits array remains unchanged as **["Apple", "Banana", "Cherry", "Mango"]**, since **slice()** only creates a new array with the selected elements.

Objects and classes

Objects and classes are core concepts in **object-oriented programming** (**OOP**). A class is a blueprint for creating objects. An object represents a real-world entity and can contain data called **properties** and behaviors called **methods**. Using objects and classes allows developers to model complex systems, organize code efficiently, and keep the code easily maintainable.

You can define a class using the **class** keyword. A class typically contains a constructor (which initializes object properties) and methods (functions that define the behavior of the objects), as follows:

```
class User {
    constructor(username, email, role) {
        this.username = username;
        this.email = email;
        this.role = role;
    }
    // Method to update the user role
```

```
    updateRole(newRole) {
        this.role = newRole;
    }
    // Method to display user details
    getProfile() {
        return `User: ${this.username}, Email:
${this.email}, Role: ${this.role}`;
    }
}
```

Here, the class **User** serves as a blueprint for creating user objects with three properties: **username**, **email**, and **role**. The constructor initializes these properties when a new user object is created.

Note: `this` **is a special keyword that refers to the object that is currently executing the function.**

The class includes a method, **updateRole(newRole)**, which allows you to modify the user's role. The **getProfile()** method returns a formatted string containing the user's details.

Once a class is defined, you can create multiple objects (or instances) from it and use their properties and methods. Let us create an **adminUser**, as follows:

```
let adminUser = new User("AliceJohnson",
"alice@company.com", "Admin");
```

This creates a new instance of the **User** class called **adminUser**. By using the new keyword, the constructor of the **User** class is invoked, initializing the object with the values **"AliceJohnson"** for the username, **"alice@company.com"** for the email, and **"Admin"** for the role. These were passed to the object when it was created. Since it is an instance of the **User** class, it can access class methods like **updateRole()** to modify the user's role or **getProfile()** to retrieve user details.

We can update the user's role using the **updateRole()** method. We can do this using the dot notation: **object_name.method(arguments)**, as:

```
adminUser.updateRole("Super Admin");
```

We can also get the updated profile details using the **`getProfile()`** method, as:

```
let updatedProfile = adminUser.getProfile();
```

Basic error handling with try...catch

Error handling is a key aspect of writing robust programs that can gracefully handle unexpected situations. The try...catch statement allows developers to catch and manage runtime errors without crashing the whole program. By wrapping potentially problematic code inside a try block and defining a catch block to handle exceptions, developers can ensure applications remain stable and provide meaningful feedback when an error occurs:

```
try {
  // Code that may throw an error
} catch (exceptionVar) {
  // Code to handle the error
} finally {
  // (Optional) Code that runs after try/catch, no
matter what
}
```

Here, the **`try`** block contains the code that might throw an error. If an error occurs inside the **`try`** block, execution is stopped inside that block and jumps to the catch block. The **`catch`** block receives an error object that contains details about the error. If no error occurs in the **`try`** block, the **`catch`** block is skipped. The **`finally`** block is optional. If present, its code will execute after the **`try`** (and **`catch`**) blocks finish, regardless of whether an error was thrown or not.

Let us take a look at an example. The following function takes two numerical inputs and divides one by the other. Inside the **`try`** block, it checks if b is equal to 0. If b is equal to 0, it throws an error with the message *Cannot divide by zero*. The catch block then catches this. Instead of crashing the program, the function returns the error message. If b is any

non-zero number, the function returns the result of **`a / b`**:

```
function divideNumbers(a, b) {
    try {
        if (b == 0) {
            throw new Error("Cannot divide by zero.");
        }
        return a / b;
    } catch (error) {
        return error.message;
    }
}
let result2 = divideNumbers(8, 0);
```

JavaScript has several built-in error types for different kinds of exceptions: **`ReferenceError`**, **`TypeError`**, **`SyntaxError`**, **`RangeError`**, etc. In a **`catch`** block, you can handle all errors generically or differentiate based on the error type.

Conclusion

JavaScript is a widely used programming language for both client-side and server-side development. It supports multiple programming paradigms, including object-oriented, functional, and imperative styles. With its ability to run in web browsers and server environments, such as Node.js, JavaScript is commonly used to develop modern web applications.

This chapter covered JavaScript's syntax, including its case sensitivity, variable declaration with var, let, and const, and the importance of structured code blocks using curly braces {}.

Control structures, such as conditional statements (if, else, switch) and loops (for, while, do-while) were discussed, highlighting their role in controlling program execution.

The chapter explored functions as a key feature for code reusability, with support for parameters, return values, and different declaration styles.

Arrays and objects were introduced as essential data structures for

organizing and managing information.

Additionally, the chapter also discussed error handling using try...catch as a method to manage runtime errors and ensure program stability.

In the next chapter, the focus shifts to the DOM. Readers will learn how to interact with HTML elements programmatically—selecting, modifying, and styling elements with JavaScript, dynamically creating or removing content, and handling user interactions through event-driven programming.

Points to remember

- JavaScript's flexibility in supporting different programming paradigms, such as object-oriented, functional, and imperative styles.
- Essential syntax rules, including case sensitivity, variable declarations with var, let, and const, and the use of semicolons.
- Flow control mechanisms, such as conditional statements (if, else, switch) and loops (for, while, do-while) that dictate the execution of code.
- Functions, which enable code reusability, with support for parameters, return values, and different types of function declarations.
- Data structures, including arrays for storing lists of values and objects for structuring complex data.
- Error handling using try...catch, which ensures programs can handle unexpected errors without crashing.

Exercises

1. What is JavaScript, and how is it used in web development?
2. What are the key features of JavaScript?
3. How does JavaScript handle variable declarations, and what is the difference between var, let, and const?
4. What are reserved keywords in JavaScript, and why can they not be used as variable names?
5. Write a JavaScript program that declares a variable, firstName, and

assigns it the value "Alice".

6. Write a JavaScript program that declares a constant PI with the value 3.14, then calculates the area of a circle (area = πr^2).
7. What are control structures in JavaScript, and how do they affect program execution?
8. How do if, else, and switch statements work in JavaScript?
9. Write a JavaScript program that checks if a number is positive, negative, or zero using an if-else statement.
10. Write a JavaScript program that takes a day of the week as input and prints whether it is a weekday or a weekend using a switch statement.
11. What are loops, and how do for, while, and do-while loops function?
12. Write a for loop that prints numbers from 1 to 10.
13. Write a while loop that prints even numbers between 2 and 20.
14. Write a do-while loop that prompts the user for a number greater than 10 and keeps asking until they provide a valid number.
15. What is a function in JavaScript, and why is it useful?
16. What is the difference between function declarations and function expressions?
17. Write a function greet(name) that takes a name as input and returns a greeting message.
18. Write a function sum(a, b) that returns the sum of two numbers.
19. What is the purpose of the return statement in a function?
20. What are default parameters, and how are they useful in functions?
21. Write a function multiply(a, b = 2) that multiplies two numbers, using 2 as the default value for b.
22. What are arrays, and how do you define them in JavaScript?
23. Write a JavaScript program that creates an array of three fruits and prints the second fruit.
24. How can you access and modify elements in an array?
25. What methods can be used to add or remove elements from an array?
26. Write a JavaScript program that adds *Mango* to an array of fruits and removes the first element.

27. What is the difference between splice() and slice() methods?
28. Write a JavaScript program that removes the second element from an array using splice().
29. Write a JavaScript program that extracts the first three elements from an array using slice().
30. What is an object in JavaScript, and how is it different from an array?
31. Write an object person with properties name, age, and city, and print the value of the name property.
32. How do you define a class in JavaScript?
33. Write a JavaScript class Car with properties brand and year,' and a method getInfo() that returns a string with the car's details.
34. What is the this keyword, and how is it used in JavaScript?
35. Write a JavaScript program that creates an instance of the Car class and prints the car’s details using getInfo().
36. What is hoisting, and how does it affect variable and function declarations?
37. How does JavaScript handle errors, and what is the purpose of try...catch?
38. Write a JavaScript program that attempts to divide by zero inside a try block and catches the error.
39. What are logical operators in JavaScript, and how do they work?
40. What is the difference between == and === operators?
41. Write a JavaScript program that checks if a number is between 10 and 20 using logical operators.
42. How can JavaScript be integrated with HTML?
43. What is the purpose of document.write() and console.log()?
44. Write a JavaScript program that prints *Hello, World!* to the console using console.log().
45. What is the difference between push(), pop(), unshift(), and shift() methods in arrays?
46. Write a JavaScript program that demonstrates the use of push(), pop(), unshift(), and shift().

47. Write a JavaScript function that takes an array of numbers and returns the sum of all elements using a loop.

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com



CHAPTER 12
Working with the DOM

Introduction

In this chapter, you will learn about the **Document Object Model** (**DOM**) and its pivotal role in web development. This chapter covers selecting and manipulating HTML elements to dynamically update content and styles. You will learn how to alter styles with JavaScript, handle various user interactions like clicks and input events, and create or remove elements on the fly. By the end, you will have a solid understanding of how to interact with and modify the DOM to enhance user experiences on your websites.

Structure

The chapter covers the following topics:

- Understanding the Document Object Model
- Selecting and manipulating HTML elements
- Changing styles with JavaScript
- Creating and removing elements dynamically
- Event handling

Objectives

By the end of this chapter, you will be able to understand the DOM and its structure, as well as explain how the browser constructs the DOM from an HTML document. You will be able to identify different types of nodes in the DOM, including document, element, attribute, and text nodes.

Additionally, you will learn how to select HTML elements using various DOM selection methods, such as `getElementById()`, `getElementsByClassName()`, `getElementsByTagName()`, `querySelector()`, and `querySelectorAll()`.

You will also be able to modify content and attributes of elements dynamically using properties like `innerHTML`, `textContent`, and `innerText`, and change element attributes with methods

like **`setAttribute()`**.

You will be able to manipulate CSS styles dynamically using the style property and manage classes with **`classList`** methods.

By learning how to create, append, and remove elements dynamically, you will be able to modify the DOM structure in real time.

