

Rajmond JÁNÓ · Adelina Ioana ILIEȘ

WEB TECHNOLOGIES

PART ONE

HTML

HTML

CSS

JS



U.T.PRESS

Cluj-Napoca, 2026

ISBN 978-606-737-863-4

Rajmond JÁNÓ

Adelina Ioana ILIEȘ

WEB TECHNOLOGIES

Part one

HTML



U.T. PRESS
Cluj-Napoca, 2026
ISBN 978-606-737-863-4



Editura U.T.PRESS
Str. Observatorului nr. 34
400775 Cluj-Napoca
Tel.: 0264-401.999
e-mail: utpress@biblio.utcluj.ro
www.utcluj.ro/editura

Recenzia: Ș.l.dr.ing. Cosmina Corcheș

Ș.l.dr.ing. Ștefania Ramona Benea

Pregătire format electronic on-line: Gabriela Groza

Copyright © 2026 Editura U.T.PRESS

Reproducerea integrală sau parțială a textului sau ilustrațiilor din această carte este posibilă numai cu acordul prealabil scris al editurii U.T.PRESS.

ISBN 978-606-737-862-7

ISBN 978-606-737-863-4 vol. 1

Contents

- Contents 1
- Chapter 1. Introduction to Web Technologies 4**
 - What are Web Technologies? 4
 - 1.2. The Role of HTML in the Web 6
 - 1.3. Evolution of HTML: From HTML 1.0 to HTML5 9
 - 1.4. Overview of Modern Web Development 12
 - Summary and Next Steps..... 15
- Chapter 2. Getting Started with HTML 17**
 - 2.1. What Is HTML? 17
 - 2.2. Basic Syntax and Structure 21
 - 2.3. Creating Your First HTML Document 25
 - 2.4. Understanding the DOCTYPE Declaration..... 29
- Chapter 3. The Anatomy of an HTML Document 32**
 - 3.1. The <html> Element 33
 - 3.2. The <head> Section: Metadata, Title, and Links 35
 - 3.3. The <body> Section: Content and Structure 39
 - 3.4. Comments and Code Organization 43
- Chapter 4. Text and Structural Elements..... 48**
 - 4.1. Headings, Paragraphs, and Block Elements 48
 - 4.2. Inline Elements and Formatting Text..... 56
 - 4.3. Lists: Ordered, Unordered, and Definition Lists 61
 - 4.4. Divisions and Spans for Grouping Content 65
- Chapter 5. Hyperlinks and Navigation 69**
 - 5.1. Creating Hyperlinks with the <a> Tag 69
 - 5.2. Absolute vs. Relative URLs 72
 - 5.3. Linking to External Resources and Anchors..... 74

5.4. Navigation Menus and Site Structure	77
Chapter 6. Working with Images and Multimedia	81
6.1. Embedding Images Using	81
6.2. Image Attributes: Alt Text, Titles, and Sizing.....	84
6.3. Incorporating Audio and Video Elements	86
6.4. Accessibility Considerations for Multimedia.....	89
Chapter 7. Tables and Data Presentation	93
7.1. Building Tables with <table>, <tr>, <th>, and <td>	93
7.2. Enhancing Tables with Captions, Headers, and Footers	96
7.3. Styling Tables for Readability	100
7.4. Best Practices for Displaying Tabular Data	103
Chapter 8. HTML Forms and User Interaction	106
8.1. Creating Forms with the <form> Tag	106
8.2. Form Elements: Input, Textarea, Select, and Button	108
8.3. Form Attributes and Data Submission	112
8.4. Basic Client-Side Validation Techniques.....	115
Chapter 9. Semantic HTML and Accessibility	119
9.1. The Importance of Semantic Markup	119
9.2. Key Semantic Elements (<header>, <nav>, <article>, <footer>, and More)	121
9.3. Enhancing Accessibility with ARIA Roles	125
9.4. SEO Benefits of Semantic HTML.....	129
Chapter 10. HTML5 and Its New Features.....	133
10.1. Overview of HTML5 Innovations	133
10.2. New Structural Elements	135
10.3. Multimedia and Graphics with HTML5.....	137
10.4. APIs and Integration with Other Web Technologies	140
Chapter 11. Best Practices and Common Pitfalls	143

11.1. Writing Clean and Maintainable HTML Code	143
11.2. Cross-Browser Compatibility Issues.....	147
11.3. Debugging and Validating HTML.....	149
11.4. Future Trends in HTML Development.....	152
Chapter 12. Hands-On Project: Building a Simple Web Page.....	155
12.1. Planning Your Web Page Layout	155
12.2. Implementing the Structure in HTML.....	157
12.3. Incorporating Multimedia and Forms	160
12.4. Testing and Iterating on Your Design	163
Conclusion	165
A Recap of Key Concepts.....	165
Next Steps in Web Development	166
Additional Resources	166
Appendices	167
A. HTML Tag Reference	167
B. Glossary of Terms	171
C. Further Reading and Online Resources	173

Chapter 1. Introduction to Web Technologies

What are Web Technologies?

Web technologies refer to the broad spectrum of tools, languages, protocols, and frameworks that are used to create, manage, and deliver content over the Internet. They form the backbone of the modern web, enabling the development of interactive, dynamic, and user-friendly websites and web applications. In this subchapter, we explore the fundamental components that constitute web technologies and explain how they work together to power the online experience.

Building Blocks

At the most basic level, web technologies are built around three core components:

- **HTML (Hypertext Markup Language)**

HTML is the standard markup language used to structure content on the web. It provides the skeleton of a webpage, defining elements such as headings, paragraphs, images, links, and more. As the foundational language of the web, HTML ensures that content is accessible and organized for both users and search engines.

- **CSS (Cascading Style Sheets)**

CSS is responsible for the visual presentation of a webpage. While HTML lays out the structure, CSS brings style to the content—controlling layout, colors, fonts, and overall design aesthetics. This separation of structure and style not only improves maintainability but also allows designers and developers to create responsive designs that adapt to various devices and screen sizes.

- **JavaScript**

JavaScript adds interactivity and dynamic behavior to websites. It enables the creation of interactive elements such as sliders, forms, animations, and real-time

updates without requiring a page reload. JavaScript, along with its many frameworks and libraries, has revolutionized the way users interact with web content, making modern websites more engaging and responsive.

Beyond the Basics

In addition to these core technologies, the modern web relies on a variety of supplementary tools and protocols that enhance functionality, performance, and security:

- **Web Servers and Protocols**

Technologies like HTTP/HTTPS, RESTful APIs, and WebSockets manage communication between clients (browsers) and servers. Web servers such as Apache or Nginx handle requests and deliver the appropriate content to users, while secure protocols ensure that data is transmitted safely.

- **Content Management Systems (CMS)**

Platforms like WordPress, Joomla, and Drupal simplify the process of creating and managing web content, allowing users without extensive technical knowledge to build and maintain websites.

- **Frameworks and Libraries**

Tools such as React, Angular, and Vue.js for JavaScript, or Bootstrap and Tailwind CSS for styling, provide pre-designed components and best practices that streamline development. These frameworks help developers build complex applications more efficiently while ensuring consistency and scalability.

- **Databases and Backend Technologies**

Behind many web applications is a robust backend that manages data storage and retrieval. Technologies like MySQL, MongoDB, Node.js, Python (with frameworks such as Django or Flask), and Ruby on Rails support server-side logic and data processing, ensuring that dynamic content is delivered smoothly to end users.

The Interconnected Ecosystem

What makes web technologies truly powerful is their interconnectivity. Each component, from HTML's structure to JavaScript's interactivity, works in concert with the others to create a cohesive user experience. The layered architecture of web development allows for

a modular approach, where developers can update, enhance, or replace individual parts without disrupting the entire system.

This interconnected ecosystem also enables rapid innovation. New tools and frameworks emerge regularly, fostering an environment where best practices are continuously refined. As a result, web technologies are not static; they evolve to meet the changing demands of users and the dynamic nature of the internet itself.

The Impact on Society

Web technologies have dramatically transformed how we access information, conduct business, and communicate with one another. From e-commerce and social networking to online education and entertainment, the reach of these technologies is virtually limitless. They have lowered barriers to entry for businesses, empowered individuals with access to a global marketplace, and fundamentally reshaped our social interactions.

Conclusion

In summary, web technologies are a multifaceted suite of tools and languages that create the digital world we interact with every day. By understanding these building blocks and their interactions, one gains insight into both the technical and cultural impact of the web. As we delve deeper into this book, we will explore each of these technologies in greater detail, uncovering how they contribute to the robust, dynamic, and ever-evolving landscape of the internet.

1.2. The Role of HTML in the Web

HTML, or HyperText Markup Language, is the cornerstone of web content. It is the language that transforms raw data into the structured pages we interact with on the internet. In this subchapter, we delve into HTML's pivotal role in the web ecosystem, discussing its functions, evolution, and enduring importance in modern web development.

The Foundation of Web Content

- **Structural Blueprint**

HTML provides the fundamental structure for a webpage. Just as a building requires a solid framework, every website relies on HTML to arrange text, images, and multimedia elements into a coherent and accessible format.

- **Content Organization**

Through the use of various tags and elements (such as headings, paragraphs, lists, and links), HTML organizes content in a way that is both meaningful and navigable for users. This clear structure is essential for effective communication and user experience.

- **Accessibility and Semantics**

By using semantic HTML elements (like `<header>`, `<nav>`, `<article>`, `<section>`, and `<footer>`), developers provide context to both browsers and assistive technologies. This semantic approach not only improves the accessibility of web content for users with disabilities but also aids search engines in understanding and indexing content more accurately.

Evolution and Standardization

- **Early Beginnings**

HTML began as a simple language for creating basic documents with hyperlinks. Over time, its capabilities expanded significantly, mirroring the growing complexity and expectations of the web.

- **HTML4 and XHTML**

With the introduction of HTML4 and later XHTML, standards became more rigorous, emphasizing well-formed code and improved interoperability among browsers. These iterations marked important steps in web evolution, ensuring that websites maintained a consistent look and function across different platforms.

- **HTML5 – A Modern Renaissance**

The development of HTML5 has been a game-changer, introducing new semantic elements, multimedia support, and APIs that allow for richer, more interactive

web experiences. HTML5 not only simplifies the development process but also reduces the need for third-party plugins, enhancing both security and performance.

HTML and Other Web Technologies

- **The Interplay with CSS and JavaScript**

While HTML lays the foundation by structuring content, CSS and JavaScript build on this framework to enhance presentation and interactivity. CSS handles the visual styling, making pages attractive and adaptable across devices, while JavaScript introduces dynamic behaviors that improve user engagement.

- **Supporting Web Standards**

HTML is governed by standards developed by organizations such as the World Wide Web Consortium (W3C) and the Web Hypertext Application Technology Working Group (WHATWG). These standards ensure that HTML remains a reliable and forward-compatible tool, promoting consistency and stability across the rapidly evolving web landscape.

- **Enhancing SEO and Performance**

Well-crafted HTML not only improves the user experience but also plays a crucial role in search engine optimization (SEO). Search engines rely on HTML's structure and semantic elements to index pages effectively, making a website more discoverable. Furthermore, clean HTML contributes to faster load times and a smoother browsing experience, essential factors for retaining modern web users.

HTML in the Broader Context of Web Development

- **Democratizing Web Creation**

HTML's relative simplicity makes it accessible to beginners and professionals alike. Its widespread adoption has democratized web creation, enabling individuals, businesses, and organizations to publish content online without requiring advanced technical skills.

- **A Living Language**

HTML continues to evolve, adapting to the new demands of the digital age. The ongoing development of HTML standards reflects the dynamic nature of the web,

ensuring that it remains a robust and adaptable tool for content creators around the globe.

Conclusion

HTML is more than just a language—it is the very fabric of the web. By providing a structured, accessible, and evolving framework, HTML enables the creation of rich and interactive digital experiences. As the gateway to more advanced technologies like CSS and JavaScript, HTML remains indispensable in both traditional website development and modern web applications. In the chapters that follow, we will explore these technologies further, uncovering how they work together to bring the web to life.

1.3. Evolution of HTML: From HTML 1.0 to HTML5

The story of HTML is one of steady evolution, reflecting the changing needs and capabilities of the web. From its humble beginnings as a simple markup language to the robust, multifaceted standard it is today, HTML's progression mirrors the broader evolution of web technologies. This subchapter explores the major milestones in HTML's history and highlights how each iteration has shaped modern web development.

The Early Days: HTML 1.0 and Beyond

- **HTML 1.0 – The Prototype of the Web**

Introduced in the early 1990s, HTML 1.0 was a basic markup language designed to structure simple text documents with hyperlinks. It provided a minimal set of tags to create and link documents, laying the groundwork for the interconnected web. Although limited by today's standards, HTML 1.0 established the core concept of hypertext, which remains central to the web.

- **HTML 2.0 – Establishing Standards**

As the web grew in popularity, the need for a more standardized approach to web content became apparent. HTML 2.0, released in 1995, formalized many of the practices that had emerged during the web's early days. It introduced a more consistent set of tags and attributes, paving the way for broader adoption and interoperability among different browsers.

The Expansion Phase: HTML 3.2 and HTML 4.x

- **HTML 3.2 – Embracing Multimedia**

During the mid-1990s, the web began to incorporate more diverse content. HTML 3.2 responded to this demand by introducing support for elements such as tables, applets, and more sophisticated image handling. This version allowed for richer page layouts and multimedia integration, helping to transform the web from a collection of static pages into a more dynamic environment.

- **HTML 4.0 and 4.01 – Refinement and Accessibility**

Released at the turn of the millennium, HTML 4.0 marked a significant step toward modern web design. This version placed a strong emphasis on separating content from presentation, encouraging developers to use Cascading Style Sheets (CSS) for styling. Additionally, HTML 4.x improved support for internationalization and accessibility, setting the stage for a more inclusive web experience. The push for semantic clarity in HTML 4.x laid the foundation for future enhancements in conveying meaning and structure.

The Modern Era: HTML5 and Its Innovations

- **HTML5 – A Paradigm Shift**

After years of incremental changes and several experimental extensions, HTML5 emerged as a comprehensive overhaul of the language. Officially recommended as a standard in 2014, HTML5 brought significant advancements designed to meet the demands of modern web applications:

- **Rich Media Integration**

HTML5 natively supports audio, video, and interactive graphics without relying on external plugins, streamlining multimedia development and enhancing cross-platform compatibility.

- **Enhanced Semantic Elements**

New elements like <header>, <nav>, <article>, <section>, and <footer> allow developers to structure pages more meaningfully. This semantic clarity improves accessibility, SEO, and overall maintainability of web documents.

- **APIs and Advanced Capabilities**

HTML5 isn't just about markup—it includes a suite of APIs that empower developers to build complex, interactive web applications. From offline storage and geolocation to drag-and-drop functionality and real-time communication, these features have expanded what the web can do.

- **Responsive and Mobile-Friendly Design**

With the explosion of mobile devices, HTML5 was designed to work seamlessly across a range of screen sizes and input methods. Its flexible structure and support for CSS3 media queries make it easier than ever to create responsive designs that adapt to different devices.

The Impact of HTML's Evolution

The evolution from HTML 1.0 to HTML5 represents more than just a series of technical updates—it reflects the growing ambitions of the web itself. Each version of HTML has contributed to making the internet a richer, more interactive, and more accessible medium. The move toward semantic, multimedia-friendly, and API-rich HTML has not only improved the user experience but also enabled developers to create applications that rival desktop software in functionality.

Looking Ahead

While HTML5 currently stands as the cornerstone of web development, the journey of HTML is ongoing. As web technologies continue to evolve—driven by emerging trends like progressive web apps, virtual reality, and enhanced accessibility—future revisions of HTML will undoubtedly build on the robust foundation established by its predecessors.

Conclusion

The evolution of HTML is a testament to the web's dynamic nature. From the simple, static pages of HTML 1.0 to the rich, interactive experiences enabled by HTML5, each stage has addressed the growing demands of users and developers alike. Understanding this evolution not only provides insight into how the web has developed over time but also highlights the continuous innovation that will shape the future of digital communication.

1.4. Overview of Modern Web Development

Modern web development is a dynamic and ever-evolving field that encompasses a wide range of technologies, practices, and tools. As the internet has grown from a collection of static pages into a robust ecosystem of interactive applications, web development has expanded to include not only the creation of content but also the delivery of engaging, high-performance user experiences across multiple devices. This subchapter provides an overview of the key aspects that define modern web development and outlines the trends shaping its future.

A Shift from Static to Dynamic Experiences

In the early days of the web, websites were primarily static, offering fixed content that rarely changed without manual updates. Today, modern web development emphasizes dynamic content that adapts to user interactions, context, and device capabilities. This transformation is driven by several factors:

- **Interactivity and User Engagement**

Modern web applications often resemble desktop software, offering rich interactivity and real-time feedback. JavaScript, along with frameworks like React, Angular, and Vue.js, enables developers to build sophisticated interfaces that update seamlessly without requiring full page reloads.

- **Responsive Design**

With the proliferation of smartphones, tablets, and various other devices, designing for a single screen size is no longer viable. Responsive design techniques—often implemented through CSS media queries and flexible grid systems—ensure that web content automatically adjusts to different screen sizes and orientations, delivering an optimal experience for every user.

- **Single-Page Applications (SPAs)**

SPAs have gained popularity for their ability to provide a more fluid user experience. By loading a single HTML page and dynamically updating content as users interact with the app, SPAs reduce loading times and create a smoother navigation flow, similar to that of native applications.

The Role of APIs and Microservices

Modern web development increasingly relies on Application Programming Interfaces (APIs) and microservices architectures to build scalable and modular applications:

- **APIs for Data Exchange**

APIs allow different systems and services to communicate with each other, facilitating data exchange between the client and server or among various backend systems. RESTful APIs, GraphQL, and WebSockets are commonly used to build interactive applications that can retrieve and update data in real time.

- **Microservices Architecture**

Instead of building monolithic applications, developers are breaking functionality into small, independent services. This microservices approach improves scalability, enables easier maintenance, and allows teams to deploy updates without affecting the entire application.