Lastly, you will understand event handling in JavaScript, including how to use **`addEventListener()`**, work with the event object, implement event delegation, and prevent default actions, allowing you to build interactive and dynamic web applications.

Understanding the Document Object Model

The DOM is a tree-like representation of an HTML document created by the web browser when a web page loads. The DOM tree represents the hierarchical structure of the HTML elements on the page. The DOM acts as an interface between the page's content and JavaScript, allowing scripts to modify the structure dynamically. For example, let us say we have a simple web page, as follows:

```
<!DOCTYPE html>
<html>
<head>
    <title>Title</title>
</head>
<body>
    <h1>Welcome to My Website</h1>
    <p>This is an example paragraph.</p>
</body>
</html>
```

This HTML would be converted into the following DOM structure, as shown in *Figure 12.1*. Here, each node represents an element used in the HTML document.

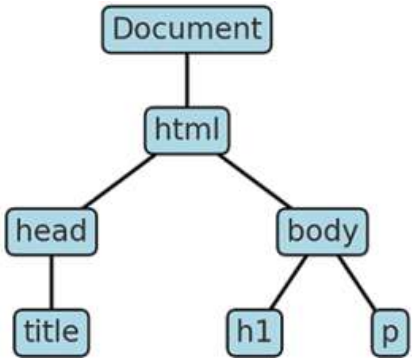


***Figure 12.1**: Document Object Model tree*

Each node has properties that describe its characteristics, relationships, and behavior within the document. For example, in *Figure 12.2*, the **`<p>`** paragraph element has the property

`.innerHTML` as well as many others:

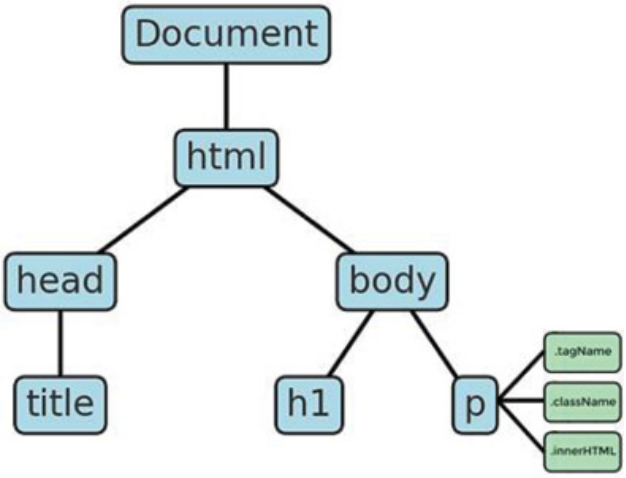


***Figure 12.2**: Document Object Model tree with properties*

Other nodes in the DOM tree also have properties. The document node has the following properties:

Property	Description
`.documentElement`	Returns the `<html>` element.
`.head`	Returns the `<head>` element.
`.body`	Returns the `<body>` element.
`.title`	Gets/sets the `<title>` content.
`.URL`	Returns the full page URL.
`.domain`	Returns the website domain.
`.cookie`	Gets/sets cookies.
`.referrer`	Returns the referring page URL.
`.readyState`	Returns the document loading state (loading, interactive, complete).
`.activeElement`	Returns the currently focused element.
`.forms`	Collection of all `<form>` elements.
`.images`	Collection of all `<img>` elements.
`.links`	Collection of all `<a>` elements with href.
`.scripts`	Collection of all `<script>` elements.

***Table 12.1**: Document node properties*

The elements nodes, such as `<h1>`, `<div>`, and `<p>`, have the following properties:

Property	Description
`.tagName`	The tag name of the element (e.g., "DIV", "P").

Property	Description
`.id`	The id attribute of the element.
`.className`	The class attribute as a string.
`.classList`	List of class names (DOMTokenList).
`.innerHTML`	The HTML inside the element.
`.outerHTML`	The element and its inner HTML.
`.textContent`	The text inside the element (ignores HTML).
`.innerText`	Similar to `.textContent` but respects CSS styling.
`.attributes`	A collection of all attributes of the element.
`.style`	Access to inline CSS styles.
`.hidden`	Boolean, controls element visibility.
`.clientWidth / .clientHeight`	Element width/height including padding.
`.offsetWidth / .offsetHeight`	Element width/height including padding and border.
`.scrollWidth / .scrollHeight`	Scrollable width/height of the element.
`.parentElement`	The parent node (if it is an element).
`.children`	A collection of child elements.
`.firstElementChild`	The first child element.
`.lastElementChild`	The last child element.
`.nextElementSibling`	The next sibling element.
`.previousElementSibling`	The previous sibling element.
`.dataset`	Access custom data-* attributes.

***Table 12.2**: Element node properties*

The anchor node `<a>` has the following properties:

Property	Description
`.href`	Gets/sets the full URL (absolute).
`.target`	Gets/sets where to open the link (`_self`, `_blank`, etc.).
`.rel`	Gets/sets the relationship (e.g., `"noopener"`).
`.innerText`	Gets/sets the visible text inside the `<a>` tag.
`.textContent`	Gets/sets the raw text inside `<a>`.

***Table 12.3**: Anchor node properties*

The image node `<img>` has the following properties:

Property	Description
`.src`	Gets/sets the image source URL.

`.alt`	Gets/sets the alternative text.
`.width`	Gets/sets the image width.
`.height`	Gets/sets the image height.
`.naturalWidth`	Returns the original image width.
`.naturalHeight`	Returns the original image height.

***Table 12.4**: Image node properties*

Selecting and manipulating HTML elements

Using the following example, we can select and manipulate HTML elements with JavaScript. Before you can manipulate the element in the DOM, you need to select it (retrieve the corresponding node object), as follows:

```
<!DOCTYPE html>
<html>
<head>
    <title>Title</title>
</head>
<body>
    <h1>Welcome to My Website</h1>
</body>
</html>
```

There are various element selection methods we can use; common methods are listed in the following table:

Method	Description	Example usage
`document.getElementById(id)`	Selects an element by its ID. Returns a single element.	`document.getElementById("");` (Put element ID in the double quotes.)
`document.getElementsByClassName(className)`	Selects elements by class. Returns an HTMLCollection (live list).	`document.getElementsByClassName("");` (Put `classname` in the double quotes)
`document.getElementsByTagName(tagName)`	Selects elements by tag name. Returns an HTMLCollection.	`document.getElementsByTagName("");` (Put tag in the double quotes, e.g., `"p"`, `"div"`)
`document.querySelector(selector)`	Selects the first element matching a CSS selector. Returns a single element or null.	`document.querySelector("");` (Put `css` selector in the double quotes, e.g, `.btn`).
`document.querySelectorAll(selector)`	Selects all	`document.querySelectorAll("");`

	elements matching a CSS selector. Returns a NodeList.	(Put **css** selector in the double quotes, e.g, **.btn**).

***Table 12.5**: Element section methods*

The first method we can use is **getElementById()**. This method selects an element by its ID, so this will only work if the HTML elements have an ID attribute assigned. We can add an ID to the **<h1>** heading element, as follows:

```
<h1 id="mainheading">Welcome to My Website</h1>
```

Once we have an ID assigned to the element, we can select it using the JavaScript code:

```
document.getElementById("mainheading");
```

This returns the element with the matching ID attribute, or null if none is found. IDs are unique, so this method returns at most one element.

Now that the element is selected, if you want to change the HTML inside the element, you can use the **.innerHTML** property, as follows:

```
document.getElementById("mainheading").innerHTML = "Welcome to JavaScript";
```

Now, we can add this code to our example:

```
<!DOCTYPE html>
<html>
<head>
    <title>Title</title>
</head>
<body>
    <h1 id= "mainheading">Welcome to My Website</h1>
</body>
<script>
    document.getElementById("mainheading").innerHTML = "Welcome to JavaScript";
</script>
</html>
```

The following are some other properties that can be used:

Property	Description	Example usage
innerHTML	Gets or sets the HTML content inside an element.	**.innerHTML = "Bold Text";**
textContent	Gets or sets the text inside an element (ignores HTML formatting).	**.textContent = "Plain Text";**
innerText	Similar to textContent, but respects hidden elements (less commonly used).	**.innerText = "Visible Text";**
style	Modifies the inline styles of an element.	**.style.color = "blue";**
className	Sets or gets the class name(s) of an element (overwrites existing classes).	**.className = "newClass";**

Property	Description	Example usage
`classList`	Provides methods to add, remove, or toggle CSS classes.	`.classList.add("highlight");`
`id`	Gets or sets the ID of an element.	`.id = "newID";`
`value`	Gets or sets the value of form elements (e.g., input, textarea).	`.value = "Hello!";`
`src`	Gets or sets the src attribute of an `<img>` or `<video>`.	`.src = "image.jpg";`
`href`	Gets or sets the href attribute of an `<a>` (link).	`.href = "https://example.com";`
`disabled`	Enables or disables form elements (buttons, inputs).	`.disabled = true;`
`checked`	Gets or sets the checked state of checkboxes/radios.	`.checked = true;`
`readOnly`	Makes an input field read-only (cannot be edited).	`.readOnly = true;`
`hidden`	Hides an element (similar to display: none;).	`.hidden = true;`
`title`	Sets or gets the tooltip text of an element.	`.title = "Hover over me!";`

***Table 12.6**: Properties*

Changing styles with JavaScript

You can modify an element's style dynamically using JavaScript by accessing its `.style` property, as follows:

```
document.getElementById("mainheading").style.color = "blue";
```

You can also change other properties, such as font size, style, and so on, as discussed in the following table:

Property	Effect	Example
`color`	Changes text color	`.style.color = "blue";`
`backgroundColor`	Changes the background color	`.style.backgroundColor = "yellow";`
`fontSize`	Changes text size	`.style.fontSize = "24px";`
`fontWeight`	Changes text weight (bold, normal)	`.style.fontWeight = "bold";`
`fontStyle`	Changes text style (italic, normal)	`.style.fontStyle = "italic";`
`textAlign`	Aligns text (left, right, center)	`.style.textAlign = "center";`
`textDecoration`	Underlines, overlines, or strikes text	`.style.textDecoration = "underline";`
`visibility`	Hides or shows an element	`.style.visibility = "hidden";`
`display`	Removes an element from the layout	`.style.display = "none";`
`border`	Adds a border around an element	`.style.border = "2px solid black";`

Property	Effect	Example
`borderRadius`	Rounds the corners of an element	`.style.borderRadius = "10px";`
`width`	Changes the width of an element	`.style.width = "200px";`
`height`	Changes the height of an element	`.style.height = "100px";`
`margin`	Adjusts the margin (spacing outside an element)	`.style.margin = "20px";`
`padding`	Adjusts padding (spacing inside an element)	`.style.padding = "10px";`
`opacity`	Makes an element transparent	`.style.opacity = "0.5";`
`boxShadow`	Adds a shadow effect	`.style.boxShadow = "5px 5px 10px gray";`
`cursor`	Changes the mouse pointer style	`.style.cursor = "pointer";`

***Table 12.7**: Changing styles*

Creating and removing elements dynamically

JavaScript allows you to create, append, and remove elements dynamically in the DOM. This is useful for interactive features like adding new list items, comments, notifications, or dynamically generated content.

To create an element dynamically, use **`document.createElement()`**, then set its content and append it to the page, as follows:

1. First, create a new element. In this case, we are adding a paragraph, as follows:
   ```
   let newParagraph = document.createElement("p");
   ```
2. Add some text content, as follows:
   ```
   newParagraph.textContent = "This is a dynamically created paragraph.";
   ```
3. Append it to the body (or another container), as follows:
   ```
   document.body.append(newParagraph);
   ```

Event handling

Event handling refers to the process of responding to actions or occurrences (called events) that take place in the web browser or system events. This can include mouse clicks, key presses, form submissions, or events triggered automatically by system events such as network responses, timers, or hardware status changes.

Types of events

The following table is a summary of the different types of events:

Event category	Event type	Description
Mouse events	`click`	The user clicks on an element.

	dblclick	The user double-clicks an element.
	mouseover	The cursor moves over an element.
	mouseout	The cursor leaves an element.
	mousemove	The cursor moves within an element.
	mousedown	The mouse button is pressed down.
	mouseup	The mouse button is released.
Keyboard events	**keydown**	A key is pressed.
	keyup	A key is released.
	keypress	A key-producing character input is pressed (deprecated but still used).
Form events	**submit**	Form is submitted.
	change	The value of the form input changes.
	focus	Element receives focus (clicked/tabbed into).
	blur	Element loses focus.
Window events	**resize**	Window size changes.
	scroll	The user scrolls the web page.
	load	The window or page fully loads.
	unload	Window or page unloads (user leaves page or refreshes).
Network events	**fetch**	Fetch request is initiated (handled via promises).
	error	A network error occurs.

***Table 12.8**: Summary of different types of events*

Event listeners

To handle events in JavaScript, we need to create an event listener. An event listener is a function that waits for an event to occur and executes a specified action when the event is triggered. When an event is triggered, JavaScript passes an event object to the event listener. This object contains information about the event (such as the target element, event type, mouse position, key pressed, etc.) and is passed as an argument to the event listener.

The most common way to create an event listener is to use the **`addEventListener()`** method. This method attaches an event listener to an element and waits for the specified event to occur. For example:

```
element.addEventListener(eventType, callbackFunction, useCapture);
```

In the preceding example, **`eventType`** is the name of the event (you can see a list of events in the preceding table). The **`callbackFunction`** function is executed when the event occurs. The **`useCapture`** argument is an optional Boolean value (default is false). If true, event capturing is enabled, as follows:

```
<button id="myButton">Click me!</button>
```

```
<script>
  const button = document.getElementById("myButton");

  button.addEventListener("click", function(event) {
      alert("Button clicked!");
      console.log(event);
  });
</script>
```

Conclusion

In this chapter, you explored the DOM and its importance in building dynamic, interactive websites. By understanding the DOM as a structured representation of your HTML document, you gained the ability to access and manipulate web page elements using JavaScript.

You learned how to select elements with methods like getElementById() and querySelector(), and how to modify their content, attributes, and styles in real time. You also discovered how to create and remove elements dynamically, allowing you to build responsive interfaces that update based on user interactions or data changes.

The chapter also introduced you to event handling—an essential concept in interactive web design. Using addEventListener(), you now know how to respond to user actions such as clicks, key presses, and form submissions, and how to manage event propagation and delegation effectively.

By mastering these skills, you can begin building web applications that respond intelligently to users and update content dynamically without needing to reload the page.

In the next chapter, you will shift your focus to planning and building complete websites. You will learn how to design intuitive user interfaces, set up your project structure, create and organize files efficiently, and apply testing strategies to ensure your website performs reliably across devices and browsers.

Points to remember

- The DOM is a tree-like representation of an HTML document that allows JavaScript to interact with and modify web pages.
- The DOM tree consists of nodes, including document nodes, element nodes, attribute nodes, and text nodes.
- You can select elements in the DOM using methods like document.getElementById(), document.getElementsByClassName(), document.getElementsByTagName(), document.querySelector(), and document.querySelectorAll().
- The innerHTML, textContent, and innerText properties allow you to change the content of elements.
- The style property allows you to modify CSS styles dynamically.
- The classList property provides methods to add, remove, toggle, and check for CSS classes.

- You can create new elements dynamically using document.createElement(), append them with element.appendChild(), and remove them with element.removeChild().
- Event handling allows JavaScript to respond to user actions such as clicks, keypresses, and mouse movements.
- The addEventListener() method attaches an event listener to an element.
- The event object provides useful properties like event.target, event.type, event.clientX, and event.key.
- Event delegation helps improve performance by attaching a single event listener to a parent element instead of multiple listeners on child elements.
- Event propagation occurs in two phases: the bubbling phase (default) and the capturing phase.
- The event.preventDefault() method stops the default action of an element, such as preventing form submission.
- The removeEventListener() method removes an event listener from an element.
- You can dynamically modify attributes like src, href, alt, value, and title using JavaScript.
- The document.readyState property helps determine whether the document is fully loaded.
- JavaScript allows you to create interactive web applications by dynamically adding, modifying, or removing content in the DOM.

Exercises

1. What is the DOM, and why is it important in JavaScript?
2. How does the browser create the DOM from an HTML document?
3. Explain the difference between document nodes, element nodes, and text nodes in the DOM.
4. What is the difference between document.getElementById() and document.querySelector()?
5. How do you select multiple elements in the DOM?
6. What is the difference between innerHTML, textContent, and innerText?
7. How can you dynamically create an HTML element using JavaScript?
8. What is the purpose of document.createElement() and appendChild()?
9. How can you remove an element from the DOM using JavaScript?
10. What is event delegation, and why is it useful?
11. How does the addEventListener() method work?
12. What is the event object, and how can it be used in an event listener?
13. What is the difference between event bubbling and event capturing?
14. How do you prevent the default behavior of an event?
15. Write the syntax of addEventListener() and explain each parameter.
16. How can you change the CSS style of an element using JavaScript?
17. What is the difference between className and classList?
18. How do you toggle a CSS class on an element using JavaScript?

19. Write a JavaScript code snippet to change the text of an <h1> element when a button is clicked.
20. Write a JavaScript program to add a new <li> item to an unordered list when a button is clicked.
21. Write a function that changes the background color of a element when the mouse moves over it.
22. Write a program that listens for a keypress event and displays the pressed key inside a <p> tag.
23. Write a JavaScript program that prevents a form from submitting and displays an alert instead.
24. Write a script that toggles the visibility of a paragraph when a button is clicked.
25. Create a counter that increases when a button is clicked and displays the count in a .
26. Write JavaScript code to change the image source of an <img> element when a button is clicked.
27. Create a simple to-do list where users can add and remove items dynamically using JavaScript.
28. Write a program that listens for the window resize event and displays the new dimensions in a <p> tag.
29. Write a script that changes the text color of all <p> elements to red when a button is clicked.
30. Write JavaScript code to remove an event listener from a button after it has been clicked once.

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com



CHAPTER 13

Building a Responsive Website from Scratch

Introduction

In this chapter, you will learn how to build a fully responsive website from the ground up. It starts with structuring the site using semantic HTML, followed by applying styles with CSS to create a visually appealing layout. You will then implement responsive design techniques, such as media queries, to ensure the site adapts seamlessly to different screen sizes. JavaScript will be introduced to make the site interactive, and finally, you will explore methods for testing and debugging the website for cross-browser compatibility and performance optimization.

Structure

The chapter covers the following topics:

- Planning
- User interface
- Creating the project
- Testing the website

Objectives

By the end of this chapter, you will be able to build a simple, fully responsive, and interactive web application from scratch. You will understand how to structure your HTML using semantic elements, apply CSS to create clean and modern layouts, and use flexbox to manage responsive design across different screen sizes. You will gain practical experience linking CSS and JavaScript to your HTML files and writing JavaScript to dynamically add, remove, and update content on the page. Additionally, you will learn how to persist data using localStorage to save user input even after refreshing the browser. Finally, you will be equipped with the skills to test and debug your application using browser developer tools, ensuring cross-browser compatibility and a polished user experience.

Planning

Before writing any code, we need to define the features of the website. The project is a basic *To-Do List* app where users can add and remove tasks. Before designing, we need to clarify the core features as follows:

- Add tasks.
- Mark tasks as completed.
- Remove tasks.
- Save tasks in local storage (so they persist after refreshing).

User interface

For this website, we will create a simple, clean, and minimal interface, with:

- **Header**: App title.
- **Input field**: To enter tasks.
- **Add button**: To add a task.
- **Task list**: Displays tasks added using the input field.
- **Action buttons**: **Done** and **Remove** buttons for each task.
  - **Done** puts a line through the task to mark it as complete.

o **Remove** removes the task from the list.