Emphasis on Performance and Accessibility

User experience in modern web development is not solely defined by visual appeal; performance and accessibility play crucial roles as well:

- **Performance Optimization**

Fast-loading pages are critical for retaining users and improving search engine rankings. Techniques such as code minification, image optimization, and leveraging browser caching contribute to enhanced performance. Additionally, Content Delivery Networks (CDNs) distribute content across multiple servers globally to reduce latency.

- **Accessibility and Inclusivity**

Ensuring that web applications are accessible to all users, including those with disabilities, has become a core principle of modern development. Semantic HTML, ARIA (Accessible Rich Internet Applications) roles, and adherence to web accessibility standards are essential for creating inclusive digital experiences. This focus on accessibility not only broadens the audience but also improves the overall usability of web applications.

Integration with Modern Tools and Workflows

The development process itself has evolved significantly, with modern workflows emphasizing automation, collaboration, and continuous improvement:

- **Version Control and Collaboration**

Tools like Git have become indispensable for managing code changes, enabling multiple developers to work on the same project efficiently. Platforms such as GitHub, GitLab, and Bitbucket support collaborative workflows, code reviews, and issue tracking.

- **Build and Deployment Automation**

Modern web projects often utilize build tools (e.g., Webpack, Parcel) and task runners (e.g., Gulp, npm scripts) to automate repetitive tasks such as bundling, transpiling, and testing. Continuous Integration/Continuous Deployment (CI/CD) pipelines further streamline the process of testing and deploying code, ensuring that new features and updates reach users quickly and reliably.

- **Development Environments and Testing**

The use of local development servers, containerization (with tools like Docker), and robust testing frameworks (such as Jest or Mocha for JavaScript) has transformed how web applications are developed and maintained. These practices lead to more resilient and scalable applications.

The Future of Web Development

Modern web development is characterized by rapid innovation and constant change. Emerging technologies such as Progressive Web Apps (PWAs), serverless architectures, and the integration of artificial intelligence and machine learning promise to further revolutionize how web applications are built and experienced. As developers continue to push the boundaries of what the web can do, the core principles of interactivity, responsiveness, performance, and accessibility remain as critical as ever.

Conclusion

The landscape of modern web development is rich and multifaceted. It extends far beyond simply writing HTML, CSS, and JavaScript; it involves creating immersive, high-performance experiences that are responsive to user needs and accessible to all. By

embracing new technologies and best practices, developers are transforming the web into a dynamic, interactive platform that meets the demands of a diverse and ever-evolving digital world. In the chapters that follow, we will delve deeper into the specific technologies and techniques that make up this modern ecosystem, starting with an in-depth exploration of HTML and its role in the ongoing evolution of web technologies.

Summary and Next Steps

As we reach the end of Chapter 1: Introduction to Web Technologies, it is clear that the modern web is built on a rich tapestry of interrelated technologies and evolving practices. Let's briefly recap the key insights from this chapter:

- **Understanding Web Technologies**

In Section 1.1, we laid the groundwork by defining web technologies as the diverse array of tools, languages, protocols, and frameworks that enable the creation and delivery of digital content. This overview highlighted the fundamental role these technologies play in shaping how we interact with information online.

- **The Central Role of HTML**

Section 1.2 focused on HTML as the structural backbone of the web. We explored how HTML transforms raw data into organized, accessible content and discussed its critical contribution to both user experience and web accessibility. HTML's role as the gateway to styling with CSS and interactivity with JavaScript was also emphasized.

- **The Evolution of HTML**

In Section 1.3, we traced the development of HTML from its early inception as HTML 1.0 to the versatile and robust standard that is HTML5 today. This historical perspective not only underscored the continuous innovation in web technologies but also demonstrated how each iteration of HTML has addressed the growing demands of developers and users alike.

- **Modern Web Development Landscape**

Section 1.4 provided an overview of modern web development practices. From dynamic, interactive single-page applications and responsive design to the

integration of APIs and microservices, we examined how current trends and tools have transformed the way websites and applications are built and maintained.

In summary, Chapter 1 has established a comprehensive framework for understanding the technological, historical, and practical dimensions of the web. The evolution of HTML, along with its pivotal role in structuring digital content, serves as the foundation upon which modern web development is built. This knowledge sets the stage for a deeper dive into the specifics of creating web content.

As we move forward, the next step is to roll up our sleeves and begin the hands-on journey of crafting your own web pages. In **Chapter 2: Getting Started with HTML**, we will guide you through the basics of HTML syntax and structure, helping you create your first HTML document and laying the groundwork for your web development skills.

Let's embark on this exciting journey into the world of HTML and discover how to bring web pages to life!

Chapter 2. Getting Started with HTML

Welcome to Chapter 2, where we transition from understanding the broad landscape of web technologies to diving into the very building block of the web: HTML. In this chapter, we will embark on a hands-on journey that demystifies HTML, empowering you to start creating your own web pages. Whether you are completely new to web development or looking to solidify your understanding of HTML's fundamentals, this chapter is designed to guide you step by step through the essential concepts and practical applications of HTML.

In the following pages, you will learn how HTML serves as the structural foundation of every website. We will explore the basic syntax and structure of an HTML document, introduce you to the core elements that make up web pages, and demonstrate how to construct a simple, yet fully functional HTML document from scratch. By breaking down each component into manageable sections, this chapter aims to build your confidence and prepare you for more advanced topics in web development.

Before we delve into the practical aspects of writing HTML, it is important to understand what HTML is and why it is so crucial in the creation of web content. In our first subchapter, **2.1. What Is HTML?**, we will answer these questions and set the stage for the detailed exploration of HTML's capabilities and its role in modern web development.

Let's begin our journey into HTML and unlock the potential of creating content that powers the web!

2.1. What Is HTML?

HTML, which stands for **Hypertext Markup Language**, is the cornerstone of web development. It is the language used to create and structure content on the web, defining both the layout and the elements of every webpage. Unlike traditional programming languages that focus on logic and computations, HTML is a markup language – a set of instructions that tells web browsers how to interpret and display text, images, and other media.

The Core Purpose of HTML

- **Structuring Content**

HTML provides a structured way to organize information. By using a predefined set of tags, developers can designate different types of content such as headings, paragraphs, lists, tables, and more. This structure not only ensures a consistent presentation across different devices and browsers but also helps in organizing the content in a logical and accessible manner. For instance, the use of `<header>`, `<footer>`, `<nav>`, and `<section>` elements helps break down the page into meaningful parts.

- **Defining Elements and Their Relationships**

HTML tags mark the beginning and end of an element. For example, `<p>`This is a paragraph. `</p>` tells the browser that the text between the `<p>` and `</p>` tags should be treated as a paragraph. The proper use of these tags creates a hierarchy within the document that browsers, screen readers, and search engines can use to understand the content's structure and importance.

- **Enabling Hypertext Functionality**

One of HTML's revolutionary features is its ability to create hyperlinks through the `<a>` tag. This feature enables the creation of an interconnected network of documents — a fundamental aspect of the World Wide Web. A simple hyperlink might look like `<a href="https://www.example.com">Visit Example</a>`, allowing users to navigate seamlessly from one page to another.

How HTML Works in a Web Browser

When you load a webpage, the browser performs several steps to render the content:

1. **Parsing the HTML Document**

The browser reads the HTML file from top to bottom, interpreting each tag to build the Document Object Model (DOM)—a tree-like structure that represents the page's content and its hierarchy.

2. **Integrating CSS and JavaScript**

Once the DOM is built, the browser applies CSS to style the content and executes any JavaScript code to add interactivity. This layered process—HTML for

structure, CSS for presentation, and JavaScript for behavior—ensures that web pages are both functional and visually engaging.

3. **Rendering the Page**

Finally, the browser renders the page based on the constructed DOM, styled elements, and any dynamic modifications made by JavaScript. This results in the final visual presentation seen by the user.

HTML in the Web Ecosystem

HTML does not work in isolation; it is part of a broader ecosystem of web technologies:

- **CSS (Cascading Style Sheets)**

CSS is responsible for the visual design and layout of the HTML content. While HTML organizes the content, CSS provides the styling—colors, fonts, spacing, and responsive adjustments—ensuring that the web pages are aesthetically pleasing and usable on various devices.

- **JavaScript**

JavaScript injects life into HTML pages by enabling dynamic content changes, user interactions, and asynchronous data fetching. It interacts with the DOM created by HTML to update content without requiring a full page reload, making web applications feel more like native software.

- **Server-Side Technologies**

HTML is often the final output delivered by server-side languages such as PHP, Python, or Ruby. These technologies generate HTML dynamically based on user input, database queries, or other logic, allowing for personalized and interactive web experiences.

Why HTML Matters

Understanding HTML is essential for several reasons:

- **Foundation for Web Development**

HTML is the starting point for anyone interested in creating web content. Its simplicity makes it accessible for beginners, while its versatility provides endless possibilities for advanced developers.

- **Accessibility and Search Engine Optimization (SEO)**

Well-structured HTML is key to ensuring that web content is accessible to all users, including those using assistive technologies. Additionally, search engines rely on the semantic structure provided by HTML to index and rank pages effectively.

- **Standardization and Interoperability**

HTML follows standards set by organizations like the World Wide Web Consortium (W3C) and the Web Hypertext Application Technology Working Group (WHATWG). These standards ensure that HTML code remains consistent and interoperable across different browsers and devices, facilitating a more unified web experience.

Real-World Examples

- **A Simple Webpage Example**

Consider a basic HTML document:

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <meta charset="UTF-8">
5      <title>My First Web Page</title>
6    </head>
7    <body>
8      <header>
9        <h1>Welcome to My Website</h1>
10     </header>
11     <section>
12       <p>This is a paragraph of text on my first webpage.</p>
13       <a href="https://www.example.com">Visit Example</a>
14     </section>
15     <footer>
16       <p>Copyright © 2025</p>
17     </footer>
18   </body>
19 </html>
```

This snippet shows the basic structure of an HTML document, including the DOCTYPE declaration, the `<html>` root element, and the division of content into semantic sections like `<header>`, `<section>`, and `<footer>`.

- **Interactivity with Hyperlinks**

The `<a>` tag in HTML is used to create clickable links that connect various pages and resources. This simple element forms the basis of the interconnected web, enabling users to navigate through vast networks of information.

Conclusion

HTML is the essential language that powers the web. It provides the structure and meaning to web content, making it accessible, understandable, and easy to interact with. By mastering HTML, you lay the groundwork for building robust, dynamic websites that integrate seamlessly with CSS for styling and JavaScript for behavior. In essence, HTML is the starting point for every web developer's journey — a journey that begins with understanding the basics and gradually advances to more complex and interactive designs.

Now that you have a detailed understanding of what HTML is and why it is so vital, we are ready to dive deeper into the practical aspects of creating HTML documents. In the next section, we will start exploring the basic syntax and structure of HTML, laying the foundation for your hands-on experience in web development.

2.2. Basic Syntax and Structure

Now that you understand what HTML is and its role in creating web pages, it's time to delve into its basic syntax and structure. In this subchapter, we will explore the fundamental building blocks of an HTML document, learn how to write clean and well-formed code, and understand the conventions that make HTML both human- and machine-readable.

1. The Document Type Declaration

Every HTML document begins with a **Document Type Declaration (DOCTYPE)**, which informs the browser about the version of HTML being used. In modern web development, HTML5 is the standard, and its declaration is very simple:

```
<!DOCTYPE html>
```

Purpose

The DOCTYPE declaration ensures that the browser renders the page in standards mode, which is critical for consistent behavior across different browsers.

2. The Basic HTML Document Structure

An HTML document is organized into two main sections: the `<head>` and the `<body>`. Together, they form the skeleton of your web page.

- **`<html>` Element**

The `<html>` tag serves as the root element of the HTML document. It wraps all other elements in the page.

- **`<head>` Section**

The `<head>` contains metadata and information that isn't directly displayed on the page. This includes the document title, character encoding, links to stylesheets, scripts, and other resources.

Example:

```
<head>
  <meta charset="UTF-8">
  <title>My First Web Page</title>
  <link rel="stylesheet" href="styles.css">
</head>
```

- **`<body>` Section**

The `<body>` includes all the content that is visible to the user. This is where you place text, images, links, forms, and more.

Example:

```
<body>
  <h1>Welcome to My Website</h1>
  <p>This is a simple paragraph introducing my site.</p>
  <a href="https://www.example.com">Visit Example</a>
</body>
```

Putting it all together, a complete HTML document might look like this:

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <meta charset="UTF-8">
5      <title>My First Web Page</title>
6      <link rel="stylesheet" href="styles.css">
7    </head>
8    <body>
9      <header>
10     | <h1>Welcome to My Website</h1>
11     </header>
12     <section>
13     | <p>This is a simple paragraph introducing my site.</p>
14     | <a href="https://www.example.com">Visit Example</a>
15     </section>
16     <footer>
17     | <p>&copy; 2025 My Website</p>
18     </footer>
19   </body>
20 </html>
```

3. HTML Elements and Tags

HTML documents are made up of **elements**, which are defined by **tags**. Tags come in pairs: an opening tag and a closing tag, with content placed in between.

- **Opening Tag**

Denotes the start of an element. For example, <p> starts a paragraph.

- **Closing Tag**

Denotes the end of an element. For example, </p> ends a paragraph.

- **Self-Closing Tags**

Some HTML elements do not require a closing tag and are written as self-closing. Examples include `<img>`, `<br>`, and `<hr>`. In HTML5, these are often written without the trailing slash, but you may also see ``.

4. Nesting and Hierarchy

HTML elements can be nested within one another to create a clear hierarchy and structure. Correct nesting is important for both readability and proper rendering by browsers.

Example of proper nesting:

```
<section>
  <article>
    <h2>Article Title</h2>
    <p>Article content goes here.</p>
  </article>
</section>
```

5. Attributes and Their Role

Attributes provide additional information about HTML elements and are included within the opening tag. They come in name/value pairs.

Example:

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

- **Common Attributes:**

- href in the `<a>` tag specifies the URL of the link.
- src in the `<img>` tag defines the path to the image file.
- alt in the `<img>` tag provides alternative text for accessibility.
- class and id attributes help in identifying elements for styling and scripting purposes.

6. Best Practices for Writing HTML

- **Use Semantic Elements**

Semantic HTML tags (like `<header>`, `<footer>`, `<article>`, `<nav>`, and `<section>`) convey meaning about the content they enclose. This improves accessibility and SEO.

- **Keep It Clean and Organized**

Structure your HTML in a readable format with proper indentation. This not only helps you understand your own code but also makes collaboration easier.

- **Validate Your Code**

Use HTML validators (such as the [W3C Markup Validation Service](https://validator.w3.org/)) to ensure your code adheres to web standards and is free of syntax errors.

Conclusion

The basic syntax and structure of HTML provide the foundation upon which web pages are built. By understanding the roles of the DOCTYPE, `<html>`, `<head>`, and `<body>` elements, as well as mastering the use of tags, attributes, and proper nesting, you will be well-equipped to create clear, well-structured web content. This knowledge is crucial as it not only ensures that your pages are rendered correctly across different browsers but also makes them accessible and maintainable.

In the next subchapter, we will begin exploring the key elements of an HTML document in more detail, starting with how to create your first HTML document and understanding the core components that bring your web pages to life.

2.3. Creating Your First HTML Document

Now that you understand the basic syntax and structure of HTML, it's time to put your knowledge into practice by creating your very first HTML document. This subchapter will walk you through the step-by-step process of writing, saving, and viewing an HTML file in your web browser.

1. Setting Up Your Environment

Before you begin coding, ensure you have a simple text editor installed on your computer. Many options are available:

- **Basic Editors:** Notepad (Windows), TextEdit (macOS, in plain text mode)
- **Code Editors:** Visual Studio Code (what the authors of this book use), Sublime Text, Atom, or Brackets

Using a dedicated code editor is recommended as they often include features like syntax highlighting and error detection, which can make writing HTML easier and more efficient.