In *Figure 13.1*, we can design the interface using the aforementioned specification:

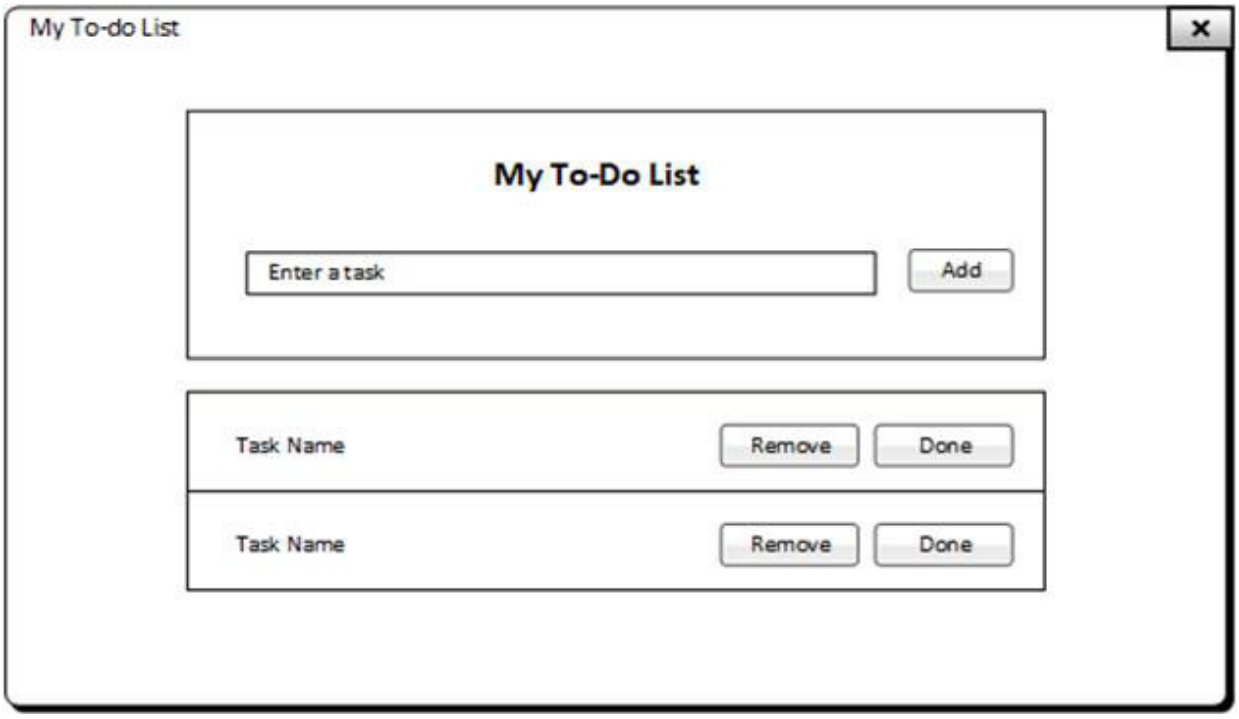


***Figure 13.1**: Wireframe design for To-Do app*

Creating the project

Based on the designs from the preceding section, we can now write the HTML, CSS, and JavaScript needed to implement the app. Here, we will implement the core functionality and UI.

For this project, we are going to use VS Code. If you have not already installed VS Code, download and install it from the official website.

Adding files

In VS Code, go to the **File** menu, then select **Open Folder**. Select and create a new folder where you want to store your project. For example, a folder named **ToDo**.

Right-click inside the **ToDo** folder in VS Code's file explorer. Select **New File** and name it accordingly. You will need three files as follows:

- **index.html**: The main HTML file.
- **styles.css**: For styling.
- **script.js**: For the JavaScript code.

You will end up with something like *Figure 13.2*:

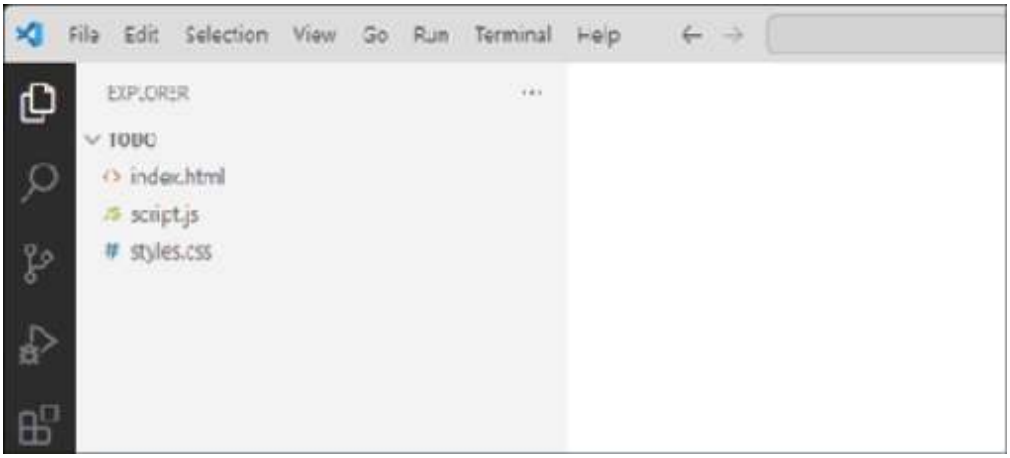


***Figure 13.2**: Project file listing in VS Code*

HTML

First, we create the basic structure of the app in HTML. This file creates the basic structure of the app, including the header, input field, add button, and task list.

The head section of the HTML document contains metadata and links to external resources necessary for proper functionality and styling. It starts with **`<meta charset="UTF-8">`**, which ensures that special characters and multiple languages are correctly displayed. The **`<meta name="viewport" content="width=device-width, initial-scale=1.0">`** makes the page responsive by adjusting its width to fit different screen sizes. The **`<title>To-Do List</title>`** sets the name that appears on the browser tab, as follows:

```
    <meta charset="UTF-8">
     <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>To-Do List</title>
```

Here, we also need to link the external CSS file to style the page. This is where we define the layout, colors, and typography, as follows:

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

The body section contains all the visible content of the To-Do List application, including the header, input field, task list, and buttons. First, we create a ** ** container with the **id="myDIV"** and **class="header"**, which holds the **<h2>** title **"My To-Do List"**. Below the title, an **<input>** field with **id="myInput"** allows users to type in tasks. Next to the field is a **<button>** with **class="addBtn"** and **onclick="newElement()"**. This calls a JavaScript function that adds new tasks to the list when clicked, as follows:

```
<div id="myDIV" class="header">
    <h2>My To-Do List</h2>
    <input type="text" id="myInput"
placeholder="Enter a task...">
    <button onclick="newElement()"
class="addBtn">Add</button>
</div>
```

An empty list **<ul>** with **id="myUL"** is included to store added tasks, as follows:

```
<ul id="myUL"></ul>
```

We need to link to our external JavaScript file that contains the JavaScript code for handling task creation, deletion, and completion, as follows:

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

When we add this to the HTML file, the output will be as shown in the following figure:

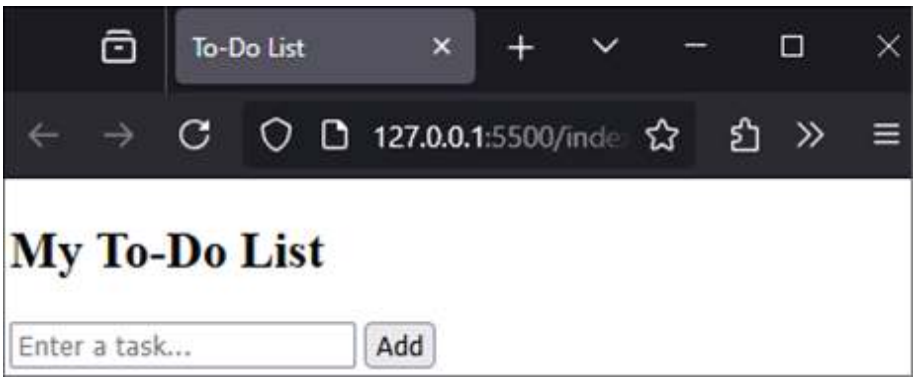


***Figure 13.3**: HTML output*

CSS

Once the HTML is complete, we need to style the page and add some color and layout. We can do this by adding CSS code to style the elements we added to the HTML file.

Style the body

First, we can style the whole body of the app. The body selector applies styles to the entire web page, as follows:

```
body {
    margin: 0;
    font-family: Arial, sans-serif;
    background: #f9f9f9;
    display: flex;
    flex-direction: column;
    align-items: center;
    min-width: 250px;
}
```

The **`margin: 0;`** ensures there is no default browser margin around the content. The **`font-family: Arial`**, **`sans-serif;`** sets the text font to Arial, with sans-serif as a fallback. The **`background: #f9f9f9;`** sets the page background to a light gray color, which helps create a subtle contrast against the white task list.

To control layout, we can use a flexbox. So, we set **`display: flex;`** This allows easy vertical stacking. The **`flex-direction: column;`** ensures that elements are arranged in a vertical format. The **`align-items: center;`** centers all content horizontally, ensuring that the list, input field, and buttons appear in the middle of the page. The **`min-width: 250px;`** prevents the page from becoming too narrow on smaller devices, improving usability.

The **`*`** selector applies the **`box-sizing: border-box;`** rule to all elements on the page, as follows:

```
* {
    box-sizing: border-box;
}
```

This ensures that when padding and borders are added to an element, they do not increase its overall width or height. This makes layout calculations easier and prevents unintended stretching of elements beyond their expected dimensions.

Styling the header

The **`.header`** selector styles the top section of the app, as follows:

```
.header {
    background: #f44336;
    padding: 20px;
    color: white;
    text-align: center;
    width: 100%;
    max-width: 400px;
    border-radius: 8px 8px 8px 8px;
    margin-top: 10px;
    margin-bottom: 20px;
}
```

The **`background: #f44336;`** gives it a red background. The **`padding: 20px;`** ensures the title and input fields have proper spacing. The **`color: white;`** makes the text white so it is readable against the red background. The **`text-align: center;`** centers the content. The **`border-radius: 8px;`** gives it slightly rounded corners.

Styling the text field

The input selector applies styles to the text input box where users enter their tasks, as follows:

```
input {
    width: 70%;
    padding: 10px;
    font-size: 16px;
    border: none;
    border-radius: 5px;
    outline: none;
}
```

The **`width: 70%;`** makes the input field take up 70% of its container's width. The padding: 10px; adds internal spacing inside the input field. The **`font-size: 16px;`** sets the text size inside the input field to 16px. The **`border: none;`** removes the default border around the input field. The **`border-radius: 5px;`** applies rounded corners to the input field, giving it a modern and smooth appearance. The outline: none; prevents the default blue outline that appears when the input field is clicked.

Styling the add button

The **`.addBtn`** selector applies styles to the **Add** button, which allows users to submit tasks. Remember, we assigned the **`.addBtn`** class to this button in the HTML code, as follows:

```
.addBtn {
    width: 25%;
    padding: 10px;
    background: #d9d9d9;
    color: #555;
    font-size: 16px;
    cursor: pointer;
    border-radius: 5px;
    transition: 0.3s;
}
```

The **`width: 25%;`** makes the button 25% of the container's width, ensuring it fits beside the input field. The **`padding: 10px;`** adds internal spacing, making the button easier to click and more visually balanced. The **`background: #d9d9d9;`** sets a light gray background color. The **`color: #555;`** colors the button text in dark gray (#555) so it shows up clearly. The **`font-size: 16px;`** ensures the button text is clear and readable. The **`cursor: pointer;`** changes the cursor to a hand icon (pointer) when hovering the mouse pointer over the button. The **`border-radius: 5px;`** gives the button rounded corners, ensuring a modern and smooth design.

Styling the task list

Next, we need to style the task list. The task list is an unordered list as we defined in HTML:

```
ul {
    margin: 0;
    padding: 0;
    width: 100%;
    max-width: 400px;
}
```

The ul selector applies styles to the unordered list that holds all the tasks. The **`margin: 0;`** and **`padding: 0;`** removes the default browser spacing for lists. The **`width: 100%;`** makes the task list span the full width of its container, while the **`max-width: 400px;`** prevents it from becoming too wide, maintaining a compact and user-friendly layout.

Styling the individual task items

The ul li selector styles each task item inside the list as follows:

```
ul li {
    list-style: none;
    padding: 12px 8px;
    background: #fff;
    font-size: 18px;
    border-radius: 5px;
    margin-bottom: 8px;
    box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);
    display: flex;
    justify-content: space-between;
    align-items: center;
    cursor: pointer;
    position: relative;
}
```

The **`list-style: none;`** removes default bullet points from the list items. The **`padding: 12px 8px;`** provides spacing inside each task item. The **`background: #fff;`** sets a white background color.

To enhance readability, **`font-size: 18px;`** increases text size. The **`border-radius: 5px;`** rounds the corners of each task item, giving it a modern look. The **`margin-bottom: 8px;`** creates spacing between task

items, preventing them from appearing cluttered. The **box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);** adds a slight shadow beneath each task item, making them stand out against the background.

For layout control, **display: flex;** is used to align elements inside the task item in a row. The **justify-content: space-between;** ensures the task text is on the left and buttons are on the right. The **align-items: center;** vertically aligns all content within the task item. The cursor: pointer changes the cursor to indicate interactivity.

Styling completed tasks

We need a way to mark tasks as complete. To do this, we will change the background color of the task to green. When the user clicks the **Done** button on a task, the JavaScript function adds the class **checked** to that task's **<li>** element, as follows:

```
ul li.checked {
    background: #2ecc71;
}
```

To further emphasize the completion, **text-decoration: line-through;** applies a strike-through effect to the task text. The color: white; ensures the text is still readable against the green background, as follows:

```
ul li.checked .task-text {
    text-decoration: line-through;
    color: white;
}
```

Button styles

The button selector applies general styling to all buttons in the app. **border: none;** removes default button borders, making them look cleaner. The **padding: 6px 12px;** provides internal spacing inside the button. The **cursor: pointer;** changes the cursor when hovering over a button. The **border-radius: 5px;** rounds the corners slightly. The **font-size: 14px;** makes the text bigger inside the button. The **transition: 0.3s;** adds a smooth color change effect when buttons are hovered over, as follows:

```
button {
```

```
    border: none;
    padding: 6px 12px;
    cursor: pointer;
    border-radius: 5px;
    font-size: 14px;
    transition: 0.3s;
    color: white;
}
```

Next, we need to style the remove. The **`.remove`** button is styled with a red color to indicate that it deletes a task as follows:

```
.remove {
    background: #f44336;
}
```

We can do the same for the **`done`** button. This button is styled with a blue background, as follows:

```
.done {
    background: #007bff;
}
```

Once we have styled everything, we will end up with something like the following:

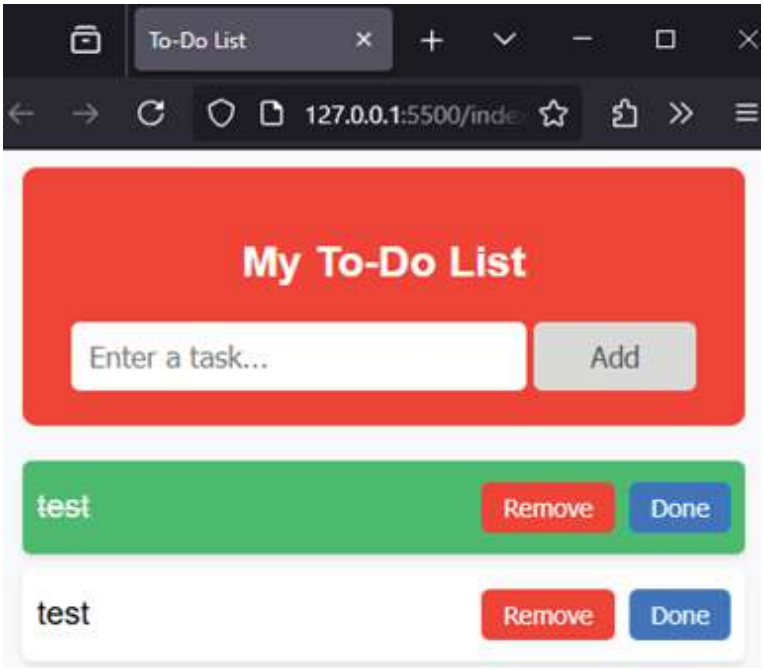


***Figure 13.4**: Styled app*

JavaScript

Now, we need to add some functionality to the app. We can do this using JavaScript. The functionality of the app can be broken down as follows:

- Loading and retrieving tasks:
  - **`loadTasks()` `loads`** and displays saved tasks from **`localStorage`**.
  - **`getStoredTasks()`** retrieves tasks from **`localStorage`** as an array.
- Adding a new task:
  - **`newElement()`** handles user input and adds a new task to the list.
  - **`createTaskElement(text, completed)`** creates a new task item (**`<li>`**) with buttons.
- Managing task status:
  - **`toggleTask(li)`** marks a task as completed or uncompleted and updates **`localStorage`**.
- Deleting a task:

- `deleteTask(li)` removes a task from both the UI and `localStorage`.

- Saving and updating tasks in local storage:
  - `saveTasks(tasks)` saves the updated task list to `localStorage`.
- Handling user interactions:
  - `taskList.addEventListener("click", function (e) {...})` uses event delegation to detect clicks on **Done** and **Remove** buttons.

Initializing the app

First, we need to create an event listener that detects when the page has fully loaded so we can retrieve and display previously saved tasks, as follows:

```
document.addEventListener("DOMContentLoaded",
loadTasks);
```

This line ensures that the `loadTasks()` function runs when the page is fully loaded. The "`DOMContentLoaded`" event is triggered when the HTML is completely parsed, ensuring all elements are available before any JavaScript runs.

We also need to select the list element from the HTML, so we can add tasks to it, as follows:

```
const taskList = document.getElementById("myUL");
```

Loading and retrieving tasks

Once we have done that, we need to check to see if there are any saved tasks in local storage, as follows:

```
function loadTasks() {
    const tasks = getStoredTasks();
```

This function is called from the event listener we added earlier. The `loadTasks()` function is responsible for retrieving previously saved tasks from local storage and displaying them in the task list when the page loads. It begins by calling `getStoredTasks()`, which is a helper function as follows:

```
function getStoredTasks() {
```

```
    return JSON.parse(localStorage.getItem("tasks")) ||
[];
}
```

This fetches and parses the saved tasks from **`localStorage`**. The retrieved tasks are stored in the tasks array. If no tasks exist, an empty array is returned to prevent errors.

The **`loadTasks()`** function then loops through each task in the array using the **`.forEach()`** method. For each task, it calls **`createTaskElement(task.text, task.completed)`**, defined as follows, which generates a new **`<li>`** element containing the task text and its completion status. The newly created task element is then appended to **`taskList`**, ensuring that all tasks from local storage are visually rendered in the list.

```
tasks.forEach(task => taskList.appendChild(createTaskElement(task.text, task.completed)));
```

This will populate the list with the saved tasks, as shown in the following figure:

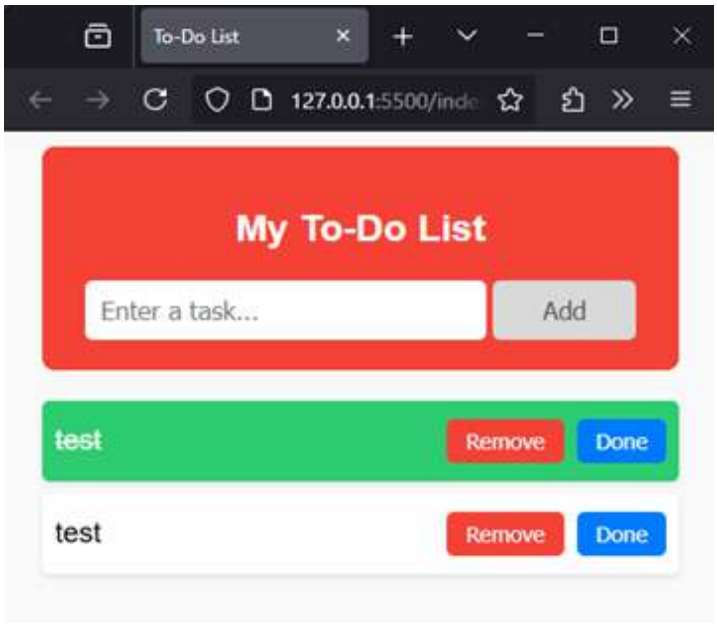


***Figure 13.5**: App with saved tasks added at the bottom*

Adding a new task

We need to define **createTaskElement()** so we can add the loaded elements from **localStorage** to the list as a **new ** element, as follows:

```
function createTaskElement(text, completed) {
    const li = document.createElement("li");
    li.innerHTML = `
        <span class="task-text">${text}</span>
        <div class="button-container">
            <button class="done">${completed ? "Undo" :
"Done"}</button>
            <button class="remove">Remove</button>
        </div>
    `;
    if (completed) li.classList.add("checked");
    return li;
}
```

This function generates an individual task item (**<li>**) in the to-do list. It takes two parameters: text, which represents the task description, and completed, a Boolean indicating whether the task has been marked as done. Inside the function, a new **<li>** element is created using **document.createElement("li")**. The **innerHTML** property of this element is then set to a template string containing a **** element to display the task text and a ** ** container for the action buttons. The **Done** button is conditionally labeled as either **Done** or **Undo**, depending on whether the task is already marked as completed. If the completed parameter is true, the function applies the **checked** class to the **<li>** element, visually differentiating completed tasks from active ones. The function then returns the fully constructed **<li>** element, which can then be appended to the task list in the DOM.