2. Writing Your HTML Code

Let's start by creating a basic HTML document. Open your text editor and type the following code:

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <meta charset="UTF-8" />
5      <title>My First Web Page</title>
6    </head>
7    <body>
8      <header>
9        <h1>Welcome to My First Web Page</h1>
10     </header>
11     <section>
12       <p>
13         This is a paragraph introducing my site. Here, you can share information
14         about yourself, your interests, or the purpose of the page.
15       </p>
16       <p>Explore the links below to learn more:</p>
17       <ul>
18         <li><a href="https://www.example.com">Example Website</a></li>
19         <li><a href="https://www.anotherexample.com">Another Example</a></li>
20       </ul>
21     </section>
22     <footer>
23       <p>&copy; 2025 My First Web Page</p>
24     </footer>
25   </body>
26 </html>
```

Let's break down what this code does:

- **DOCTYPE Declaration**

The first line (`<!DOCTYPE html>`) declares that the document is written in HTML5, ensuring that browsers render the page in standards mode.

- **HTML Structure**

The `<html>` element encapsulates all the content of the page, while the `<head>` and `<body>` sections define the metadata and visible content, respectively.

- **Metadata in the `<head>`**

Inside `<head>`, the `<meta charset="UTF-8">` tag sets the character encoding, and the `<title>` tag defines the title of the web page, which appears on the browser tab.

- **Content in the `<body>`**

The `<body>` contains all the visible content of your web page. In this example, we've included:

- A `<header>` section with a main heading (`<h1>`) that welcomes visitors.
- A `<section>` containing paragraphs (`<p>`) and an unordered list (`<ul>`) with hyperlinks (`<a>`) to other websites.
- A `<footer>` that includes a copyright notice.

3. Saving the Document

Once you've written your HTML code, save the file with a `.html` extension. For example, you might name it `index.html`. Ensure that:

- The file is saved in plain text format.
- The extension is `.html`, which tells your operating system and browser that this is an HTML document.

4. Viewing Your HTML Document in a Browser

To see your creation in action:

1. Locate the `index.html` file you just saved.

2. Double-click the file, or right-click and select “Open With” to choose your preferred web browser (Chrome, Firefox, Safari, etc.).
3. The browser will render your HTML file, displaying your web page as intended.

As you view your page, observe how each element appears—headings, paragraphs, links, and lists. This immediate feedback is one of the great advantages of HTML: you can easily modify your code, save the changes, and refresh the browser to see the results.

5. Experiment and Explore

Now that you have a basic HTML document, consider experimenting with the following:

- **Modify Content**

Change the text in the headings or paragraphs to see how it affects the page.

- **Add New Elements**

Try adding more elements, such as images using the `<img>` tag or additional sections and articles.

- **Alter Attributes**

Experiment with attributes such as `href` in `<a>` tags or `src` in `<img>` tags to learn how they influence the behavior and appearance of your content.

6. Using Developer Tools

Most modern browsers come with built-in developer tools that allow you to inspect the HTML structure of your page. Right-click on any element and select “Inspect” (or “Inspect Element”) to see how your browser interprets your code. This is a powerful way to debug and learn more about HTML rendering.

Conclusion

Creating your first HTML document is a significant milestone in your journey as a web developer. This process introduces you to the workflow of writing code, saving files, and viewing results in a browser. With practice, you'll develop an intuition for structuring content and gradually build more complex web pages.

In the next chapters, we will delve deeper into HTML elements and attributes, exploring the various building blocks that make up a fully functional web page. Get ready to expand your toolkit and further your understanding of HTML's capabilities!

2.4. Understanding the DOCTYPE Declaration

Every HTML document begins with a DOCTYPE declaration, a seemingly small but crucial piece of code that plays a significant role in how your web page is rendered by browsers. In this subchapter, we'll explain what the DOCTYPE declaration is, why it is important, and how it has evolved alongside HTML.

What Is the DOCTYPE Declaration?

The DOCTYPE declaration is an instruction at the very top of an HTML document that tells the browser which version of HTML the page is written in. Although it isn't an HTML tag, it is a required component of an HTML file. For example, in HTML5, the declaration is as simple as:

```
<!DOCTYPE html>
```

This declaration signals to the browser that the document follows the HTML5 standard. In earlier versions of HTML, the DOCTYPE was more complex and included references to a Document Type Definition (DTD) that specified the rules for the language.

Why Is the DOCTYPE Important?

1. Standards Mode vs. Quirks Mode

The DOCTYPE declaration plays a key role in determining how the browser renders a web page. There are two primary rendering modes:

- **Standards Mode**

When a correct DOCTYPE is provided, the browser interprets and displays the page using current web standards. This ensures that your CSS, HTML, and JavaScript work as intended.

- **Quirks Mode**

If the DOCTYPE is missing or incorrect, browsers may revert to quirks mode. In this mode, the browser emulates older, non-standard behavior to maintain compatibility with legacy websites, which can lead to inconsistent or unexpected rendering.

2. Consistency Across Browsers

A proper DOCTYPE declaration helps ensure that your page displays consistently across different browsers and devices. By explicitly declaring the version of HTML, you reduce the risk of browser-specific rendering issues.

3. Future-Proofing Your Code

Using the HTML5 DOCTYPE (`<!DOCTYPE html>`) is not only simpler but also future-proof. It tells browsers to apply the latest standards, making your document more robust and easier to maintain as web standards continue to evolve.

Evolution of the DOCTYPE Declaration

- **HTML 4.01 and XHTML 1.0**

Earlier versions of HTML required more complex DOCTYPE declarations. For example, an HTML 4.01 Transitional document might start with:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
```

And an XHTML 1.0 Strict document would use:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

These declarations reference a DTD that defines the rules and structure of the document. While these versions provided detailed instructions to the browser, they also added complexity for developers.

- **HTML5 Simplification**

With HTML5, the need for such detailed declarations was eliminated. The streamlined DOCTYPE:

```
<!DOCTYPE html>
```

is not only easier to remember but also serves the same purpose—triggering standards mode in all modern browsers.

Best Practices for Using the DOCTYPE

- **Always Include the DOCTYPE**

Even if you are building a very simple or experimental page, always start with the DOCTYPE declaration. This habit ensures that your document is interpreted correctly by browsers.

- **Use the HTML5 DOCTYPE**

Since HTML5 is the current standard and widely supported by modern browsers, use the simplified DOCTYPE (`<!DOCTYPE html>`) for all new projects. It reduces potential errors and confusion that might arise from using outdated or complex declarations.

- **Check Your Document**

If your page isn't rendering as expected, verify that the DOCTYPE is present and correctly formatted at the very beginning of your HTML file. This is one of the simplest troubleshooting steps when facing unexpected behavior.

Conclusion

The DOCTYPE declaration might seem like a small detail, but it is fundamental to ensuring that your web pages are rendered correctly and consistently across all browsers. By signaling which version of HTML your document uses, it helps maintain standards compliance and avoids the pitfalls of quirks mode. With the simplified declaration in HTML5, the process is now more straightforward than ever, allowing you to focus on crafting quality web content.

In the next chapters, we will continue our exploration of HTML by diving into the detailed elements and attributes that bring your web pages to life. Get ready to build on this foundation as we start exploring the structure and components of HTML in greater depth.

Chapter 3. The Anatomy of an HTML Document

Welcome to Chapter 3, where we take a closer look at the inner workings of an HTML document. Building on the foundation laid in the previous chapters, this chapter will guide you through the intricate structure and key components that come together to form a complete web page. By understanding the anatomy of an HTML document, you will gain the insight needed to craft well-organized, accessible, and efficient websites.

In this chapter, we will explore the following:

- **The Root Element**

Discover how the `<html>` tag serves as the container for all other elements and establishes the basic framework of your document.

- **The `<head>` Section**

Learn about the crucial role of the `<head>` element, which houses metadata, links to external resources, and other non-visible elements that are vital for page functionality and search engine optimization.

- **The `<body>` Section**

Delve into the heart of your HTML document, where visible content is placed. We will discuss how to structure content using semantic elements, ensuring clarity for both users and search engines.

- **Comments and code organization**

Understand how to introduce and use comments in our HTML code and how to organize the code in blocks for better readability and maintainability.

By breaking down the anatomy of an HTML document into its essential parts, this chapter will provide you with the practical knowledge needed to design robust and standards-compliant web pages. Whether you are refining an existing project or building one from scratch, understanding these core concepts is key to effective web development.

Let's begin our deep dive into the anatomy of an HTML document and unlock the secrets behind building web pages that are both beautiful and functional.

3.1. The <html> Element

The <html> element is the cornerstone of every HTML document—it serves as the root container for all other elements (with the exception of the DOCTYPE declaration). This subchapter delves into the role, structure, and best practices associated with the <html> element, providing you with a clear understanding of its importance in constructing a well-formed web page.

Definition and Role

- **Root Container**

The <html> element encapsulates the entire content of your web page. Every visible element on your page, whether it appears in the <head> or the <body>, must reside within the opening <html> tag and its corresponding closing </html> tag. This encapsulation forms the basis of the Document Object Model (DOM), which browsers use to render and manage the content.

- **Starting Point for the Browser**

As the very first tag (following the DOCTYPE declaration), the <html> element signals to the browser that what follows is an HTML document. This allows the browser to correctly interpret and apply standards, rendering the page in "standards mode" rather than "quirks mode."

Common Attributes

While the <html> element is often used in its simplest form, it can include attributes that enhance its functionality and accessibility:

- **lang:**

The lang attribute specifies the primary language of the document. This is particularly important for search engines and assistive technologies (such as screen readers), as it aids in delivering content in the correct language context.

Example:

```
<html Lang="en">
```

Structure Within the <html> Element

The <html> element generally contains two primary sections:

1. The <head> Section

This section is dedicated to metadata and non-visual elements. It includes elements such as:

- <meta> tags for character encoding, viewport settings, and SEO.
- The <title> tag, which sets the title of the document.
- Links to external resources like CSS stylesheets and scripts.

2. The <body> Section

This is where all the content intended for display resides. It includes text, images, links, forms, and other interactive or visual elements.

A simple HTML document structure would look like this:

```
1  <!DOCTYPE html>
2  <html lang="en">
3      <head>
4          <meta charset="UTF-8">
5          <title>My First Web Page</title>
6          <!-- Link to external stylesheets or scripts can be added here -->
7      </head>
8      <body>
9          <header>
10             <h1>Welcome to My Website</h1>
11          </header>
12          <section>
13             <p>This is where your main content goes.</p>
14          </section>
15          <footer>
16             <p>&copy; 2025 My Website</p>
17          </footer>
18      </body>
19 </html>
```

Best Practices

- **Always Declare the Language**

Including the lang attribute in your `<html>` tag is a best practice that promotes accessibility and ensures that browsers and assistive technologies can correctly interpret the content.

- **Keep It Clean and Simple**

While the `<html>` element can include various attributes, its primary role is to act as the container for the document. Avoid cluttering it with unnecessary attributes or inline styles that could be managed more effectively through CSS or JavaScript.

- **Consistency and Standards**

Following standards by placing all content within the `<html>` element ensures that your document is well-structured and easily maintained. This consistency is key for cross-browser compatibility and for developers who might work on the project in the future.

Conclusion

The `<html>` element is much more than just a container—it is the foundation upon which your entire web document is built. By understanding its role, common attributes, and proper usage, you ensure that your HTML documents are well-formed, accessible, and compatible with modern web standards. In the next subchapter, we will explore the `<head>` section in detail, discussing the metadata and resources that prepare your document for successful rendering and interaction.

3.2. The `<head>` Section: Metadata, Title, and Links

The `<head>` section of an HTML document is a crucial part of its anatomy, even though its contents are not directly displayed on the web page. Instead, the `<head>` provides essential information about the document—metadata, title, and links to external resources—that helps browsers, search engines, and other services process and render the page effectively. In this subchapter, we'll explore the key components of the `<head>` section, their purposes, and best practices for using them.

What Goes Inside the `<head>` Section?

The <head> section typically includes several types of elements that define the document's properties and behavior:

- **Metadata:**

Metadata consists of information about the document that is not displayed directly on the page. This includes character encoding, viewport settings, author information, keywords, and descriptions. Metadata helps browsers correctly render the page and improves search engine optimization (SEO).

- **Title:**

The <title> tag specifies the title of the document. This title appears on the browser tab, in bookmarks, and in search engine results, making it a critical element for both usability and SEO.

- **Links to External Resources**

The <head> section is also used to link external stylesheets, fonts, icons, and even JavaScript files. These resources are essential for styling the page and adding functionality.

Key Elements of the <head> Section

1. Meta Tags

Meta tags provide vital information about your document. Common examples include:

- **Character Encoding:**

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

This tag specifies the character encoding for the document, ensuring that text is displayed correctly.

- **Viewport Settings:**

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

This tag is essential for responsive design, instructing the browser on how to control the page's dimensions and scaling on different devices.

- **Description and Keywords:**

```
<meta name="description" content="A brief description of the web page.">
<meta name="keywords" content="HTML, CSS, JavaScript, web development">
```

These tags provide search engines with a summary of the page's content and relevant keywords, which can help improve the page's SEO ranking.

- **Title Tag**

The <title> tag sets the title of the document:

```
<title>My First Web Page</title>
```

This title is displayed in the browser's title bar or tab, making it an important element for both navigation and SEO. A clear, descriptive title helps users understand the purpose of the page at a glance.

- **Link Tags**

Link tags are used to connect external resources to your HTML document. The most common use is linking a CSS stylesheet:

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

Additionally, you might use link tags to specify icons (such as a favicon) or preconnect to important resources:

```
<link rel="icon" href="favicon.ico" type="image/x-icon">
<link rel="preconnect" href="https://fonts.googleapis.com">
```

- **Script Tags (Optional)**

Although scripts can be placed in the <head>, it's common to include them at the end of the <body> for better performance. However, if necessary, you can include critical JavaScript files in the <head>:

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

The defer attribute ensures that the script is executed after the document has been parsed, which can help improve page loading speed.

Example of a Well-Structured <head> Section

Here's an example that brings together all the elements discussed:

```

1 <head>
2   <!-- Specify the character encoding -->
3   <meta charset="UTF-8">
4
5   <!-- Set the viewport for responsive design -->
6   <meta name="viewport" content="width=device-width, initial-scale=1.0">
7
8   <!-- Provide a description and keywords for SEO -->
9   <meta name="description" content="An introductory web page to learn HTML.">
10  <meta name="keywords" content="HTML, web development, beginner, tutorial">
11
12  <!-- Define the title of the document -->
13  <title>My First Web Page</title>
14
15  <!-- Link to an external CSS stylesheet -->
16  <link rel="stylesheet" href="styles.css">
17
18  <!-- Link to a favicon -->
19  <link rel="icon" href="favicon.ico" type="image/x-icon">
20
21  <!-- Optionally, include a deferred script -->
22  <script src="head-script.js" defer></script>
23 </head>

```

Best Practices for the <head> Section

- **Keep It Organized**

Group similar elements together (e.g., all meta tags at the top) to make the <head> section easy to read and maintain.

- **Use Descriptive Metadata**

Ensure that meta tags like description and keywords accurately reflect the content of your page, which can help improve SEO and accessibility.

- **Prioritize Performance**

Use attributes like defer or async for script tags when including JavaScript to avoid blocking the rendering of the page. Also, consider preloading or preconnecting to critical resources.

- **Validate Your Code**

Regularly check your HTML for errors using validators like the [W3C Markup Validation Service](#). This helps ensure that your document adheres to web standards.

Conclusion

The `<head>` section is an indispensable part of every HTML document. It may not display content directly to users, but it sets the stage for how your page is processed and rendered by browsers and search engines. By understanding and effectively utilizing metadata, the title, and links to external resources, you can ensure that your web pages are both functional and optimized for performance and discoverability.

In the next subchapter, we will shift our focus to the `<body>` section—the part of your document that holds all the visible content. This will guide you through organizing and structuring content for an engaging user experience.

3.3. The `<body>` Section: Content and Structure

The `<body>` section is where the heart of your web page comes to life. Unlike the `<head>`, which contains metadata and links to external resources, the `<body>` holds all the visible content that users interact with—text, images, links, multimedia, and interactive elements. In this subchapter, we'll explore the structure and significance of the `<body>` section, how to organize content effectively, and best practices to ensure your pages are both engaging and accessible.

The Role of the `<body>` Section

- **Primary Display Area**

Everything you want your visitors to see and interact with is placed inside the `<body>`. This includes headings, paragraphs, images, videos, forms, and more.

- **Content Organization**

The <body> is where you structure your content logically using semantic HTML elements. Proper organization not only improves readability but also enhances accessibility and search engine optimization (SEO).

- **Interactivity and User Experience**

By combining HTML with CSS for styling and JavaScript for behavior, the <body> section becomes the stage where dynamic and interactive experiences are created.

Structuring the Content

A well-structured <body> typically includes the following components:

1. **Headers and Navigation**

Elements like <header> and <nav> are used at the top of the <body> to introduce the page and provide a way for users to navigate the site.

```
<header>
  <h1>Welcome to My Website</h1>
  <nav>
    <ul>
      <li><a href="#about">About</a></li>
      <li><a href="#services">Services</a></li>
      <li><a href="#contact">Contact</a></li>
    </ul>
  </nav>
</header>
```

2. **Main Content Area**

The main content is usually wrapped in elements such as <main>, <section>, or <article>. These semantic tags help group related content together and provide meaning about its purpose.

```
<main>
  <section id="about">
    <h2>About Us</h2>
    <p>This section describes who we are and what we do.</p>
  </section>
  <article>
    <h2>Latest News</h2>
    <p>Read about the most recent updates and events.</p>
  </article>
</main>
```

3. Sidebars and Complementary Content

Additional content, such as related links, advertisements, or supplementary information, can be organized using `<aside>` elements.

```
<aside>
  <h3>Related Articles</h3>
  <ul>
    <li><a href="#article1">Article 1</a></li>
    <li><a href="#article2">Article 2</a></li>
  </ul>
</aside>
```

4. Footers

The `<footer>` element typically contains information like contact details, copyright notices, and links to legal information.

```
<footer>
  <p>&copy; 2025 My Website</p>
</footer>
```

Best Practices for the `<body>` Section

- **Semantic Markup**

Use semantic HTML tags to clearly delineate different parts of your content. This not only improves accessibility for screen readers but also enhances SEO by conveying the structure and relevance of your content to search engines.

- **Proper Nesting**

Ensure that elements are correctly nested within the `<body>` to maintain a logical hierarchy. For example, a `<section>` should contain related content, and headings should follow a logical order.

- **Maintain Readability**

Keep your code clean and well-indented. Organized markup makes it easier to maintain your document and collaborate with other developers.

- **Responsive Design**

Structure your content in a way that it can adapt to various screen sizes and devices. This often involves using CSS alongside HTML to create a flexible layout.

Example of a Well-Structured `<body>` Section

Below is a simple example that demonstrates how the `<body>` can be organized using semantic elements:

```
<body>
  <header>
    <h1>Welcome to My Website</h1>
    <nav>
      <ul>
        <li><a href="#home">Home</a></li>
        <li><a href="#about">About</a></li>
        <li><a href="#contact">Contact</a></li>
      </ul>
    </nav>
  </header>
  <main>
    <section id="home">
      <h2>Home</h2>
      <p>This is the home section, where we welcome visitors and introduce the site.</p>
    </section>
    <section id="about">
      <h2>About Us</h2>
      <p>Learn more about our mission, values, and team.</p>
    </section>
    <article>
      <h2>Latest Blog Post</h2>
      <p>Read our latest insights and updates on industry trends.</p>
    </article>
  </main>
  <aside>
    <h3>Related Links</h3>
    <ul>
      <li><a href="#resource1">Resource 1</a></li>
      <li><a href="#resource2">Resource 2</a></li>
    </ul>
  </aside>
  <footer>
    <p>&copy; 2025 My Website</p>
  </footer>
</body>
```

Conclusion

The `<body>` section is where your web page's content is brought to life. By understanding its role and applying best practices in structuring content, you ensure that your pages are not only visually appealing but also logically organized and accessible. A well-structured `<body>` improves user experience, facilitates easier maintenance, and enhances the overall effectiveness of your web content.