Once we have done that, we need to select the HTML elements that our script will interact with. The input field and the task list. Remember, we added an input text field with the ID **"myInput"**. We can select this element using the **document.getElementById():** We can then assign these to a constant in JavaScript, as follows:

```
const input = document.getElementById("myInput");
```

Now, we need a function that will add a task when the user clicks the **Add** button, as follows:

```
function newElement() {
    const taskText = input.value.trim();
    if (!taskText) {
        alert("You must write something!");
        return;
    }
    const tasks = getStoredTasks();
    if (tasks.some(task => task.text === taskText))
        return; // Prevent duplicates
    taskList.appendChild(createTaskElement(taskText,
false));
    saveTasks([...tasks, { text: taskText, completed:
false }]);
    input.value = "";
}
```

This function adds a new task to the to-do list when the user enters text and clicks the **Add** button. It starts by retrieving the text entered in the input field and using **`.trim()`** to remove any extra spaces. If the input is empty, an alert is displayed, and the function exits early to prevent adding blank tasks. Next, it calls **`getStoredTasks()`**, which we defined earlier, to retrieve the current list of saved tasks from **`localStorage`**. Before adding a new task, it checks whether a task with the same text already exists using **`.some()`**, which iterates through the existing tasks and prevents duplicates from being added. If the task does not exist, it proceeds to create a new task element by calling **`createTaskElement(taskText, false)`**, which we defined earlier, where false indicates that the task is not yet completed. The new task is then appended to **`taskList`**, making it visible on the HTML page. Finally, the updated list of tasks is stored in **`localStorage`** using the **`saveTasks()`** function defined in the next section.

Managing task status

We also need a function that will mark a task as complete or not when the user clicks on the done/undo button, as follows:

```
function toggleTask(li) {
    const text = li.querySelector(".task-
text").textContent;
    li.classList.toggle("checked");
    // Find the Done/Undo button and update its text
    const doneButton = li.querySelector(".done");
    doneButton.textContent =
li.classList.contains("checked") ? "Undo" : "Done";
    let tasks = getStoredTasks();
    tasks = tasks.map(task =>
        task.text === text ? {
           text, completed:
li.classList.contains("checked")
        } : task
    );
    saveTasks(tasks);
}
```

This function takes a **`<li>`** element (representing a task) as an argument and retrieves the task text from the **`<span>`** element inside the **`<li>`**. The function then toggles the **`checked`** class on the **`<li>`** element, visually indicating whether the task is marked as completed.

Next, it selects the **Done** button inside the **`<li>`** and updates its text dynamically. If the **`checked`** class is present, meaning the task is completed, the button text changes to **Undo**, allowing the user to revert the task to an incomplete state. Otherwise, the button text remains **Done**.

After updating the UI, the function ensures that the task's status is correctly updated in local storage. It retrieves the current list of tasks using **`getStoredTasks()`**, then updates the completed property of the corresponding task using **`.map()`**. The function checks if the task text matches the stored text and updates its completed status based on whether the **`checked`** class is present. Finally, the modified task list is saved back to **`localStorage`** using **`saveTasks()`**, ensuring that the completion status

persists even after the page is refreshed.

Deleting a task

Next, we need a function to delete a task when the user clicks the remove button, as follows:

```
function deleteTask(li) {
    const text = li.querySelector(".task-
text").textContent;
    li.remove();
    let tasks = getStoredTasks();
    tasks = tasks.filter(task => task.text !== text);
    saveTasks(tasks);
}
```

Here, this function takes a **<li>** element (representing the task item) as an argument. First, it retrieves the task text from the **** element inside the **<li>** using **querySelector(".task-text").textConten**t, ensuring that it captures the exact text of the task to identify it in storage.

Next, it calls **li.remove()**, which removes the task element from the HTML document, updating the UI so that the task disappears from the list, then updates the stored tasks. It retrieves the current task list from **localStorage** using **getStoredTasks()** and then filters out the task that matches the retrieved text using **.filter()**. This creates a new array that excludes the deleted task while keeping all others intact.

The updated task list is saved back to **localStorage** using **saveTasks(tasks)**, ensuring that the deletion is permanent and that the task does not reappear when the page is refreshed. This function synchronizes the UI and stored data, ensuring that tasks are removed from both the screen and **localStorage**.

Saving and updating tasks in local storage

The **saveTasks()** function is responsible for storing the updated list of tasks in **localStorage** to ensure that tasks persist even after the page is refreshed. It takes an array of tasks (tasks) as an argument and converts it into a JSON string using **JSON.stringify(tasks)**, since **localStorage** can only store data as strings. The function then saves this

string in **localStorage** under the key *tasks* using **localStorage.setItem("tasks", ...)**. This ensures that the current state of the task list, including both completed and uncompleted tasks, is stored and can be retrieved later using **getStoredTasks()**:

```
function saveTasks(tasks) {
    localStorage.setItem("tasks",
JSON.stringify(tasks));
}
```

Handling user interactions

Finally, we need another event listener to deal with when the user clicks the **Done** button and the **Remove** button, as follows:

```
taskList.addEventListener("click", function (e) {
    if (e.target.classList.contains("done")) {

toggleTask(e.target.parentElement.parentElement);
    } else if (e.target.classList.contains("remove")) {

deleteTask(e.target.parentElement.parentElement);
    }
});
```

This event listener is attached to **taskList** and listens for click events on any of its child elements. Instead of adding separate event listeners to each individual **Done** and **Remove** button, this function uses event delegation to handle clicks dynamically. This is useful because new tasks are continuously being added to the list, and this approach allows us to manage all buttons efficiently.

When a user clicks anywhere inside the **taskList**, the function checks the target of the event (**e.target**). The **e.target.classList.contains("done")** condition checks whether the clicked element has the "**done**" class, which means the **Done** button was clicked. If so, the function calls **toggleTask()**, passing **e.target.parentElement.parentElement** as an argument. This ensures that the function receives the correct **<li>** element (which represents the task) by moving up the DOM tree from the button to its parent

`<li>`.

Similarly, if the clicked element has the `"remove"` class (`e.target.classList.contains("remove")`), the function calls `deleteTask()` with the same method of selecting the correct `<li>` element. This ensures that the corresponding task is removed when the **Remove** button is clicked, as shown in the following figure:

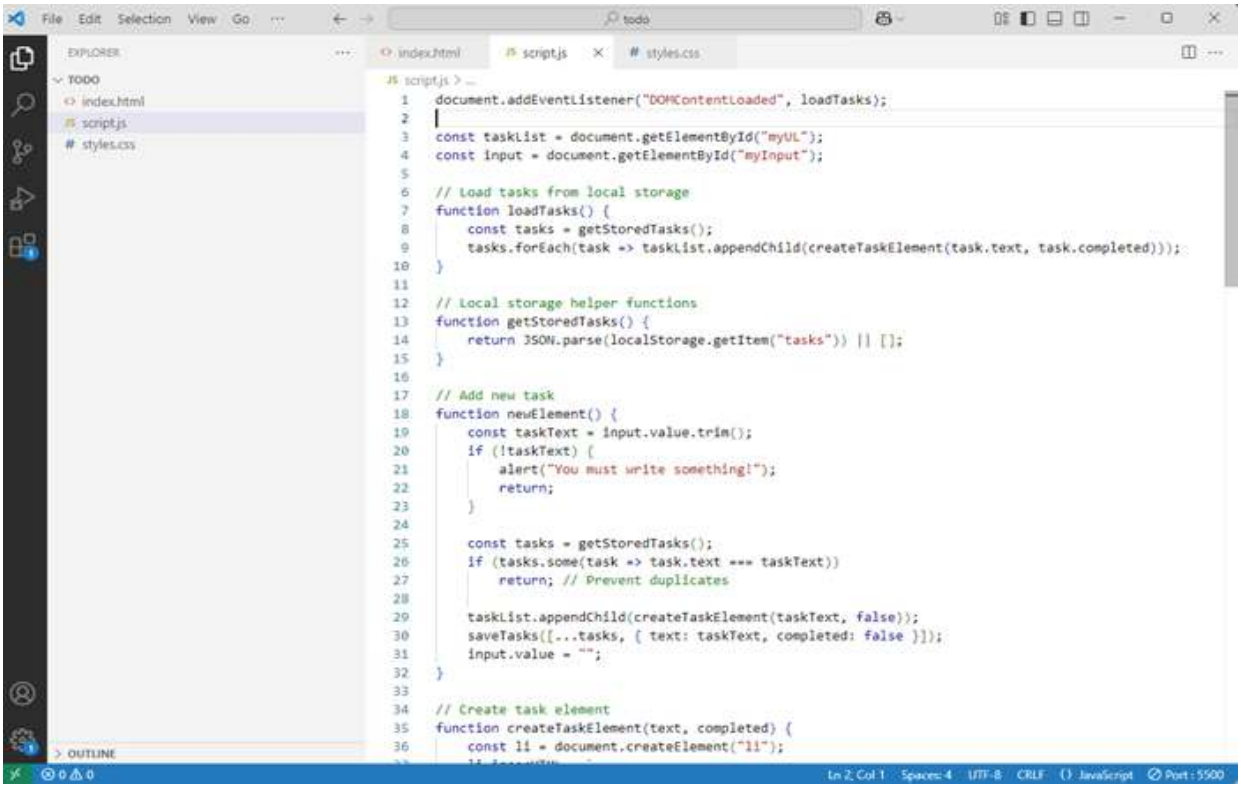


***Figure 13.6**: Clicking remove button*

Testing the website

Once the To-Do List app is complete, it is important to thoroughly test it to ensure all features work correctly across different browsers and devices.

Functional testing

First, we will do some functional texting. This means going through the app's functions and testing each one, using the following steps:

1. Add task:

   a. Type a task and click **Add**.
   b. Check that it appears in the list.
   c. Ensure the input field clears afterward.
   d. Prevents adding empty or duplicate tasks.
2. Mark the task as completed:
   a. Click the **Done** button.
   b. Task should get a green background and a strikethrough.
   c. Button label should change to **Undo**.
3. Undo completed task:
   a. Click the **Undo** button.
   b. Task should revert to normal appearance.
4. Remove task:
   a. Click the **Remove** button.
   b. The task should be deleted from the list.
5. Local storage persistence:
   a. Add several tasks.
   b. Refresh the browser.
   c. Confirm tasks reappear with correct states (completed or not).

User interface testing

Next, we need to check all visual elements to make sure they appear correctly according to the original specification. We need to check the following:

- Header is visible and properly centered.
- The input field and Add button are aligned.
- Tasks are displayed in a list with proper spacing.
- **Done** and **Remove** buttons appear for each task.
- Colors and fonts are consistent and readable.
- Spelling.

Responsive design testing

Since this is a responsive design, meaning it adapts to different screen sizes,

we need to ensure the app works well on different screen sizes, as follows:

1. Load the app on mobile, tablet, laptop, and desktop devices.
2. Resize the browser window manually on laptops and desktops.
3. Check tasks, input field, and buttons should realign correctly.
4. Check no overflow or broken layout on small screens.
5. Min-width (250px) is enforced and prevents unusable narrow views.

Cross-browser compatibility

Not everyone uses the same web browser, so we need to test the app in multiple browsers, as follows:

- Google Chrome
- Mozilla Firefox
- Microsoft Edge
- Safari (especially on macOS/iOS)

Introduction to debugging

Debugging is the process of identifying, analyzing, and fixing problems (bugs) in your code. Bugs can cause your app to behave incorrectly, like buttons that do not respond, elements that do not display, or errors that break your script entirely. Debugging is not just about fixing code; it is about understanding what the code is doing at runtime and comparing that to what it should be doing.

In web development, bugs often come from small mistakes like typos in HTML element IDs, logic errors in JavaScript, or styles not applying correctly in CSS. When things do not work as expected, the browser does not just give up; it often leaves clues. These clues show up in tools like the Console, Elements, and Sources tabs inside the **developer tools** (**DevTools**) of your web browser.

Debugging is a skill you develop over time, but learning to read error messages, trace your code, and investigate the DOM will quickly turn you into a more confident and capable developer. Think of it as being a detective: the browser gives you clues, and your job is to follow the trail, ask the right questions, and fix the root cause, not just the symptom.

Let us say we found a small issue with our To-Do List app, the **Add** button is not working. We need to figure out what is happening and where, so we can fix it, as follows:

- We typed a task like **`Buy groceries`** into the input field.
- We click the **Add** button, and nothing happens. No task is added.

For the sake of this demo, an error in the HTML code has been deliberately introduced, as follows:

```
<input type="text" id="myInpoot" placeholder="Enter a task...">
```

Try the app in the browser.

Type a task like **`Buy Groceries`**, then click the **Add** button. Nothing happens. This is the clue that something is broken.

In the web browser, e.g., Chrome, press *F12* to open the developer tools. Click on the **Console** tab. Refer to *Figure 13.7*:

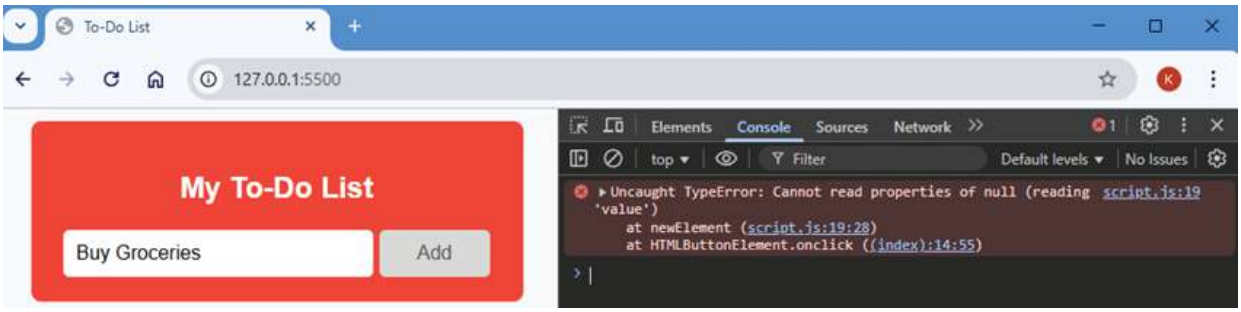


***Figure 13.7**: Console tab*

JavaScript is trying to read **`.value`** on a null variable, most likely because it is trying to access an input field that does not exist (or is not found yet). Look at the following error:



***Figure 13.8**: Error*

This points to line 19 in **`script.js`**. Click the **script.js:19** link in the error message. This jumps to that line in the **Sources** tab, as shown in the following figure:

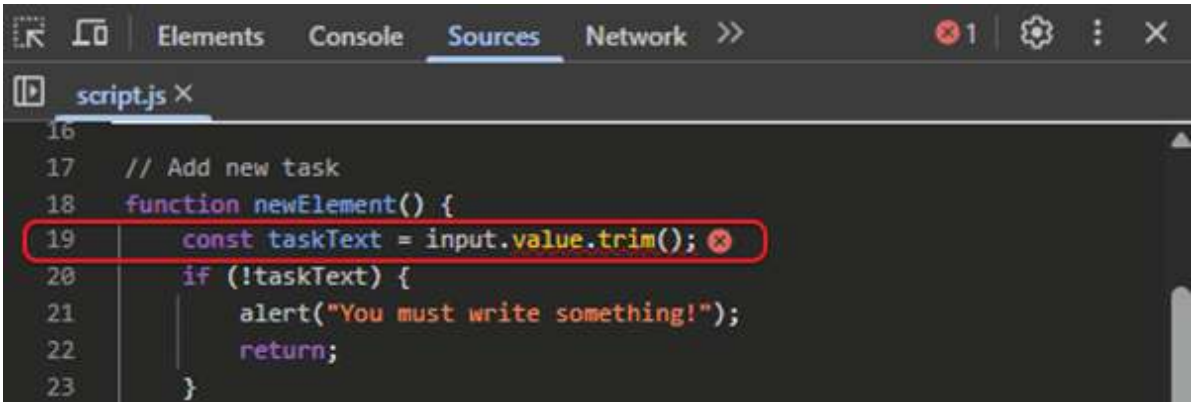


***Figure 13.9**: Sources tab*

Let us check if the element exists on the page. Click on the **Elements** tab. There it is. Typo! The ID of the input field is **`myInpoot`** instead of **`myInput`** in the HTML file, as shown in the following figure:

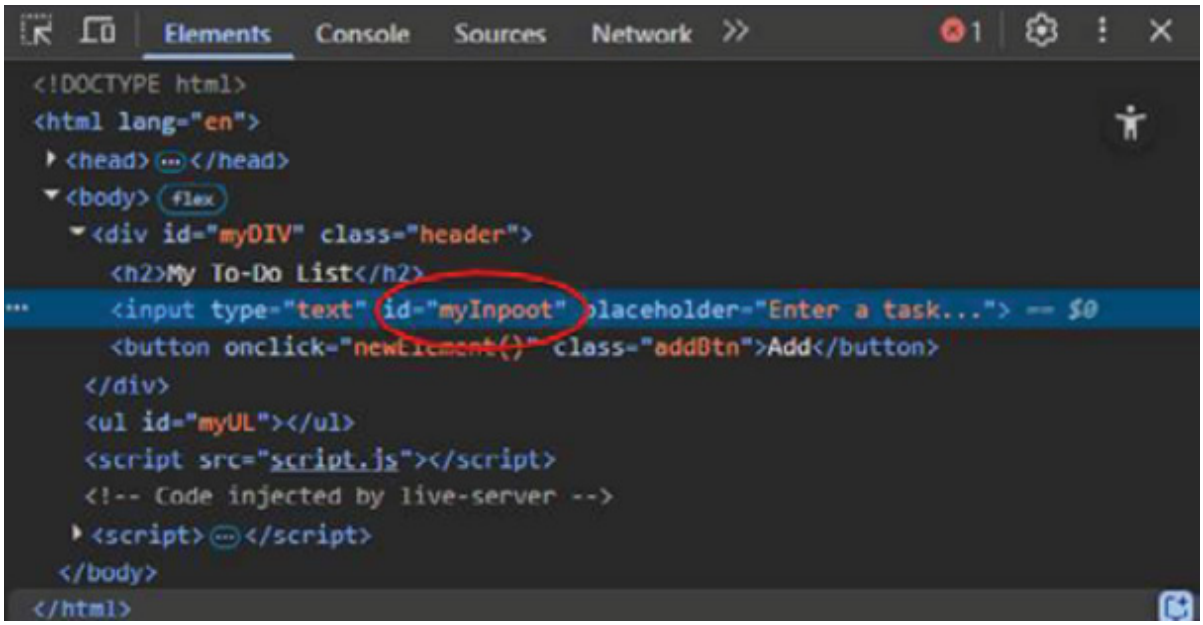


***Figure 13.10**: Typo in the elements tab*

Conclusion

In this chapter, you learned how to build a simple, responsive To-Do List application from scratch using HTML, CSS, and JavaScript. Starting with the HTML structure, you implemented a clean user interface and then progressively enhanced it with responsive CSS layouts and dynamic interactivity using JavaScript. You explored key web development concepts, including DOM manipulation, event handling, local storage, and responsive design techniques like flexbox and media queries. Most importantly, you practiced debugging using browser developer tools, an essential skill for any web developer.