In the next chapters, we will also examine additional elements and best practices that contribute to the overall integrity of your HTML document, deepening our understanding of web page construction.

3.4. Comments and Code Organization

Comments and code organization are essential practices in HTML development that help make your documents more readable, maintainable, and collaborative. While comments are not rendered by the browser, they serve as internal documentation, and well-organized code facilitates easier updates and debugging. In this subchapter, we will explore how to effectively use comments and structure your HTML code.

Using Comments in HTML

HTML comments are created using the following syntax:

```
<!-- This is an HTML comment -->
```

- **Purpose of Comments**
 - **Documentation:** Provide explanations about sections of your code. For example, you might explain why a particular section exists or note any non-obvious design decisions.
 - **Organization:** Label different parts of your document, making it easier to navigate and update the code later.
 - **Debugging:** Temporarily disable parts of your code without deleting them. This is useful when testing changes or troubleshooting issues.
- **Best Practices for Comments**

- **Be Concise and Clear:** Write comments that are to the point. Avoid overly verbose explanations that might clutter your code.
- **Keep Comments Updated:** Outdated comments can lead to confusion. Make sure they accurately reflect the current state of the code.
- **Use Comments Sparingly:** While comments are helpful, overusing them can make your code harder to read. Focus on commenting complex or non-intuitive sections.

Organizing Your Code

A well-organized HTML document is easier to read and maintain. Here are some tips for organizing your HTML code effectively:

- **Consistent Indentation**

Use consistent indentation (usually two or four spaces per level) to visually represent the hierarchy and nesting of elements. This makes it easier to follow the structure of your document.

- **Logical Sectioning**

Divide your document into logical sections using semantic elements. For example, group related content within `<header>`, `<nav>`, `<main>`, `<section>`, `<article>`, `<aside>`, and `<footer>` tags. Each section can be preceded by a comment to denote its purpose:

```
<!-- Header Section: Contains the site logo and navigation -->
<header>
  <h1>My Website</h1>
  <nav>
    <ul>
      <li><a href="#home">Home</a></li>
      <li><a href="#about">About</a></li>
      <li><a href="#contact">Contact</a></li>
    </ul>
  </nav>
</header>
```

- **Separation of Concerns**

Keep your HTML focused on structure and content. Link external stylesheets and scripts via the <head> or at the end of the <body>, rather than embedding them directly into the document. This separation makes your HTML easier to manage and reduces clutter.

- **Modular Code**

For larger projects, consider breaking your HTML into smaller, reusable components. This could involve using server-side includes or client-side frameworks that allow you to manage content in a modular fashion.

Example of Well-Organized Code with Comments

Below is an example that demonstrates both proper code organization and the use of comments:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <!-- Metadata and external resources -->
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>My Organized Web Page</title>
    <link rel="stylesheet" href="styles.css">
  </head>
  <body>
    <!-- Header Section: Contains the site logo and main navigation -->
    <header>
      <h1>My Organized Web Page</h1>
      <nav>
        <ul>
          <li><a href="#home">Home</a></li>
          <li><a href="#services">Services</a></li>
          <li><a href="#contact">Contact</a></li>
        </ul>
      </nav>
    </header>
```

```
<!-- Main Content Section: Primary content of the page -->
<main>
  <!-- About Section: Introduces the site -->
  <section id="home">
    <h2>Welcome</h2>
    <p>This section welcomes visitors and introduces the site.</p>
  </section>
```

```
<!-- Services Section: Details the services offered -->
<section id="services">
  <h2>Our Services</h2>
  <p>Here we describe the services we offer.</p>
</section>
</main>
```

```
<!-- Sidebar Section: Contains related links or additional information -->
<aside>
  <h3>Related Links</h3>
  <ul>
    <li><a href="#blog">Our Blog</a></li>
    <li><a href="#faq">FAQ</a></li>
  </ul>
</aside>

<!-- Footer Section: Contains copyright and contact info -->
<footer>
  <p>&copy; 2025 My Organized Web Page</p>
</footer>
</body>
</html>
```

Conclusion

Effective use of comments and well-organized code are key to maintaining a clean, readable, and efficient HTML document. Comments provide context and clarity, while structured, modular code makes it easier to manage and scale your project. By adopting these best practices, you'll not only improve your own workflow but also make your work more accessible to other developers who might collaborate with you in the future.

In the next chapter, we will look at elements that we can use in our HTML document to build the desired web page.

Chapter 4. Text and Structural Elements

4.1. Headings, Paragraphs, and Block Elements

The primary purpose of a website is to present information. Whether the content consists of a personal blog, an online newspaper, a company website, or a scientific publication, the information must be organized in a logical, readable, and accessible manner. HTML provides a rich collection of elements for structuring textual content, allowing browsers, search engines, and assistive technologies to understand the purpose and hierarchy of the information presented.

In this section, we will examine three of the most fundamental building blocks of every HTML document:

- headings, which establish the logical hierarchy of the document;
- paragraphs, which organize textual information into readable units;
- block-level elements, which provide the structural framework of a web page.

Understanding these elements is essential before learning more advanced HTML features.

Headings

Headings define the structure and hierarchy of a document. Just as chapters and sections organize a book, headings organize the content of a web page.

HTML provides **six heading levels**, from `<h1>` to `<h6>`:

- `<h1>` – Main heading (highest level)
- `<h2>` – Major section
- `<h3>` – Subsection
- `<h4>` – Lower-level subsection
- `<h5>` – Minor heading

- `<h6>` – Lowest heading level

A simple example is shown below.

```
<h1>Introduction to HTML</h1>
<h2>History of HTML</h2>
<h3>The Early Years</h3>
<h3>HTML5</h3>
<h2>Future of HTML</h2>
```

A browser typically displays headings using different font sizes and weights.

Element	Typical Appearance	Purpose
<code><h1></code>	Largest	Page title
<code><h2></code>	Slightly smaller	Main section
<code><h3></code>	Medium	Subsection
<code><h4></code>	Smaller	Sub-subsection
<code><h5></code>	Small	Minor heading
<code><h6></code>	Smallest	Lowest level heading

The Importance of Proper Heading Hierarchy

Many beginners choose heading levels based solely on font size. For example, they may use an `<h3>` because it "looks better" than an `<h2>`. This is considered poor practice. Headings should reflect the logical structure of the document.

For example:

```
<h1>Computer Networks</h1>

<h2>Network Models</h2>

<h3>OSI Model</h3>

<h3>TCP/IP Model</h3>

<h2>Network Devices</h2>

<h3>Routers</h3>

<h3>Switches</h3>
```

Notice that the heading levels increase and decrease naturally according to the document hierarchy.

Avoid structures such as:

```
<h1>Main Title</h1>

<h4>Introduction</h4>
```

unless the skipped heading levels are intentionally omitted for a valid structural reason.

A proper heading hierarchy provides several benefits:

- improves readability;
- helps screen readers navigate the document;
- assists search engines in understanding the page;
- simplifies document maintenance.

The <p> Element

The paragraph element (<p>) is used to represent a paragraph of text.

Syntax:

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

A web page typically contains numerous paragraphs.

Example:

```
<h1>Artificial Intelligence</h1>

<p>
Artificial Intelligence (AI) is one of the fastest-growing fields in computer science.
</p>

<p>
Modern AI systems are capable of recognizing images, understanding natural language,
playing games, and assisting humans in many daily activities.
</p>
```

Browsers automatically insert vertical spacing before and after each paragraph, making the content easier to read.

Paragraphs Are Not Line Breaks

A common beginner mistake is using multiple paragraphs simply to create vertical spacing.

Incorrect:

```
<p>First sentence.</p>

<p></p>

<p></p>

<p>Second sentence.</p>
```

Paragraphs represent **independent blocks of text**, not empty space.

Spacing should be controlled using **CSS**, not by inserting empty HTML elements.

Similarly, pressing the **Enter** key multiple times inside the HTML source does not create multiple blank lines in the browser.

For example,

```
<p>
This sentence.

This sentence appears immediately afterwards.
</p>
```

will be rendered as

This sentence. This sentence appears immediately afterwards.

Block-Level Elements

Most structural HTML elements are **block-level elements**.

A block-level element typically:

- begins on a new line;
- occupies the full available width of its parent container;
- can contain other block and inline elements (depending on the element).

Examples include:

- <h1>–<h6>
- <p>
- <div>
- <section>
- <article>
- <header>
- <footer>
- <main>
- <nav>
- <aside>

Consider the following code:

```
<h1>Programming Languages</h1>

<p>Python is widely used in data science.</p>

<p>JavaScript is the language of the Web.</p>
```

The browser places each element on a separate line because they are block-level elements.

Visually, the layout resembles:

```
Programming Languages

Python is widely used in data science.

JavaScript is the language of the Web.
```

The browser automatically creates the separation between blocks.

The <div> Element

One of the most frequently used block-level elements is <div>.

The word **div** stands for **division**.

Unlike <section> or <article>, the <div> element has **no semantic meaning**. It simply groups other elements together.

Example:

```
<div>
  <h2>News</h2>

  <p>Today's headlines include developments in artificial intelligence.</p>

  <p>Several universities announced new research projects.</p>
</div>
```

The <div> element is commonly used:

- for page layout;
- to group related elements;
- as a target for CSS styling;

- as a container manipulated by JavaScript.

Later chapters will demonstrate how CSS uses `<div>` elements to create flexible page layouts.

Nested Block Elements

Block elements may contain other block elements, creating a hierarchical document structure.

Example:

```
<article>
  <h2>Web Technologies</h2>
  <section>
    <h3>HTML</h3>
    <p>
      HTML defines the structure of a web page.
    </p>
  </section>

  <section>
    <h3>CSS</h3>
    <p>
      CSS defines the presentation of a web page.
    </p>
  </section>
</article>
```

This hierarchical organization resembles the outline of a book:

- Article
 - Section
 - Heading
 - Paragraph
 - Section
 - Heading
 - Paragraph

Such organization improves readability for both humans and software.

Common Beginner Mistakes

Students learning HTML frequently make several mistakes when working with text elements.

Using headings for styling

```
<h3>This text should simply be larger.</h3>
```

If the purpose is only to increase the font size, CSS should be used instead.

Using multiple `<br>` elements instead of paragraphs

```
Text line 1.<br><br><br>  
Text line 2.
```

This does not represent proper document structure.

Using empty paragraphs

```
<p></p>
```

Empty paragraphs should be avoided.

Skipping heading levels

```
<h1>Main Title</h1>  
  
<h4>Section</h4>
```

Maintain a logical heading hierarchy whenever possible.

Best Practices

When creating textual content in HTML, follow these recommendations:

- Use exactly one `<h1>` element to represent the primary title of the page.
- Organize headings hierarchically.
- Use `<p>` only for paragraphs of text.
- Separate presentation from structure—do not use headings to obtain larger text.

- Use semantic block elements (<section>, <article>, <header>, etc.) whenever possible.
- Reserve <div> for situations where no semantic element is appropriate.
- Keep HTML well-indented and properly organized.

Conclusions

Headings, paragraphs, and block-level elements form the foundation of every HTML document. Headings establish the logical hierarchy of the content, paragraphs divide text into meaningful units, and block-level elements organize the page into distinct structural sections. Together, these elements make web pages easier to read, easier to maintain, and more accessible to users and search engines.

In the next section, we will examine **inline elements** and learn how HTML can be used to emphasize text, apply formatting, and represent specialized textual content without disrupting the overall document structure.

4.2. Inline Elements and Formatting Text

While the previous section focused on block-level elements that establish the overall structure of a page, this section turns to inline elements: the tags that operate within the flow of text itself. Inline elements do not begin on a new line and do not claim the full width of their container; instead, they sit alongside surrounding content, allowing authors to mark up individual words or short phrases for meaning, emphasis, or specialized presentation.

In this section, we will examine:

- the distinction between inline and block-level elements;
- semantic elements used to emphasize or annotate text;
- presentational elements and why semantic alternatives are usually preferable;
- the
 element and when line breaks are appropriate.

Inline vs. Block-Level Elements

An inline element flows within a line of text rather than interrupting it. It only occupies as much horizontal space as its content requires, and several inline elements can appear one after another on the same line.

Common inline elements include:

- `<a>` — hyperlinks
- `<strong>` and `<em>` — semantic emphasis
- `<span>` — a generic inline container
- `<img>` — embedded images
- `<code>` — inline code fragments
- `<small>`, `<sub>`, `<sup>` — minor text and scripted text
- `<abbr>` — abbreviations

Consider the following paragraph:

```
1 <p>
2   Learning <strong>HTML</strong> is the first step toward
3   becoming a <em>web developer</em>. Visit
4   <a href="https://www.w3.org">the W3C website</a> to read
5   the official specification.
6 </p>
```

Notice that `<strong>`, `<em>`, and `<a>` do not break the paragraph into separate lines. They are rendered inline, as part of the same continuous block of text produced by the surrounding `<p>` element.

Emphasis and Importance: `<strong>` and `<em>`

Two of the most commonly used inline elements are `<strong>` and `<em>`.

`<strong>` indicates strong importance, seriousness, or urgency, while `<em>` indicates stressed emphasis, changing the meaning of the sentence in which

it appears. Both carry semantic meaning that assistive technologies such as screen readers can announce, unlike purely visual styling.

```
1 <p><strong>Warning:</strong> Do not disconnect the power supply
2 while the update is in progress.</p>
3 <p>I <em>did</em> submit the assignment before the deadline.</p>
4
```

In the second example, italicizing the word "did" changes how the sentence is read aloud and understood; it is not merely a visual effect.

Additional Semantic Text-Level Elements

HTML provides several other inline elements that convey specific meaning about a piece of text:

- `<abbr title="...">` — marks an abbreviation and supplies its expansion;
- `<cite>` — references the title of a creative work;
- `<code>` — marks a fragment of computer code;
- `<mark>` — highlights text that is relevant in the current context;
- `<small>` — represents side comments and small print;
- `<sub>` and `<sup>` — subscript and superscript text;
- `<time datetime="...">` — represents a specific date or time in a machine-readable form.

```
1 <p>
2   The <abbr title="World Wide Web Consortium">W3C</abbr>
3   maintains the HTML specification. Water is
4   H<sub>2</sub>O, and the article was
5   <mark>updated on</mark>
6   <time datetime="2026-03-01">March 1, 2026</time>.
7 </p>
```

Each of these elements adds meaning that a browser, a search engine, or an assistive technology can make use of, beyond the way the text merely looks.

Presentational Elements: ****, **<i>**, **<u>**, and the Legacy **** Tag

HTML also defines a small number of purely presentational inline elements: **** (bold text), **<i>** (italic text), and **<u>** (underlined text). These remain valid in HTML5, but they carry no semantic meaning — they only affect appearance.

Use ** and **** when meaning is intended, and reserve ****, **<i>**, and **<u>** for cases where a stylistic convention applies without implying importance or emphasis — for example, italicizing a foreign-language phrase or bolding a keyword the first time it is introduced in a glossary.**

```
1 <!-- Stylistic, no semantic weight -->
2 <p>The term <i>habeas corpus</i> is Latin.</p>
3
4 <!-- Semantic emphasis -->
5 <p><strong>Note:</strong> this field is required.</p>
```

The **** element, once used to set typeface, size, and color directly in HTML, has been removed from the HTML5 specification. All visual styling of this kind now belongs in CSS, which cleanly separates content from presentation.

Line Breaks: The **
** Element

The **
** element inserts a single line break within a block of text. It is a void element and never has a closing tag.

**
** is appropriate in content where the line breaks are themselves part of the meaning, such as a mailing address or the lines of a poem:

```
1 <p>
2   Technical University of Cluj-Napoca<br>
3   Str. Memorandumului nr. 28<br>
4   Cluj-Napoca, Romania
5 </p>
```

As discussed in the previous section, `<br>` should never be used to simulate the spacing between paragraphs.

Common Beginner Mistakes

- Using `<b>` or `<i>` when `<strong>` or `<em>` is semantically intended.
- Chaining multiple `<br>` elements to create vertical spacing.
- Wrapping an entire paragraph of text in `<span>`, instead of a block-level element such as `<p>`.
- Placing block-level elements (such as `<div>` or `<p>`) inside an inline element such as `<span>` or `<a>`, which produces invalid HTML.

Best Practices

- Choose elements based on meaning, not on how they happen to look by default.
- Use `<strong>` and `<em>` for genuine importance and emphasis.
- Reserve `<b>`, `<i>`, and `<u>` for established stylistic conventions.
- Use CSS, not `<br>`, to control spacing.
- Keep inline elements inside an appropriate block-level parent.

Conclusions

Inline elements let authors mark up words and short phrases without disrupting the flow of surrounding text. Choosing the right inline element, semantic when meaning is intended, presentational only for established stylistic conventions, keeps HTML documents accessible, well-structured, and easy to maintain.

In the next section, we will look at how HTML organizes items into lists, using ordered, unordered, and definition lists to represent collections of related content.

4.3. Lists: Ordered, Unordered, and Definition Lists

A great deal of the content on the web is naturally list-like: navigation menus, steps in a set of instructions, ingredients in a recipe, or terms and their definitions. HTML provides three dedicated list types for representing this kind of content in a way that is both visually consistent and structurally meaningful.

In this section, we will examine:

- unordered lists, for collections of items with no inherent sequence;
- ordered lists, for items that follow a specific sequence;
- definition lists, for pairing terms with their descriptions;
- how lists can be nested to represent hierarchical content.

Unordered Lists

An unordered list represents a collection of items where the order does not matter. It is created with the `<ul>` element, and each item is wrapped in an `<li>` (list item) element.

```
1 <ul>
2   <li>HTML</li>
3   <li>CSS</li>
4   <li>JavaScript</li>
5 </ul>
```

By default, browsers render each `<li>` on its own line, preceded by a bullet marker. The bullet style itself is a matter of presentation and can be changed freely with CSS.

Ordered Lists

An ordered list represents items that follow a meaningful sequence, such as the steps of a procedure. It is created with the `<ol>` element.

```
1 <ol>
2   <li>Open a text or code editor.</li>
3   <li>Write the HTML structure.</li>
4   <li>Save the file with an .html extension.</li>
5   <li>Open the file in a web browser.</li>
6 </ol>
```

Browsers number the items automatically, starting at 1. Several attributes let you adjust this default behavior:

- `start` — sets the number of the first item (e.g., `start="5"`);
- `reversed` — counts downward instead of upward;
- `type` — chooses the numbering style ("I", "A", "a", "I", or "i").

```
1 <ol start="3" type="A">
2   <li>Introduction</li>
3   <li>Methodology</li>
4   <li>Results</li>
5 </ol>
```

Choose `<ol>` when the sequence carries meaning, and `<ul>` when it does not. Renumbering the items of an `<ol>` should never change what the content means.

Nesting Lists

Lists may be nested inside one another to represent hierarchical content, such as a multi-level outline or a navigation menu with sub-items. A nested list is placed inside the `<li>` element of the parent list.

```
1 <ul>
2   <li>Front-end Technologies
3     <ul>
4       <li>HTML</li>
5       <li>CSS</li>
6       <li>JavaScript</li>
7     </ul>
8   </li>
9   <li>Back-end Technologies</li>
10 </ul>
```

Browsers typically indent nested lists and may vary the bullet style at each level, which visually reinforces the hierarchy.

Definition Lists

A definition list associates terms with their descriptions. It is created with the `<dl>` element, and each entry consists of a `<dt>` (definition term) followed by one or more `<dd>` (definition description) elements.

```
1 <dl>
2   <dt>HTML</dt>
3   <dd>The markup language used to structure web content.</dd>
4   <dt>CSS</dt>
5   <dd>The language used to style HTML content.</dd>
6 </dl>
7
```

Definition lists are well suited to glossaries, metadata displays, and question-and-answer style content, situations where each entry naturally pairs a label with an explanation.

Choosing the Right List Type

- Use `<ul>` for a collection with no inherent order.
- Use `<ol>` when sequence matters, such as steps or rankings.

- Use `<dl>` when each item is a term paired with a description.
- Avoid using a list element purely to obtain indentation; use CSS for that instead.

Common Beginner Mistakes

- Placing text or other elements directly inside `<ul>` or `<ol>` without wrapping it in `<li>`.
- Using an ordered list when the sequence has no real meaning.
- Manually typing numbers or bullet characters into the text instead of relying on `<ol>` or `<ul>`.
- Nesting a list inside another list without placing it inside an `<li>` element.

Best Practices

- Select the list type based on the meaning of the content, not on its default appearance.
- Keep list markup shallow and readable; avoid unnecessary nesting.
- Use CSS, such as `list-style-type` or `list-style-image`, to customize appearance.
- Remember that `<li>` elements are only valid inside `<ul>`, `<ol>`, or `<menu>`.

Conclusions

Lists are one of the most frequently used structures in HTML, appearing in navigation menus, articles, and forms alike. Choosing between `<ul>`, `<ol>`, and `<dl>` based on the meaning of the content, rather than its visual appearance, keeps documents both easier to style and easier to understand.

In the next section, we return to `<div>` and introduce `<span>`, examining how the two elements are used together to group content for styling and scripting purposes.

4.4. Divisions and Spans for Grouping Content

Not every grouping of content has an obvious semantic label. HTML provides two generic elements, precisely for these cases: `<div>` for block-level grouping and `<span>` for inline grouping. Neither element carries any inherent meaning; their purpose is to serve as a hook for CSS styling or JavaScript behavior when no more specific element applies.

The `<div>` Element Revisited

As introduced in Section 4.1, `<div>` is a block-level container with no semantic meaning of its own. It begins on a new line and occupies the full width available to it.

```
1 <div class="card">
2   <h3>Web Technologies</h3>
3   <p>An introductory course on modern web development.</p>
4 </div>
```

A `<div>` should only be used when no semantic block-level element, such as `<section>`, `<article>`, or `<aside>`, fits the content it contains.

The `<span>` Element for Inline Grouping

`<span>` is the inline counterpart to `<div>`. It groups a portion of text or other inline content without affecting the surrounding layout, and, like `<div>`, it carries no semantic meaning on its own.

```
1 <p>
2   The total is <span class="price">49.99 RON</span>,
3   including VAT.
4 </p>
```

Here, `<span>` makes it possible to target the price with CSS, for example to display it in bold or a different color, without introducing a line break or otherwise changing how the surrounding sentence flows.

`div` vs. `span` vs. Semantic Elements

Because `<div>` and `<span>` are so flexible, it can be tempting to reach for them by default. A useful rule of thumb is to ask whether a more specific element already describes the content:

- Is this the main navigation? Use `<nav>`, not `<div>`.
- Is this a self-contained piece of content, such as a blog post? Use `<article>`, not `<div>`.
- Is this a single word or phrase that needs styling, with no semantic role? `<span>` is appropriate.
- Is this a layout wrapper with no semantic role, such as a grid column? `<div>` is appropriate.

```
1  <!-- Prefer this -->
2  <nav>
3      <a href="/">Home</a>
4      <a href="/about">About</a>
5  </nav>
6
7  <!-- Over this -->
8  <div>
9      <a href="/">Home</a>
10     <a href="/about">About</a>
11 </div>
```

Chapter 9 will return to this idea in more depth when we discuss semantic HTML and accessibility.

Using class and id for Styling and Scripting Hooks

Because `<div>` and `<span>` carry no meaning of their own, they are almost always paired with a `class` or `id` attribute that identifies their purpose.

- `class` — assigns one or more reusable labels; the same class can appear on many elements;
- `id` — assigns a single, unique identifier to one element within the page.

```
1 <div id="main-content" class="container">
2   <span class="highlight">Note:</span>
3   Registration closes on March 31.
4 </div>
```

An id must be unique within a page, while a class may be reused as many times as needed. Later chapters on CSS and JavaScript rely heavily on this distinction: id is typically used to target one specific element, while class is used to apply shared styling or behavior to a group of elements.

Common Beginner Mistakes

- Reaching for `<div>` or `<span>` before checking whether a semantic element already fits.
- Reusing the same id on more than one element in the same page.
- Wrapping a single word in `<div>` when it is inline content and `<span>` is the correct choice.
- Adding a class or id without ever using it in CSS or JavaScript.

Best Practices

- Reach for a semantic element first; use `<div>` or `<span>` only when nothing more specific applies.
- Give class and id names that describe purpose ("main-navigation") rather than appearance ("blue-box").
- Keep id values unique across the entire document.
- Avoid unnecessary nesting of `<div>` elements, sometimes called "div soup," which makes documents harder to read and maintain.

Conclusions

`<div>` and `<span>` are the general-purpose containers of HTML, indispensable when no semantic element fits, but easy to overuse. Paired with class and id attributes, they provide the hooks that CSS and JavaScript rely on to style and manipulate a page, while semantic elements remain the preferred choice whenever the content's role is well defined.

This concludes our study of text and structural elements. You now have the vocabulary needed to mark up headings, paragraphs, inline text, lists, and general-purpose containers.

In the next chapter, we turn to hyperlinks and navigation, the feature that gives the web its name: a network of documents connected to one another.

Chapter 5. Hyperlinks and Navigation

5.1. Creating Hyperlinks with the <a> Tag

The hyperlink is the feature that gives the World Wide Web its name: a web of documents connected to one another. Without links, every page would be an isolated island; with them, users can move seamlessly from one document, and one site, to another. In HTML, hyperlinks are created with the anchor element, <a>.

In this section, we will examine:

- the basic syntax of the <a> element and the href attribute;
- linking to other pages within the same site;
- opening links in a new tab;
- styling links using their interaction states.

Basic Syntax

An anchor element wraps the content that becomes clickable, which can be text, an image, or a combination of inline elements. The destination of the link is specified with the href (hypertext reference) attribute.

```
1 <a href="https://www.w3.org">Visit the W3C</a>
```

When rendered, the text between the opening and closing tags becomes a clickable link, typically underlined and colored differently from the surrounding text by default.

Linking to Other Pages on the Same Site

Links are not limited to external websites; they are just as often used to connect pages within the same project.

```
1 <nav>
2   <a href="index.html">Home</a>
3   <a href="about.html">About</a>
4   <a href="contact.html">Contact</a>
5 </nav>
```

The href attribute can point to a full web address, a file on the same server, or a location within the current page, a distinction explored further in the next two sections.

Opening Links in a New Tab

By default, clicking a link replaces the current page. The target attribute changes this behavior; setting it to "_blank" opens the linked document in a new tab or window.

```
1 <a href="https://www.w3.org" target="_blank" rel="noopener noreferrer">
2   Visit the W3C
3 </a>
```

When target="_blank" is used, it is good practice to also include rel="noopener noreferrer". This prevents the newly opened page from gaining direct access to the original window object, which is both a performance and a security consideration.

Links as More Than Text

An <a> element is not limited to wrapping text. Because it is an inline element, it can also wrap an image or a short group of inline elements, turning the entire group into a single clickable link.

```
1 <a href="profile.html">
2   
3   <span>View profile</span>
4 </a>
```

Link States and Basic Styling

Browsers apply default styling to links based on their state: unvisited, visited, and active (while being clicked). These states can be targeted individually with CSS pseudo-classes, which will be covered in the companion volume on CSS; for now, it is enough to know that a link's appearance is not fixed and is commonly restyled to match a site's design.

Common Beginner Mistakes

- Omitting the href attribute, which leaves the link non-functional.
- Using target="_blank" for every link, which disrupts normal browsing and harms accessibility when overused.
- Wrapping a block-level element, such as a whole <div> section, in an <a> without considering whether that is really the intended click target.
- Using a link where a <button> would be more appropriate, such as for an action that does not navigate to a new URL.

Best Practices

- Always include a meaningful href value.
- Use target="_blank" sparingly, and pair it with rel="noopener noreferrer" when used.
- Write descriptive link text ("Read the full report" rather than "click here"), which benefits both usability and accessibility.
- Reserve <a> for navigation, and <button> for actions that do not change the page's location.

Conclusions

The <a> element, and its href attribute, form the foundation of navigation on the web. Whether linking to another site, another page, or another part of the same document, the anchor element is what turns a collection of separate HTML files into an interconnected experience.

In the next section, we will look more closely at the values that href can take, and the difference between absolute and relative URLs.

5.2. Absolute vs. Relative URLs

The value of the href attribute is a URL (Uniform Resource Locator), and URLs can be written in two fundamentally different ways: absolute and relative. Understanding the difference is essential for building sites that are both portable and easy to maintain.

Absolute URLs

An absolute URL specifies the complete address of a resource, including the protocol (such as https://) and the domain name. It points to the same location no matter where it is used.

```
1 <a href="https://www.utcluj.ro/despre-noi">About UTCN</a>
```

Absolute URLs are necessary when linking to a resource hosted on a different domain, such as an external website or a service outside the current project.

Relative URLs

A relative URL specifies a path to a resource relative to the location of the current document. It omits the protocol and domain, and is resolved by the browser based on where the current page lives.

```
1 <!-- Assuming the current page is /courses/index.html -->
2 <a href="html-basics.html">HTML Basics</a>
3 <a href="../about.html">About the Department</a>
4 <a href="/contact.html">Contact</a>
```

Three relative forms appear in this example:

- `html-basics.html` — a file in the same folder as the current page;
- `../about.html` — the `..` segment moves up one folder before resolving the rest of the path;
- `/contact.html` — a path starting with `/` is resolved from the root of the site, regardless of the current page's location.

Choosing Between Absolute and Relative URLs

- Use relative URLs for links within your own project. Doing so allows the entire site to be moved to a new domain, or tested locally, without rewriting every link.
- Use absolute URLs for links to external sites, since a relative path cannot reach outside the current domain.

A common beginner error is hardcoding a full domain name into every internal link. This works until the site is renamed, moved to a new domain, or served from a different environment during development, at which point every internal link breaks at once.

The <base> Element

In rare cases, it can be useful to define a single base URL that all relative URLs in a document are resolved against. This is done with the <base> element, placed inside <head>.

```
1 <head>
2   <base href="https://www.example.com/docs/">
3 </head>
4 <!-- Now resolves to https://www.example.com/docs/guide.html -->
5 <a href="guide.html">Guide</a>
6
```

A document may contain at most one <base> element, and it affects every relative URL on the page, including those used by images, scripts, and stylesheets.

Common Beginner Mistakes

- Mixing up ./ (current folder) and / (site root), which resolve to very different locations.
- Hardcoding the domain name in internal links.
- Forgetting that relative paths are resolved against the current page's URL, not against the project's folder structure on disk.

Best Practices

- Use relative URLs for internal navigation whenever possible.
- Use absolute URLs only for external resources.
- Keep a consistent folder structure so relative paths remain predictable as the site grows.

Conclusions

Choosing correctly between absolute and relative URLs keeps a site portable, easier to test locally, and simpler to maintain as it grows. As a general rule, link within your own site using relative paths, and reserve absolute URLs for resources that live elsewhere.

In the next section, we will look at linking to specific resources and to specific locations within a page, using file links, download links, and anchor targets.

5.3. Linking to External Resources and Anchors

Beyond linking one page to another, HTML supports linking to specific resources, such as documents or files, and to specific locations within a page. This section covers both.

Linking to Files and Documents

The href attribute can point to any type of file, not just HTML documents. A browser will typically try to open the file directly if it can, such as with a PDF or an image.

```
1 <a href="/files/syllabus.pdf">Download the course syllabus (PDF)</a>
```

Forcing a Download with the download Attribute

By default, some file types open in the browser rather than downloading. The download attribute instructs the browser to save the file instead of displaying it.

```
1 <a href="/files/dataset.csv" download>
2   Download the dataset (CSV)
3 </a>
4 <a href="/files/report.pdf" download="annual-report.pdf">
5   Download report
6 </a>
7
```

Giving download a value, as in the second example, lets you control the filename the browser suggests when saving the file, independent of the file's name on the server.

Linking to a Specific Part of a Page: Fragment Identifiers

A URL can include a fragment identifier, a # followed by an id, which tells the browser to scroll to the element with that id after loading the page.

```
1 <h2 id="accessibility">Accessibility Considerations</h2>
2 ...
3 <a href="#accessibility">Jump to Accessibility section</a>
```

This same technique also works across pages, by combining a page URL with a fragment:

```
1 <a href="chapter6.html#accessibility">
2   Read about accessibility in Chapter 6
3 </a>
```

A fragment identifier always refers to an element's id attribute, which must be unique within the page.

"Back to Top" Links

A common use of fragment identifiers is a link back to the top of a long page.

```
1 <body id="top">
2   ...
3   <a href="#top">Back to top</a>
4 </body>
```

Email and Telephone Links

The href attribute also supports special schemes for launching other applications: mailto: for email, and tel: for telephone numbers.

```
1 <a href="mailto:office@example.utcluj.ro">Email the office</a>
2 <a href="tel:+40264401200">Call us: +40 264 401 200</a>
```

Clicking a mailto: link opens the user's default email application with the address pre-filled; a tel: link similarly offers to dial the number on devices that support phone calls.

Common Beginner Mistakes

- Forgetting that a fragment identifier requires a matching id somewhere on the target page.
- Assuming the download attribute works for cross-origin links; browsers may ignore it when the file is hosted on a different domain.
- Using a plain href="email@example.com" instead of the mailto: scheme, which does not open an email client.

Best Practices

- Use meaningful, stable id values for sections that are linked to by fragment identifiers.
- Use the download attribute when a link is meant to save a file rather than display it.
- Use mailto: and tel: links for contact information so they remain actionable on all devices.
-

Conclusions

Beyond linking between pages, HTML's anchor element can point to files, trigger downloads, jump to a specific section of a page, and even launch email or phone applications. Together, these capabilities make the `<a>` element one of the most versatile tools in HTML.

In the final section of this chapter, we will bring these ideas together to build a complete navigation menu and consider how site structure affects usability.

5.4. Navigation Menus and Site Structure

Individual links become a navigation system once they are organized consistently across a site. This final section of the chapter looks at how to structure navigation menus and how site structure affects the experience of finding content.

The `<nav>` Element

HTML5 introduced the `<nav>` element to identify a section of a page dedicated to navigation. Wrapping a group of links in `<nav>` tells browsers, search engines, and assistive technologies that this is a navigation block, not just an arbitrary collection of links.

```
1 <nav>
2   <ul>
3     <li><a href="/">Home</a></li>
4     <li><a href="/courses">Courses</a></li>
5     <li><a href="/research">Research</a></li>
6     <li><a href="/contact">Contact</a></li>
7   </ul>
8 </nav>
```

Wrapping the links in a `<ul>` is a common and recommended pattern: a navigation menu is, structurally, a list of destinations, even though it is rarely styled to look like a bulleted list.

Multiple Navigation Regions

A single page may contain more than one `<nav>` element, such as a primary menu and a separate footer navigation. When more than one `<nav>` is present, an `aria-label` attribute helps distinguish them for assistive technology.

```
1 <nav aria-label="Primary">
2   ...
3 </nav>
4 <nav aria-label="Footer">
5   ...
6 </nav>
7
```

Indicating the Current Page

It is good practice to mark which navigation item corresponds to the page currently being viewed, both for sighted users and for assistive technology. The `aria-current` attribute is the standard way to do this.

```
1 <nav>
2   <a href="/courses" aria-current="page">Courses</a>
3   <a href="/research">Research</a>
4 </nav>
```

Planning Site Structure

Before writing any markup, it helps to sketch the pages a site will need and how they relate to one another. A simple hierarchy, such as the one below, keeps navigation predictable:

- Home
- Courses
 - Course details
- Research

- Publications
- Contact

A shallow, well-organized structure like this one generally serves users better than one that buries important pages several clicks deep.

Breadcrumbs

For sites with deeper hierarchies, a breadcrumb trail helps users understand where they are and navigate back up the structure.