By breaking down each component, structure, styling, functionality, testing, and debugging, you gained a clear, end-to-end understanding of how a modern web application is planned, built, and polished for real-world use. Whether you are developing solo projects or preparing for more complex applications, the skills you have acquired here will form a solid foundation for front-end development and beyond.

Points to remember

- Use semantic HTML to create a clean and structured layout.
- The <meta viewport> tag is essential for responsive design on mobile devices.
- Organize your project with separate files: index.html, styles.css, and script.js.
- Use flexbox in CSS for flexible and responsive layout design.
- The box-sizing: border-box rule simplifies sizing and layout consistency.
- Apply min-width to prevent layout collapse on very small screens.
- Use class selectors and element IDs consistently across HTML, CSS, and JavaScript.
- JavaScript interacts with HTML elements using document.getElementById() and event listeners.
- Use event delegation to handle dynamic elements (like buttons inside

list items).

- Store data persistently using localStorage.setItem() and localStorage.getItem().
- Parse and stringify data when saving and retrieving from local storage.
- Prevent adding blank or duplicate tasks by validating user input.
- Toggle task completion state by adding/removing CSS classes.
- Separate logic into reusable functions: createTaskElement(), saveTasks(), deleteTask(), etc.
- Use meaningful class names for buttons (.done, .remove) to simplify styling and event handling.
- Test your app for functionality, responsiveness, and cross-browser compatibility.
- Use browser Developer Tools (Console, Elements, Sources) for debugging.
- Understand and fix common JavaScript errors like null references or typos.
- Responsive design ensures the app works on desktops, tablets, and phones.
- Consistent testing and debugging are critical for building reliable web apps.

Exercises

In this project, you will build a simple, responsive web-based notes application. This app will allow users to create, edit, and manage multiple plain-text notes. All data should be stored locally in the browser using localStorage. The app will need to do the following:

1. **HTML layout**: **Design the following components**:
   a. **Header**: Title (e.g., *My Notes*).
   b. **Sidebar**: List of saved notes (by title).
   c. **Main pane**:
      - Title input.
      - Note content area (textarea).

- Save button.
- Delete button.

2. **CSS styling**: **Style the app with a modern, minimal look**:
   a. Wrap everything in a container that uses flexbox or CSS grid.
   b. Sidebar and editor pane side by side on desktop.
   c. Stack vertically on mobile.
   d. Use readable fonts, spacing, and consistent colors.
   e. Highlight the selected note in the sidebar.
   f. Show feedback when saving (e.g., button flash or message).
3. **JavaScript functionality**: **Implement the following features**:
   a. **Create a note**:
      i. User types in a title and content.
      ii. Clicks the *Save* button.
      iii. Note is saved to localStorage as an object: { id, title, content, timestamp }.
      iv. Note appears in the sidebar.
   b. **Edit a note**:
      i. Click a note in the sidebar.
      ii. Title and content appear in the editor.
      iii. Changes auto-save or update with the *Save* button.
   c. **Delete a note**:
      i. Clicking *Delete* removes it from the sidebar, editor, and localStorage.
   d. **Persistent storage**:
      i. On page load, load all notes from localStorage.
      ii. Use JSON.stringify() and JSON.parse() to manage saved data.
   e. **Responsive design**: **On screens < 768px wide**:
      i. Sidebar becomes collapsible (or stacked).
      ii. Make use of @media queries and flexbox layout shifts.
   f. **Deliverables**: **A working notes-app folder with**:
      i. index.html

ii. styles.css
iii. script.js

Index

Symbols

<b> 40
<body> 27
<h1> 37
<head> 27
<html> 27
<i> 41
<label> 71
<p> 38
<title> 27

A

Angular 155
Animations 100
Animations, configuring 101, 102
Animations, properties 100
Arrays 174
Arrays, elements
  Length Property 176
  Remove/Adding 175
  Splice/Slice 177
  Values Assigning 175
  Zero-Based Indexing 174
Audio Tags 61
Audio Tags, configuring 61

B

Booleans 166
Booleans, configuring 166
Box Model 103
Box Model, configuring 103
Browser Developer 32
Browser Engine 3, 4
Browser Rendering 4
Browser Rendering, steps 4-6
Built-In Functions 139
Built-In Functions, types
  Calc 139

Max 140
Min 140

C

Class Selector 82
Code Editor 13
Code Editor, ensuring 13, 14
Colors 84
Colors, ensuring 85, 86
Conditional Statements 168
Conditional Statements, types
If 168
If-Else 168
Switch 169
Cross-Browser Compatibility 217
CSS 7
CSS, architecture 80
CSS, elements 203-207
CSS, objectives
Float 121
Grid Layout 117
Layout Model 113
Measurement Units 122
Positioning 110
CSS, properties
Colors 84
Fonts 89
Hexadecimal Color 87
CSS Selectors 81
CSS Selectors, types
Class 82
Type 82
Universal 82
CSS, styles
External 83
Inline 83
Internal 83
CSS, term
Gradients 96
Pseudo-Classes 97
Pseudo-Elements 93
Rounded Corners 95
Shadows 94
Text Effects 92

D

Debugging 217

Debugging, ensuring 217-219
Display Property 104
Display Property, illustrating 104
Document Object Model (DOM) 186
DOM, elements 188
DOM, ensuring 186, 187

E

Error Handling 179
Error Handling, illustrating 179, 180
Event Handling 193
Event Handling, types 194
Event Listeners 194, 195
External Styles 83

F

Flow Control 167
Flow Control, terms
  Arrays 174
  Conditional Statements 168
  Function 171
  Loops 169
  Object-Oriented Programming (OOP) 178
Fonts 89
Fonts, steps 89-91
Forms 66
Forms, configuring 66-68
Forms, steps 68, 69
Forms, terms
  Data Processing 74
  Form Validation 75
  Group Elements 71
  Input Types 69
  Labels Adding 71
Form Validation 75, 76
Function 171
Functional Testing 215
Function, parameters
  Default 172
  Required 172
  Rest 172
Function, syntax 171
Function, terms
  Function Scope 173
  Hoisting 174
  Return Statements 172

G

Gradients 96
Grid Layout 117
Grid Layout, illustrating 117-120

H

Hexadecimal Color 87
Hexadecimal Color, configuring 87, 88
HTML 6, 7
HTML, architecture 26, 27
HTML Element, manipulating 189, 190
HTML, elements
  Hyperlinks 49
  Nesting Lists 53
  Ordered Lists 51
  Tables 47
  Unordered Lists 52
HTML File 27
HTML File, steps 27-30
HTML Form 146
HTML Form, illustrating 146, 147
HTML Heading 30
HTML Heading, integrating 30, 31
HTML, objectives
  Audio 61
  Image 58
  Videos 60
HTML Tags 36, 37
HTML Tags, types
  Background Color 44
  Bold 39
  Color 42
  Fonts 43
  Headings 37, 38
  Italic 41
  Paragraphs 38, 39
Hyperlinks 49
Hyperlinks, ensuring 49-51

I

IDE 12
Images Tags 58
Images Tags, configuring 58, 59
Inline Styles 83
Input Types 69, 70
Input Types, configuring 70

Internal Styles 83

J

JavaScript 8
JavaScript, architecture 154
JavaScript, configuring 155, 156
JavaScript Data, types
  Booleans 166
  Numbers 161
  Strings 162
JavaScript, elements 209-213
JavaScript, features 155
JavaScript Mathematical, functions 161
JavaScript, operations 161
JavaScript, properties 192

L

Layout Model 113
Layout Model, configuring 114-116
Loops 169, 170
Loops, types
  Do-While 171
  For 170
  While 170

M

Media Queries 129
Media Queries, architecture 129
Media Queries, illustrating 130-133
Mobile-First Designs 134
Mobile-First Designs, steps 135, 136

N

Nesting Lists 53
Nesting Lists, configuring 53
Node.js 154
Numbers 161

O

Object-Oriented Programming (OOP) 178
OOP, ensuring 178, 179
Ordered Lists 51
Ordered Lists, configuring 52

P

Positioning 110
Positioning, ensuring 110-112
Pseudo-Classes 97, 98
Pseudo-Elements 93

R

React 155
Reserved Keywords 156
Reserved Keywords, ensuring 156-159
Responsive Design 128
Responsive Design Testing 216
Responsive Grid Layout 129
Responsive Websites 136
Responsive Websites, features
  Cross-Browser Compatibility 217
  Debugging 217
  Functional Testing 215
  Responsive Design Testing 216
  User Interface Testing 216
Responsive Websites, illustrating 137-139
Rounded Corners 95

S

Sass, configuring 105
Shadows 94
Strings 162
Strings, configuring 162
Strings, illustrating 164, 165
Submit Button 149
Submit Button, illustrating 149, 150
Syntactically Awesome Stylesheets (Sass) 105

T

Tables 47
Tables, ensuring 47, 48
Text Effects 92
Transitions 99
Type Selector 82

U

Universal Selector 82
Unordered Lists 52
Unordered Lists, configuring 52, 53
User Interface 200

User Interface Testing 216

V

Variables 160
Videos Tags 60
Videos Tags, configuring 60
VS Code 15
VS Code, implementing 21
VS Code, platform
  macOS 19, 20
  Window 15-17
VS Code, steps 22
VS Code, terms
  CSS 203
  HTML 202
  JavaScript 208

W

Web Applications 6
Web Applications, technologies
  CSS 7
  HTML 6
  JavaScript 8
Web Browsers 3
Web Development 2
Web Development, domains 2, 3