```
1 <nav aria-label="Breadcrumb">
2   <ol>
3     <li><a href="/">Home</a></li>
4     <li><a href="/courses">Courses</a></li>
5     <li>Web Technologies</li>
6   </ol>
7 </nav>
```

Notice that an ordered list is used here, since a breadcrumb trail represents a specific sequence, from the site's root down to the current page.

Common Beginner Mistakes

- Building a navigation menu out of unlabeled `<div>` and `<span>` elements instead of `<nav>`.
- Repeating the primary navigation markup on every page with small, unintentional inconsistencies.
- Nesting a site's structure too deeply, forcing users to click through several pages to reach common content.
- Omitting any indication of which page is currently active.

Best Practices

- Wrap navigation menus in `<nav>`, and structure the links as a list.
- Keep the primary navigation consistent across every page of a site.

- Use aria-label to distinguish multiple <nav> regions on the same page.
- Use aria-current="page" to indicate the current location.
- Favor a shallow site structure, and add breadcrumbs when depth is unavoidable.

Conclusions

Good navigation is what turns a set of individual pages into a coherent site. The <nav> element, thoughtful use of lists, and a shallow, consistent site structure all work together to help users find what they are looking for with a minimum of friction.

In the next chapter, we move from text and navigation to visual and multimedia content, examining how HTML embeds images, audio, and video.

Chapter 6. Working with Images and Multimedia

6.1. Embedding Images Using

Text alone rarely tells the whole story. Photographs, diagrams, icons, and illustrations are a core part of most web pages, and HTML embeds them with the element.

In this section, we will examine:

- the basic syntax of the element;
- supported image formats and when to use each;
- the difference between embedding an image and linking to one;
- the <figure> and <figcaption> elements for captioned images.

Basic Syntax

The element is a void element; it never has a closing tag or wraps content. It requires, at minimum, a src attribute pointing to the image file.

```
1 
```

As with links, the src attribute accepts both relative and absolute URLs, following the same rules covered in the previous chapter.

Common Image Formats

- JPEG (.jpg) — best for photographs; uses lossy compression;
- PNG (.png) — supports transparency; best for graphics, logos, and screenshots;
- SVG (.svg) — a vector format described in XML; scales to any size without losing quality; ideal for icons and simple illustrations;

- WebP (.webp) — a modern format offering strong compression with quality comparable to JPEG and PNG, supported by all major browsers.

Choosing the right format is a balance between visual quality, file size, and whether the image needs to scale or animate.

Embedding vs. Linking

It is important to distinguish between embedding an image with `<img>` and linking to an image file with `<a>`. An `<img>` element displays the image directly on the page; an `<a>` element, as seen in the previous chapter, only creates a clickable reference to it.

```
1 <!-- Displays the image on the page -->
2 
3
4 <!-- Displays a clickable link to the image file -->
5 <a href="diagram.png">Open the network diagram</a>
```

The two techniques are often combined, wrapping an `<img>` in an `<a>`, to display a thumbnail that links to a full-size version of the same image.

```
1 <a href="diagram-full.png">
2   
3 </a>
```

Captioning Images with `<figure>` and `<figcaption>`

When an image needs an accompanying caption, HTML5 provides the `<figure>` and `<figcaption>` elements, which associate the caption with the image in a semantically meaningful way.

```
1 <figure>
2   
3   <figcaption>
4     The main campus of the Technical University of
5     Cluj-Napoca.
6   </figcaption>
7 </figure>
```

<figure> is not limited to images; it can also wrap a table, a code listing, or a diagram, whenever the content is referenced as a unit and benefits from a caption.

Common Beginner Mistakes

- Forgetting the alt attribute, covered in depth in the next section.
- Using an oversized source image and relying on width and height alone to shrink it, which wastes bandwidth.
- Using <figcaption> outside of a <figure>, where it has no defined meaning.
- Uploading an image in a format poorly suited to its content, such as a screenshot saved as a highly compressed JPEG, which introduces visible artifacts.

Best Practices

- Choose an image format appropriate to the content: JPEG for photos, PNG or SVG for graphics and icons.
- Serve images at a size close to their intended display size.
- Use <figure> and <figcaption> for any image that needs a caption.
- Keep image files organized in a dedicated folder within the project.

Conclusions

The element is the starting point for including visual content in a web page. Choosing an appropriate format and using <figure> and <figcaption> when a caption is needed lay the groundwork for images that are both well presented and well structured.

In the next section, we will look closely at the attributes that make images accessible and properly sized: alt, title, width, and height.

6.2. Image Attributes: Alt Text, Titles, and Sizing

An `<img>` element does much more than point to a file. Its attributes describe the image to users who cannot see it, and to the browser rendering it, and they control how much space the image reserves on the page.

The alt Attribute

The alt attribute provides a text alternative for an image. It is read aloud by screen readers, displayed if the image fails to load, and used by search engines to understand the image's content.

```
1 
```

The alt attribute is not optional for meaningful content. A missing alt attribute leaves screen reader users with no information about the image at all, while an empty one (`alt=""`) explicitly signals that the image is decorative and should be skipped.

```
1 <!-- Decorative image: safe to skip -->
2 
3
4 <!-- Meaningful image: needs a real description -->
5 
```

Writing Good Alt Text

- Describe the content and function of the image, not that it is an image ("a photo of" is usually unnecessary).
- Keep it concise; a long, detailed description belongs in surrounding text or a linked description instead.
- For an image that is also a link, describe the link's destination, not just the picture.
- Leave alt empty for purely decorative images.

The title Attribute

The title attribute provides supplementary information, typically shown as a tooltip when the user hovers over the element. Unlike alt, it is not a substitute for a text alternative and is not reliably announced by assistive technology.

```
1 
```

Setting width and height

The width and height attributes specify an image's intended display dimensions, in pixels.

```
1 
```

Specifying width and height allows the browser to reserve the correct amount of space for the image before it finishes loading, preventing the rest of the page from shifting once the image appears.

This layout stability is significant enough to be measured directly as part of a page's Core Web Vitals, under the Cumulative Layout Shift metric.

Responsive Sizing with CSS

The width and height attributes should describe the image's natural aspect ratio; they are not the right tool for making an image responsive to different screen sizes. That responsibility belongs to CSS, typically with a rule such as:

```
1 img {
2   max-width: 100%;
3   height: auto;
4 }
```

This CSS rule lets the width and height attributes continue to reserve the correct aspect ratio, while allowing the image to shrink on narrower screens.

Common Beginner Mistakes

- Leaving out alt entirely, rather than using an empty value for decorative images.
- Writing alt text that simply repeats the file name.
- Using title as a replacement for alt; the two serve different purposes and are not interchangeable.
- Stretching an image by setting width and height to values that do not match its natural aspect ratio.

Best Practices

- Always include alt; leave it empty only for decorative images.
- Write alt text that conveys meaning, not just that an image is present.
- Always set width and height to the image's natural pixel dimensions to prevent layout shift.
- Use CSS, not the HTML attributes, to make images responsive.

Conclusions

Alt text, titles, and sizing attributes together determine whether an image is accessible, informative, and well-behaved within the page layout. Of the three, alt text deserves the most care: it is often the only description a user receives of an image's content.

In the next section, we turn from static images to audio and video, and the HTML5 elements that embed them natively.

6.3. Incorporating Audio and Video Elements

Before HTML5, embedding audio or video in a web page required third-party browser plugins such as Flash. HTML5 introduced native `<audio>` and `<video>` elements, removing that dependency and giving browsers direct control over media playback.

The `<audio>` Element

The `<audio>` element embeds a sound file, with the `controls` attribute displaying the browser's built-in playback interface.

```
1 <audio src="lecture-01.mp3" controls>
2   Your browser does not support the audio element.
3 </audio>
```

Text placed between the opening and closing tags, as in the example above, is only displayed as fallback content by browsers that do not support the element at all, which is extremely rare today.

The <video> Element

The <video> element works similarly, and additionally accepts width, height, and poster attributes to control its display area and its preview image.

```
1 <video src="intro.mp4" controls width="640" height="360"
2       poster="intro-thumbnail.jpg">
3   Your browser does not support the video element.
4 </video>
```

Supplying Multiple Formats with <source>

Not every browser supports every media format. Instead of a single src attribute, both <audio> and <video> accept multiple <source> child elements, letting the browser choose the first format it can play.

```
1 <video controls width="640" height="360">
2   <source src="intro.webm" type="video/webm">
3   <source src="intro.mp4" type="video/mp4">
4   Your browser does not support the video element.
5 </video>
```

Common Playback Attributes

- controls — displays the browser's built-in playback interface;
- autoplay — begins playback automatically (browsers generally require the media to also be muted for autoplay to work);

- `loop` — restarts playback automatically when it ends;
- `muted` — starts the media without sound;
- `preload` — hints whether the browser should load none, metadata, or the entire file in advance.

```
1 <video src="background-loop.mp4" autoplay muted loop playsinline>
2 </video>
```

Autoplaying video with sound is blocked by most browsers, and autoplaying any media without a clear reason is generally discouraged, since it can be disruptive and consumes bandwidth the user may not have chosen to spend.

Embedding Video from External Platforms

Video hosted on platforms such as YouTube or Vimeo is typically embedded with an `<iframe>` rather than a `<video>` element, using an embed URL supplied by the platform.

```
1 <iframe width="640" height="360"
2     src="https://www.youtube.com/embed/VIDEO_ID"
3     title="Lecture recording"
4     allowfullscreen>
5 </iframe>
```

In this case, the platform handles encoding, streaming, and player controls; the page only embeds a frame pointing to it.

Common Beginner Mistakes

- Relying on a single media format without a fallback `<source>`, which can leave some visitors unable to play the file.
- Using autoplay with sound, which most browsers block outright.
- Forgetting fallback text for the rare browser without native support.
- Omitting width and height on `<video>`, which can cause the layout to shift once the video loads.

Best Practices

- Provide at least one broadly supported format, such as MP4 for video or MP3 for audio.
- Always include the controls attribute unless there is a specific, deliberate reason to omit it.
- Use autoplay only when muted, and only when it clearly serves the user.
- Set explicit width and height on <video> to avoid layout shift.

Conclusions

The <audio> and <video> elements bring native media playback to HTML, without requiring any third-party plugin. Used thoughtfully, with sensible fallback formats and restrained use of autoplay, they make multimedia content a first-class part of the page.

In the final section of this chapter, we will look specifically at accessibility considerations for multimedia, including captions, transcripts, and audio descriptions.

6.4. Accessibility Considerations for Multimedia

Images, audio, and video each present their own accessibility challenges. This section brings together the practices covered throughout the chapter and extends them to captions, transcripts, and audio descriptions for time-based media.

Why Multimedia Accessibility Matters

A user who is blind or has low vision cannot see an image without a text alternative. A user who is deaf or hard of hearing cannot follow spoken audio without captions or a transcript. A user with a slow or unreliable connection may not be able to load video at all, and relies on text to understand what it contains. Accessible multimedia serves all of these situations at once.

Captions for Video

Captions display a synchronized text transcript of a video's audio, including spoken dialogue and relevant sound effects. HTML5 supports captions through the <track> element, using the WebVTT format.

```
1 <video src="lecture-01.mp4" controls width="640" height="360">
2   <track kind="captions" src="lecture-01-en.vtt"
3       srclang="en" label="English" default>
4 </video>
```

A WebVTT file is a plain text file that pairs timestamps with the corresponding caption text:

```
1 WEBVTT
2
3 00:00:00.000 --> 00:00:04.000
4 Welcome to the first lecture on Web Technologies.
5
6 00:00:04.500 --> 00:00:08.000
7 Today we will look at how HTML structures content.
```

Subtitles vs. Captions

- Subtitles assume the viewer can hear the audio and mainly need a translation of the spoken dialogue;
- Captions assume the viewer cannot hear the audio and additionally describe relevant non-speech sound, such as [applause] or [phone ringing].

The track element's kind attribute distinguishes the two: kind="subtitles" or kind="captions".

Transcripts

A transcript is a full text version of a piece of audio or video content, presented alongside it rather than synchronized to it. Transcripts benefit users who prefer reading, users on slow connections, and search engines, which cannot index audio or video content directly.

```
1 <video src="lecture-01.mp4" controls></video>
2 <details>
3   <summary>Read the transcript</summary>
4   <p>Welcome to the first lecture on Web Technologies...</p>
5 </details>
```

Audio Descriptions

For video content where important information is conveyed visually, such as on-screen text or an action with no accompanying dialogue, an audio description narrates that information aloud. This can be provided as an alternate audio track, or, for simpler content, folded directly into the original narration.

Bringing It Together

A well-produced piece of multimedia content typically combines several of the techniques covered in this chapter:

- meaningful alt text for any accompanying images;
- captions for spoken dialogue and important sound;
- a transcript for users who prefer or require text;
- audio description when the visuals convey essential information not present in the dialogue.

Common Beginner Mistakes

- Publishing video with no captions or transcript at all.
- Auto-playing video with sound, which can be jarring for users relying on screen readers running at the same time.
- Treating captions and subtitles as interchangeable, and omitting non-speech sound cues that captions are meant to include.
- Assuming accessibility features only matter for a small number of users, when in practice they also benefit anyone in a noisy environment, a quiet one, or with an unreliable connection.

Best Practices

- Treat captions as a standard part of publishing video, not an optional extra.
- Provide a transcript for any substantial audio or video content.
- Use the <track> element rather than burning captions directly into the video image, so they can be toggled, translated, or read by assistive technology.
- Test multimedia content with the sound off and with a screen reader to catch gaps early.

Conclusions

Accessible multimedia is not a separate feature bolted onto a page; it is the result of consistently applying the practices covered throughout this chapter, meaningful alt text, captions, transcripts, and thoughtful use of autoplay, to every image, audio clip, and video a page includes.

This concludes our chapter on images and multimedia. You now know how to embed and caption images, incorporate native audio and video, and make multimedia content accessible to the widest possible audience.

In the next chapter, we turn to tables and the presentation of tabular data.

Chapter 7. Tables and Data Presentation

7.1. Building Tables with `<table>`, `<tr>`, `<th>`, and `<td>`

Some information is inherently tabular: exam schedules, price lists, comparison charts, and datasets all consist of rows and columns of related values. HTML represents this kind of content with the `<table>` element and a small family of related tags.

In this section, we will examine:

- the core structure of a table: `<table>`, `<tr>`, `<th>`, and `<td>`;
- the difference between header cells and data cells;
- spanning cells across multiple rows or columns;
- when a table is, and is not, the right element to use.

The Basic Table Structure

A table is built from three elements working together:

- `<table>` — the container for the entire table;
- `<tr>` — a table row;
- `<td>` — a standard data cell (table data);
- `<th>` — a header cell, used to label a row or column.

```
1 <table>
2   <tr>
3     <th>Course</th>
4     <th>Credits</th>
5   </tr>
6   <tr>
7     <td>Web Technologies</td>
8     <td>5</td>
9   </tr>
10  <tr>
11    <td>Microelectronics</td>
12    <td>6</td>
13  </tr>
14 </table>
```

Each `<tr>` defines one row, and the `<th>` or `<td>` elements inside it define the cells of that row, read from left to right.

Header Cells and the scope Attribute

A `<th>` element is not merely a bold `<td>`; it carries semantic meaning, identifying a cell as a header for the row or column it introduces. This meaning is exposed to screen readers, which can announce the relevant header as a user navigates through data cells.

The scope attribute makes the relationship explicit: `scope="col"` for a column header, `scope="row"` for a row header.

```

1 <table>
2   <tr>
3     <th scope="col">Course</th>
4     <th scope="col">Credits</th>
5   </tr>
6   <tr>
7     <th scope="row">Web Technologies</th>
8     <td>5</td>
9   </tr>
10 </table>

```

Spanning Rows and Columns

Sometimes a single cell needs to span more than one row or column. The `colspan` and `rowspan` attributes control this.

```

1 <table>
2   <tr>
3     <th>Semester</th>
4     <th colspan="2">Exam Period</th>
5   </tr>
6   <tr>
7     <td rowspan="2">Fall</td>
8     <td>January</td>
9     <td>February</td>
10  </tr>
11 </table>

```

Here, `colspan="2"` makes the "Exam Period" header stretch across two columns, while `rowspan="2"` would make "Fall" stretch across two rows if a second row followed.

When to Use a Table

Tables should be used for genuinely tabular data, and never purely to control page layout. Using tables for layout, common in the early days of the web, produces

markup that is difficult to maintain and confusing for screen reader users, who expect a table to represent structured data with rows and columns of meaning.

Common Beginner Mistakes

- Using `<td>` everywhere and never using `<th>` for headers.
- Using tables to lay out an entire page instead of using CSS.
- Forgetting that `colspan` and `rowspan` change how many cells are expected in each row, which can silently misalign a table if miscounted.
- Leaving out `scope` on header cells in more complex tables.

Best Practices

- Use `<th>` for every header cell, with an appropriate `scope`.
- Reserve `<table>` for data that is genuinely tabular.
- Keep the number of cells consistent across rows, accounting for any `colspan` or `rowspan`.
- Keep tables as simple as the data allows; deeply nested or highly irregular tables are hard for everyone to follow.

Conclusions

The `<table>`, `<tr>`, `<th>`, and `<td>` elements provide the structural foundation for tabular data on the web. Using `<th>` correctly, with `scope` where useful, keeps tables readable both visually and through assistive technology.

In the next section, we will extend this structure with captions, header and footer groups, and the elements that make larger tables easier to understand.

7.2. Enhancing Tables with Captions, Headers, and Footers

Beyond the basic grid of rows and cells, HTML provides several elements that add structure and meaning to larger or more complex tables.

Adding a Caption with <caption>

The <caption> element provides a title for a table, announced by screen readers and displayed, by default, above the table. It must be the first child of <table>.

```
1 <table>
2   <caption>Course Credits, Fall Semester</caption>
3   <tr>
4     <th scope="col">Course</th>
5     <th scope="col">Credits</th>
6   </tr>
7   <tr>
8     <td>Web Technologies</td>
9     <td>5</td>
10  </tr>
11 </table>
```

Grouping Rows: <thead>, <tbody>, and <tfoot>

Larger tables benefit from being divided into three regions: a header, a body, and, when needed, a footer.

- <thead> — contains the header row or rows;
- <tbody> — contains the main data rows;
- <tfoot> — contains footer rows, such as totals or summaries.

```

1 <table>
2   <caption>Quarterly Sales (thousands RON)</caption>
3   <thead>
4     <tr>
5       <th scope="col">Quarter</th>
6       <th scope="col">Revenue</th>
7     </tr>
8   </thead>
9   <tbody>
10    <tr><td>Q1</td><td>120</td></tr>
11    <tr><td>Q2</td><td>135</td></tr>
12  </tbody>
13  <tfoot>
14    <tr><td>Total</td><td>255</td></tr>
15  </tfoot>
16 </table>

```

Dividing a table into <thead>, <tbody>, and <tfoot> not only clarifies its structure, it also allows a browser to keep the header visible while the body scrolls, and lets a printed version repeat the header row on every page.

Grouping Columns: <colgroup> and <col>

Just as rows can be grouped, columns can be described in advance using <colgroup> and <col>, most often to apply shared styling to an entire column without repeating it on every cell.

```

1 <table>
2   <colgroup>
3     <col>
4     <col style="background-color: #f5f5f5;">
5   </colgroup>
6   <tr>
7     <th scope="col">Course</th>
8     <th scope="col">Credits</th>
9   </tr>
10  <tr>
11    <td>Web Technologies</td>
12    <td>5</td>
13  </tr>
14 </table>

```

Common Beginner Mistakes

- Placing <caption> anywhere other than immediately after the opening <table> tag.
- Using <tfoot> for a row that is not actually a summary or total.
- Forgetting that <thead>, <tbody>, and <tfoot> each expect complete <tr> rows inside them, not loose cells.
- Adding <colgroup> without any real need for shared column styling.

Best Practices

- Add a <caption> whenever a table's purpose is not obvious from context.
- Use <thead>, <tbody>, and <tfoot> for any table long enough to benefit from a repeated header.
- Reserve <tfoot> for genuine summary rows.
- Use <colgroup> only when styling or grouping entire columns is actually needed.

Conclusions

Captions, header and footer groups, and column groups turn a simple grid of cells into a well-organized, more accessible table. These elements cost little to add and pay off significantly on any table long or complex enough to need them.

In the next section, we turn to styling tables for readability, and the CSS techniques most commonly used with tabular data.

7.3. Styling Tables for Readability

An unstyled HTML table is functional but visually plain: thin default borders, cramped spacing, and no visual distinction between header and data rows. This section introduces the CSS techniques most commonly applied to tables to make them easier to read at a glance.

A full treatment of CSS is outside the scope of this volume and is covered in the companion book; the goal here is to show how tables, specifically, are commonly styled.

Borders and border-collapse

By default, each cell in an HTML table draws its own separate border, which produces a doubled-line effect. The `border-collapse` property merges adjacent borders into a single line.

```
1 table {  
2     border-collapse: collapse;  
3     width: 100%;  
4 }  
5 th, td {  
6     border: 1px solid #cccccc;  
7     padding: 8px 12px;  
8 }  
9
```

Aligning Content

Numeric data is generally easier to compare when right-aligned, while text is usually left-aligned. Header cells are commonly centered or left-aligned depending on the design.

```
1 th {
2   text-align: left;
3   background-color: #f0f0f0;
4 }
5 td.numeric {
6   text-align: right;
7 }
8
```

Zebra Striping

Alternating the background color of rows, sometimes called zebra striping, makes it easier to follow a row across many columns without losing your place. This is typically done with the `:nth-child` selector.

```
1 tbody tr:nth-child(even) {
2   background-color: #f9f9f9;
3 }
```

Highlighting Rows on Hover

A subtle hover effect can help users track a row visually as they read across a wide table.

```
1 tbody tr:hover {
2   background-color: #eef6fb;
3 }
```

Responsive Tables

Wide tables can overflow the width of a small screen. A common solution is to wrap the table in a scrollable container, rather than shrinking the table itself to the point of illegibility.

```
1 <div class="table-scroll">
2   <table>
3     ...
4   </table>
5 </div>
```

```
1 .table-scroll {
2   overflow-x: auto;
3 }
```

Common Beginner Mistakes

- Leaving border-collapse at its default value, producing unintentionally doubled borders.
- Center-aligning long blocks of text, which makes them harder to read than left-aligned text.
- Styling a table so heavily that its underlying structure becomes hard to perceive.
- Letting a wide table overflow the page with no scrolling container on small screens.

Best Practices

- Use border-collapse: collapse for a cleaner default appearance.
- Align numeric columns to the right and text columns to the left.
- Use zebra striping or hover states sparingly, to aid readability without adding visual noise.
- Wrap wide tables in a scrollable container for small screens.

Conclusions

A handful of CSS rules, collapsed borders, clear alignment, subtle striping, go a long way toward making a table easy to scan. The underlying HTML structure from the previous two sections is what makes this styling possible in the first place.

In the final section of this chapter, we will step back and summarize best practices for displaying tabular data as a whole.

7.4. Best Practices for Displaying Tabular Data

This closing section brings together the guidance from the rest of the chapter into a single, practical checklist for working with tables.

Choosing a Table in the First Place

- Use a table when the data has two or more meaningful dimensions, such as rows of items and columns of attributes.
- Do not use a table simply to achieve a particular visual layout; use CSS for that instead.
- For a single list of items with no meaningful columns, a list (`<ul>` or `<ol>`) is usually a better fit than a table.

A Structure Checklist

- Every table has a clear caption or is introduced by surrounding text explaining what it shows.
- Header cells use `<th>`, with scope set for anything beyond a trivial table.
- Rows are grouped into `<thead>`, `<tbody>`, and, when relevant, `<tfoot>`.
- `colspan` and `rowspan` are used only where cells genuinely span multiple rows or columns.

An Accessibility Checklist

- Every data cell can be understood in relation to its header, either visually or through scope.

- Complex tables, with multiple levels of headers, use the headers and id attributes to associate cells explicitly when scope alone is not enough.
- Tables are tested by tabbing and reading through them with a screen reader, not just by looking at them.

A Simple Example, Done Well

```
1 <table>
2   <caption>Final Grades, Web Technologies</caption>
3   <thead>
4     <tr>
5       <th scope="col">Student ID</th>
6       <th scope="col">Project</th>
7       <th scope="col">Exam</th>
8       <th scope="col">Final Grade</th>
9     </tr>
10  </thead>
11  <tbody>
12    <tr>
13      <th scope="row">2024117</th>
14      <td>9</td>
15      <td>8</td>
16      <td>8.5</td>
17    </tr>
18  </tbody>
19 </table>
```

This table declares its purpose with a caption, labels both its columns and its rows with `<th>`, and groups its header and body rows, everything covered in this chapter, applied together.

Conclusions

Tables remain the correct tool for genuinely tabular data, and a well-structured table, captioned, properly headed, and sensibly styled, serves sighted users, screen reader users, and print output alike.

This concludes our chapter on tables and data presentation. You now have the tools to structure, enhance, style, and validate tabular content in HTML.

In the next chapter, we turn to one of the most interactive parts of HTML: forms, and how they collect input from users.

Chapter 8. HTML Forms and User Interaction

8.1. Creating Forms with the <form> Tag

So far, every element in this book has displayed content to the user. Forms are different: they collect information from the user and send it somewhere for processing. Every login screen, search box, and registration page on the web relies on the HTML <form> element.

In this section, we will examine:

- the structure of the <form> element;
- the action and method attributes;
- the difference between the GET and POST methods;
- how a form is submitted.

The <form> Element

A <form> element wraps every input control that belongs to a single submission, from text fields to buttons.

```
1 <form action="/register" method="post">
2   <!-- form controls go here -->
3 </form>
```

The action Attribute

The action attribute specifies the URL that will receive the form's data when it is submitted. If omitted, the form submits to the current page's own URL.

```
1 <form action="https://example.com/api/register" method="post">
2 </form>
```

The method Attribute: GET vs. POST

The method attribute determines how the form's data is sent to the server.

- GET appends the form data to the URL as a query string. It is appropriate for forms that retrieve data, such as a search box, and produces a URL that can be bookmarked or shared.
- POST sends the form data in the body of the HTTP request, invisibly to the URL. It is appropriate for forms that create, update, or delete data, or that submit sensitive information.

```
1 <!-- GET: appropriate for a search form -->
2 <form action="/search" method="get">
3   <input type="text" name="q">
4   <button type="submit">Search</button>
5 </form>
6 <!-- Resulting URL: /search?q=html+forms -->
7
```

Never use GET for a form that submits a password or other sensitive information; the data would appear directly in the URL, and typically ends up stored in browser history and server logs.

Submitting a Form

A form is submitted either by clicking a submit button or, in most browsers, by pressing Enter while focus is inside a text field.

```
1 <form action="/login" method="post">
2   <input type="text" name="username">
3   <input type="password" name="password">
4   <button type="submit">Log In</button>
5 </form>
```

When submitted, the browser gathers the value of every named form control and sends it to the URL specified by action, using the method specified by method.

Common Beginner Mistakes

- Using `method="get"` for a form that submits sensitive or state-changing data.
- Forgetting the `name` attribute on an input, which prevents its value from being submitted at all.
- Nesting one `<form>` inside another, which is not permitted in HTML.
- Omitting `action` and assuming the form does nothing, when in fact it submits to the current page.

Best Practices

- Choose GET or POST based on the nature of the request, not habit.
- Always give every form control a meaningful `name` attribute.
- Keep a page to a single `<form>` per distinct action, rather than combining unrelated inputs into one form.

Conclusions

The `<form>` element, together with its `action` and `method` attributes, defines where and how user input is sent. Everything discussed in the rest of this chapter, text fields, checkboxes, dropdowns, and buttons, exists inside this basic container.

In the next section, we will look at the individual form controls available in HTML: inputs, text areas, selects, and buttons.

8.2. Form Elements: Input, Textarea, Select, and Button

HTML provides a range of form controls suited to different kinds of input, from a single line of text to a choice among several options.

The `<input>` Element and Its `type` Attribute

The `<input>` element is the most versatile form control. Its `type` attribute determines how it behaves and what kind of value it collects.

- `text` — a single line of free text;

- password — a single line of text, masked on screen;
- email — a single line of text, validated as an email address;
- number — a numeric value, often shown with increment controls;
- date — a date, typically shown with a calendar picker;
- checkbox — an independent on/off option;
- radio — one choice among a mutually exclusive group;
- file — a file selected from the user's device;
- hidden — a value submitted with the form but not shown to the user.

```
1 <label for="email">Email address</label>
2 <input type="email" id="email" name="email" required>
```

Checkboxes and Radio Buttons

Checkboxes allow independent, multiple selections; radio buttons restrict the user to one choice within a group sharing the same name.

```
1 <p>Select your favorite topics:</p>
2 <input type="checkbox" id="html" name="topics" value="html">
3 <label for="html">HTML</label>
4
5 <input type="checkbox" id="css" name="topics" value="css">
6 <label for="css">CSS</label>
7
8 <p>Preferred contact method:</p>
9 <input type="radio" id="contact-email" name="contact" value="email">
10 <label for="contact-email">Email</label>
11
12 <input type="radio" id="contact-phone" name="contact" value="phone">
13 <label for="contact-phone">Phone</label>
```

Radio buttons only behave as a mutually exclusive group when they share the same name attribute.

Multi-line Text with <textarea>

For input longer than a single line, such as a comment or message, <textarea> is used instead of <input>. Unlike <input>, it is not a void element; its default content goes between the opening and closing tags.

```
1 <label for="message">Message</label>
2 <textarea id="message" name="message" rows="5" cols="40"></textarea>
```

Dropdown Menus with <select>

The <select> element presents a list of choices, defined with <option>, in a compact dropdown.

```
1 <label for="department">Department</label>
2 <select id="department" name="department">
3   <option value="">Choose one</option>
4   <option value="ae">Applied Electronics</option>
5   <option value="cs">Computer Science</option>
6   <option value="ee">Electrical Engineering</option>
7 </select>
```

Adding the multiple attribute to <select> allows more than one option to be chosen at once; grouping related options with <optgroup> keeps a long list organized.

```
1 <select name="department">
2   <optgroup label="Engineering">
3     <option value="ae">Applied Electronics</option>
4     <option value="ee">Electrical Engineering</option>
5   </optgroup>
6   <optgroup label="Sciences">
7     <option value="cs">Computer Science</option>
8   </optgroup>
9 </select>
```

The <button> Element

The <button> element creates a clickable button, whose behavior depends on its type attribute.

- type="submit" — submits the enclosing form (the default when type is omitted);
- type="reset" — resets every form control to its initial value;
- type="button" — performs no default action, intended for use with JavaScript.

```
1 <button type="submit">Register</button>
2 <button type="reset">Clear form</button>
```

Because type="submit" is the default, a <button> placed inside a form without an explicit type will submit that form, sometimes unintentionally.

Labeling Form Controls with <label>

Every form control should have an associated <label>, connected either by wrapping the control or through the for and id attributes, as seen in the previous examples. A label increases the clickable area of a control and is read aloud by screen readers when the control receives focus.

```
1 <!-- Wrapping method -->
2 <label>
3   Username
4   <input type="text" name="username">
5 </label>
6 <!-- for/id method -->
7 <label for="username">Username</label>
8 <input type="text" id="username" name="username">
9
```

Common Beginner Mistakes

- Using <input type="text"> for everything, even when a more specific type such as email, number, or date is available.

- Giving grouped radio buttons different name attributes, breaking the mutual exclusivity.
- Leaving a form control without any associated `<label>`.
- Placing a `<button>` inside a form without considering that it defaults to `type="submit"`.

Best Practices

- Choose the most specific `<input>` type available for each field.
- Always associate a `<label>` with every form control.
- Group related radio buttons under a shared name, and, where appropriate, a `<fieldset>` (covered in the next section).
- Be explicit about a button's type rather than relying on the default.

Conclusions

Between `<input>`, `<textarea>`, `<select>`, and `<button>`, HTML provides controls for nearly any kind of user input. Choosing the right control, and labeling it properly, is the foundation of a usable form.

In the next section, we will look at additional form attributes and how form data is structured when it is submitted.

8.3. Form Attributes and Data Submission

Beyond the individual controls covered in the previous section, several attributes shape how a form behaves as a whole, and how its data is packaged when submitted.

The name and value Attributes

Only a form control with a name attribute is included in the submitted data. The name becomes the key, and the control's current content becomes the value.

```

1 <input type="text" name="fullName" value="">
2
3 <!-- Submitted (GET) as: ?fullName=Ana+Popescu -->

```

Grouping Related Fields with <fieldset> and <legend>

The <fieldset> element groups related controls together, with <legend> providing a caption for the group. This is especially useful for a group of radio buttons or checkboxes.

```

1 <fieldset>
2   <legend>Preferred contact method</legend>
3   <input type="radio" id="c-email" name="contact" value="email">
4   <label for="c-email">Email</label>
5   <input type="radio" id="c-phone" name="contact" value="phone">
6   <label for="c-phone">Phone</label>
7 </fieldset>
8

```

The placeholder and required Attributes

placeholder displays hint text inside an empty field, and disappears once the user starts typing; required prevents the form from being submitted until the field has a value.

```

1 <input type="email" name="email" placeholder="name@example.com" required>

```

A placeholder is not a substitute for a <label>. It disappears as soon as the user types, and is not a reliable way to identify a field for assistive technology.

Disabling and Read-Only Fields

disabled removes a control from both interaction and submission entirely; readonly allows the value to be seen and submitted, but not edited.

```

1 <input type="text" name="plan" value="Standard" readonly>
2 <input type="text" name="referralCode" disabled>

```

How Form Data Is Encoded

By default, form data is encoded as `application/x-www-form-urlencoded`, a simple key-value format. When a form includes a file upload, this default is not sufficient, and the `enctype` attribute must be set explicitly.

```
1 <form action="/upload" method="post" enctype="multipart/form-data">
2   <label for="cv">Upload your CV</label>
3   <input type="file" id="cv" name="cv">
4   <button type="submit">Upload</button>
5 </form>
```

A form with `type="file"` almost always needs `enctype="multipart/form-data"`; without it, the file's contents are not actually transmitted.

Common Beginner Mistakes

- Relying on placeholder text in place of a real label.
- Forgetting `enctype="multipart/form-data"` on a form that uploads files.
- Confusing `disabled` (excluded from submission) with `readonly` (included, but not editable).
- Leaving related checkboxes or radio buttons outside of any `<fieldset>`, making the group harder to perceive.

Best Practices

- Use `<fieldset>` and `<legend>` to group related controls.
- Use placeholder only as a hint, never as a replacement for a label.
- Set `enctype="multipart/form-data"` whenever a form includes a file input.
- Choose `disabled` or `readonly` deliberately, based on whether the value should be submitted.

Conclusions

The attributes covered in this section, `name`, `fieldset`, `placeholder`, `required`, and `enctype` among them, determine how a form's data is organized, grouped, and packaged for

submission. Together with the elements from the previous section, they cover the great majority of forms encountered in everyday web development.

In the final section of this chapter, we will look at basic client-side validation, and how HTML alone can catch many common input errors before a form is ever submitted.

8.4. Basic Client-Side Validation Techniques

Validating user input, checking that it is complete, well-formed, and within expected limits, is essential to any form. HTML provides a first layer of validation directly in markup, before any JavaScript or server-side code runs.

Client-side validation improves the user experience by giving immediate feedback, but it is not a security measure. Server-side validation is still required, since client-side checks can always be bypassed.

The required Attribute

As seen in the previous section, `required` prevents submission until a field has a value. Browsers automatically display a message and focus the first invalid field when submission is attempted.

```
1 <label for="name">Full name</label>
2 <input type="text" id="name" name="name" required>
```

Type-Based Validation

Choosing a specific input type, as covered in Section 8.2, also enables automatic validation. An input of `type="email"` rejects submission unless the value looks like a valid email address; `type="number"` only accepts numeric input.

```
1 <input type="email" name="email" required>
2 <input type="number" name="age" min="0" max="120">
```

Pattern Matching with the pattern Attribute

For validation rules not covered by a built-in type, the pattern attribute accepts a regular expression that the field's value must match.

```
1 <label for="studentId">Student ID</label>
2 <input type="text" id="studentId" name="studentId"
3     pattern="[0-9]{7}"
4     title="Enter a 7-digit student ID">
```

The title attribute, in this context, supplies the message shown to the user when the pattern is not matched.

Length and Range Constraints

- minlength and maxlength — the minimum and maximum number of characters accepted by a text field;
- min and max — the minimum and maximum value accepted by a numeric or date field;
- step — the increment allowed for a numeric field.

```
1 <input type="password" name="password" minlength="8" required>
2 <input type="number" name="quantity" min="1" max="10" step="1">
```

Styling Valid and Invalid Fields

The :valid and :invalid CSS pseudo-classes match form controls based on their current validity, making it possible to give the user visual feedback as they type.

```
1 input:invalid {  
2   border-color: #c0392b;  
3 }  
4 input:valid {  
5   border-color: #2e8b57;  
6 }  
7
```

The Limits of HTML Validation

HTML validation checks form, not meaning: it can confirm that a value looks like an email address or falls within a numeric range, but it cannot confirm that an email address actually exists, or that a username is not already taken. Those checks require server-side logic, often assisted by JavaScript for a smoother experience in the browser.

Common Beginner Mistakes

- Treating HTML validation as sufficient on its own, without any server-side check.
- Writing an overly strict pattern that rejects legitimate input, such as valid but unusual email addresses.
- Forgetting to provide a title on a pattern-constrained field, leaving the user without guidance when validation fails.
- Disabling built-in validation with `novalidate` without providing an equivalent replacement.

Best Practices

- Use the most specific input type and built-in constraints available before reaching for a custom pattern.
- Always pair client-side validation with server-side validation.
- Provide clear guidance, through title or nearby text, for any field with a non-obvious pattern.
- Test forms with both valid and deliberately invalid input.

Conclusions

HTML's built-in validation attributes, required, type, pattern, and the length and range constraints, catch a large share of common input errors without a single line of JavaScript. They are a first line of defense, not a complete solution, but a well-validated form is more pleasant to use and produces cleaner data for whatever processes it next.

This concludes our chapter on HTML forms. You now know how to build a form, choose appropriate controls, structure and label them, and validate input directly in the browser.

In the next chapter, we turn to semantic HTML and accessibility in more depth, building directly on ideas introduced throughout this book.

Chapter 9. Semantic HTML and Accessibility

9.1. The Importance of Semantic Markup

Throughout this book, we have occasionally reached for `<div>` and `<span>` when no more specific element fit the content. This chapter turns that idea around: rather than treating semantic elements as an occasional upgrade, we treat them as the default, and examine exactly why that default matters so much.

In this chapter, we will examine:

- why semantic markup matters, beyond visual appearance;
- the key layout-level semantic elements HTML provides;
- ARIA roles and attributes, for the cases semantic HTML alone cannot cover;
- how semantic structure affects a page's visibility in search results.

What "Semantic" Means in HTML

A semantic element is one whose name describes the meaning of its content, not just its appearance. `<p>` means "paragraph"; `<nav>` means "navigation"; `<table>` means tabular data. A `<div>` styled to look exactly like a navigation menu, by contrast, communicates nothing about its purpose to anything reading the underlying markup rather than its rendered appearance.

```
1 <!-- Non-semantic: looks like a header, but only to a human -->
2 <div class="header">
3   <div class="site-title">UTCN</div>
4 </div>
5 <!-- Semantic: is a header, to a human and to software -->
6 <header>
7   <h1>UTCN</h1>
8 </header>
```

Who Reads Semantic Markup

- Screen readers and other assistive technology, which use semantic elements to build a navigable outline of the page;
- Search engines, which use them to understand what a page is about, discussed further in Section 9.4;
- Browsers themselves, which apply sensible default behavior (such as making `<button>` focusable and clickable without extra code);
- Other developers, including your future self, reading the markup months or years later.

A page built entirely from `<div>` elements can look identical to a semantic one in a browser, while being substantially harder to use with a screen reader and less clear to everyone who maintains it afterward.

Semantic HTML Is Not an Accessibility Feature Alone

It is tempting to think of semantic markup purely as something that benefits users of assistive technology. In practice, its benefits are broader: consistent semantic structure makes a codebase easier for a team to maintain, makes automated testing more reliable (tests can target `<nav>` or `<main>` rather than a fragile class name), and, as covered later in this chapter, has a direct effect on search visibility.

Common Beginner Mistakes

- Treating semantic elements as purely cosmetic, interchangeable with a styled `<div>`.
- Assuming that if a page looks correct, it is necessarily structured correctly underneath.
- Reaching for ARIA attributes to compensate for non-semantic markup, instead of using the semantic element that already provides the same meaning natively.

Best Practices

- Default to a semantic element whenever one accurately describes the content.
- Reserve `<div>` and `<span>` for cases with no semantic equivalent, as established in Section 4.4.

- Treat semantic structure as a first-class design decision, not an afterthought applied at the end of a project.

Conclusions

Semantic markup is not a stylistic preference; it is what allows a web page to be understood by more than just a pair of human eyes looking at a rendered layout. The rest of this chapter builds on that idea, starting with the specific elements HTML provides for describing the major regions of a page.

In the next section, we will look closely at the semantic elements used to describe page-level structure: `<header>`, `<nav>`, `<main>`, `<article>`, `<section>`, `<aside>`, and `<footer>`.

9.2. Key Semantic Elements (`<header>`, `<nav>`, `<article>`, `<footer>`, and More)

HTML5 introduced a set of elements specifically for describing the major regions of a page. Several, such as `<nav>`, have already appeared earlier in this book; this section brings the full set together and shows how they combine into a complete page.

`<header>`

The `<header>` element represents introductory content for its nearest sectioning ancestor, typically containing a heading, a logo, or introductory text. A page can have more than one `<header>`: one for the page as a whole, and others introducing individual `<article>` or `<section>` elements.

```
1 <header>
2   <h1>Web Technologies</h1>
3   <p>A course from the Applied Electronics Department</p>
4 </header>
```

<nav>

As introduced in Section 5.4, <nav> identifies a section dedicated to navigation. Not every group of links needs <nav>, only major navigation blocks, such as a primary menu or a table of contents.

<main>

The <main> element wraps the primary content of a page, excluding repeated elements such as headers, footers, and navigation menus. A document should contain at most one visible <main> element.

```
1 <main>
2   <h2>Course Overview</h2>
3   <p>This course introduces the fundamentals of HTML...</p>
4 </main>
```

Many screen readers offer a "skip to main content" shortcut that relies entirely on the presence of a <main> element.

<article>

An <article> represents a self-contained piece of content that would make sense on its own, independent of the rest of the page, such as a blog post, a news story, or a single product listing.

```
1 <article>
2   <h2>New Lab Equipment Arrives</h2>
3   <p>The Applied Electronics Department has received...</p>
4 </article>
```

<section>

A <section> represents a thematic grouping of content, generally introduced by its own heading. Unlike <article>, a section is not necessarily meant to stand alone outside the context of the document that contains it.

```
1 <section>
2   <h2>Admission Requirements</h2>
3   <p>Applicants must submit the following documents...</p>
4 </section>
```

A useful rule of thumb: use `<article>` for content that could be syndicated or read in isolation, and `<section>` for a thematic chunk of a larger page. If neither description fits and the grouping exists purely for styling, `<div>` remains the correct choice, as discussed in Section 4.4.

`<aside>`

The `<aside>` element represents content that is tangentially related to the surrounding content, such as a sidebar, a pull quote, or a set of related links.

```
1 <aside>
2   <h2>Related Courses</h2>
3   <ul>
4     <li>Computer Networks</li>
5     <li>Database Systems</li>
6   </ul>
7 </aside>
```

`<footer>`

The `<footer>` element represents closing content for its nearest sectioning ancestor, commonly containing copyright information, contact details, or links to related documents. Like `<header>`, it can appear both at the page level and within an `<article>` or `<section>`.

```
1 <footer>
2   <p>&copy; 2026 Technical University of Cluj-Napoca</p>
3 </footer>
```

Putting the Page Together

These elements are designed to be combined into a full page skeleton:

```
1 <body>
2   <header>
3     <h1>Web Technologies</h1>
4     <nav>...</nav>
5   </header>
6
7   <main>
8     <article>
9       <h2>Chapter 9: Semantic HTML and Accessibility</h2>
10      <p>...</p>
11    </article>
12
13    <aside>
14      <h2>Further Reading</h2>
15      <p>...</p>
16    </aside>
17  </main>
18
19  <footer>
20    <p>&copy; 2026 Technical University of Cluj-Napoca</p>
21  </footer>
22</body>
```

Notice that none of these elements imply any particular visual styling on their own; their default appearance is close to that of a `<div>`. Their value lies entirely in the meaning they convey.

Common Beginner Mistakes

- Using more than one `<main>` element visible on the same page.
- Confusing `<article>` and `<section>`, or using either where a plain `<div>` would actually be more appropriate.

- Wrapping the entire page in `<section>`, treating it as a generic container rather than a thematic grouping.
- Nesting `<header>` or `<footer>` inside another `<header>` or `<footer>`, which is not permitted.

Best Practices

- Build the overall page skeleton from `<header>`, `<nav>`, `<main>`, and `<footer>`.
- Use `<article>` for standalone, syndicable content and `<section>` for thematic groupings within a larger page.
- Introduce each `<section>` and `<article>` with its own heading.
- Reserve `<div>` for grouping that carries no semantic meaning.

Conclusions

Together, `<header>`, `<nav>`, `<main>`, `<article>`, `<section>`, `<aside>`, and `<footer>` let a page's structure be read directly from its markup, without relying on class names or visual layout to convey meaning.

In the next section, we turn to ARIA, and the situations where semantic HTML alone is not enough to fully describe a complex user interface.

9.3. Enhancing Accessibility with ARIA Roles

Semantic HTML covers a great deal of ground, but some interface patterns, such as a tab panel, a modal dialog, or a live status message, have no dedicated HTML element. ARIA (Accessible Rich Internet Applications) is a set of attributes that fills this gap, describing roles, states, and properties that assistive technology can use.

The First Rule of ARIA

If a native HTML element or attribute already provides the semantics and behavior you need, use it, rather than reimplementing it with a `<div>` and ARIA attributes.

```

1 <!-- Avoid -->
2 <div role="button" tabindex="0" onclick="submitForm()">
3   Submit
4 </div>
5 <!-- Prefer -->
6 <button onclick="submitForm()">Submit</button>
7

```

The native `<button>` is already focusable, already responds to both mouse clicks and the keyboard, and is already announced correctly by every screen reader, none of which the `<div>` version provides without additional work.

The role Attribute

The role attribute tells assistive technology what kind of component an element represents. It is most useful when no native element matches, or when reinforcing the meaning of landmark regions.

```

1 <div role="alert">
2   Your session will expire in 2 minutes.
3 </div>

```

Landmark roles largely mirror the semantic elements from the previous section (banner, navigation, main, complementary, contentinfo); using `<header>`, `<nav>`, `<main>`, `<aside>`, and `<footer>` directly already conveys these roles automatically, without needing role at all.

Labeling with `aria-label` and `aria-labelledby`

`aria-label` provides an accessible name directly as text; `aria-labelledby` points to the id of another element whose text should serve as the label. These are especially useful when a visual label would be redundant or absent.

```

1 <button aria-label="Close dialog">&times;</button>
2
3 <h2 id="billing-heading">Billing Address</h2>
4 <section aria-labelledby="billing-heading">
5   ...
6 </section>

```

Describing State with aria-* Attributes

- `aria-expanded` — indicates whether a collapsible section is currently open;
- `aria-hidden` — hides decorative or redundant content from assistive technology, while leaving it visible on screen;
- `aria-current` — indicates the current item within a set, as seen in Section 5.4's navigation example;
- `aria-live` — marks a region whose content updates dynamically, so that changes are announced automatically.

```

1 <button aria-expanded="false" aria-controls="faq-1">
2   What are the admission requirements?
3 </button>
4 <div id="faq-1" hidden>
5   <p>Applicants must submit...</p>
6 </div>

```

aria-hidden and Decorative Content

Icons and purely decorative graphics are often marked `aria-hidden="true"` so that assistive technology skips them, complementing the empty `alt=""` pattern introduced in Section 6.2 for decorative images.

```
1 <button>
2   <svg aria-hidden="true">...</svg>
3   Save changes
4 </button>
```

Common Beginner Mistakes

- Reaching for ARIA before checking whether a native HTML element already solves the problem.
- Adding a role that contradicts an element's native semantics, such as `role="button"` on an `<a>` without also handling keyboard interaction.
- Overusing `aria-label` on elements that already have clear visible text, which can result in the visible text and the accessible name disagreeing.
- Assuming that adding ARIA attributes alone makes a custom component accessible, without also implementing correct keyboard behavior.

Best Practices

- Apply the first rule of ARIA: prefer native elements over role-based substitutes.
- Use ARIA to fill genuine gaps, such as live regions and complex widgets with no native equivalent.
- Keep `aria-expanded`, `aria-hidden`, and similar state attributes synchronized with the actual state of the interface.
- Test ARIA-enhanced components with an actual screen reader, not just by reading the markup.

Conclusions

ARIA extends what semantic HTML can express, but it works best as a supplement to native elements, not a replacement for them. Used carefully, it closes the gap between complex interactive interfaces and the assistive technology that needs to describe them to users.

In the final section of this chapter, we will look at how the same semantic structure that benefits accessibility also benefits a page's visibility in search engines.

9.4. SEO Benefits of Semantic HTML

Search Engine Optimization (SEO) is the practice of making a page easier for search engines to find, understand, and rank appropriately. A significant portion of on-page SEO overlaps directly with the semantic and accessible practices covered throughout this book.

Why Structure Matters to Search Engines

Search engines do not see a rendered page the way a human does; they parse the underlying HTML. The same semantic elements that help a screen reader build an outline of the page, headings, landmarks, and lists, help a search engine determine what a page is about and how its content is organized.

Heading Hierarchy

As established in Section 4.1, a single `<h1>` followed by a logical progression of `<h2>`, `<h3>`, and further subheadings helps search engines identify a page's main topic and its supporting subtopics, in much the same way it helps a human skim the page.

The `<title>` and Meta Description

As introduced in Section 3.2, the `<title>` element is typically displayed as the clickable headline in search results, and the meta description, while not a ranking factor itself, frequently appears as the summary text beneath it.

```
1 <head>
2   <title>Web Technologies Course | UTCN</title>
3   <meta name="description"
4         content="An introductory course on HTML, CSS, and
5                 JavaScript at the Technical University of
6                 Cluj-Napoca.">
7 </head>
```

Meaningful Link Text

As covered in Section 5.1, descriptive link text ("Read the admissions guide" rather than "click here") gives search engines additional context about the page being linked to, in addition to benefiting users who scan a page for its links.

Image Alt Text

Search engines cannot interpret the pixels of an image directly. The alt text covered in Section 6.2 is one of the primary signals used for image search, and its absence is a missed opportunity for a page's images to be found at all.

Semantic Landmarks and Content Structure

Using <article>, <section>, <nav>, and <main> correctly, as covered earlier in this chapter, helps a search engine distinguish primary content from navigation, advertisements, or boilerplate, which can affect how accurately a page's main topic is identified.

Structured Data

Beyond semantic HTML elements, many sites add structured data, most commonly using JSON-LD, a small script block describing the page's content in a standardized vocabulary (schema.org). Search engines use this information to build rich results, such as star ratings or event dates, displayed directly in search listings.

```
1 <script type="application/ld+json">
2 {
3   "@context": "https://schema.org",
4   "@type": "Course",
5   "name": "Web Technologies",
6   "provider": {
7     "@type": "CollegeOrUniversity",
8     "name": "Technical University of Cluj-Napoca"
9   }
10 }
11 </script>
```

Structured data is optional and does not replace semantic HTML; it supplements it, giving search engines an explicit, machine-readable summary in addition to what they infer from the page's markup.

Accessibility and SEO Overlap, But Are Not Identical

A well-structured, accessible page tends to perform better in search results, but accessibility and SEO are not the same goal, and optimizing purely for search rankings should never come at the expense of genuine usability.

Common Beginner Mistakes

- Skipping heading levels or using multiple `<h1>` elements in a way that obscures the page's main topic.
- Writing alt text or link text stuffed with keywords rather than genuinely descriptive language.
- Adding structured data that does not accurately describe the page's actual content.
- Treating SEO and accessibility as separate concerns to be addressed independently, rather than largely the same underlying practice.

Best Practices

- Maintain a single, clear `<h1>` and a logical heading hierarchy beneath it.
- Write a concise, accurate `<title>` and meta description for every page.
- Use descriptive link text and meaningful alt text throughout.
- Structure content with semantic landmarks, and add structured data where it genuinely applies.

Conclusions

Much of what makes a page accessible, clear heading structure, descriptive text, and meaningful semantic elements, also makes it easier for search engines to understand and rank. This concludes our chapter on semantic HTML and accessibility. You now have the vocabulary and the practices needed to structure a page so that it can be understood by browsers, assistive technology, search engines, and other developers alike.

In the next chapter, we take a step back and look at HTML5 as a whole, tracing the innovations that have appeared throughout this book back to a single, coherent set of goals.

Chapter 10. HTML5 and Its New Features

10.1. Overview of HTML5 Innovations

Every element and attribute used so far in this book, from `<video>` to `<nav>` to the email input type, is part of HTML5. Rather than introducing an entirely new topic, this chapter looks back across the book and organizes what we have already learned around the goals that shaped it, and fills in the handful of HTML5 features not yet covered.

Before HTML5

HTML4, standardized in 1997, and the stricter XHTML that followed it, described a web built mainly from text, images, and links. Anything beyond that, video, audio, interactive graphics, offline functionality, required third-party plugins such as Flash or Silverlight, each with its own installation, security concerns, and lack of support on mobile devices.

Why HTML5 Was Created

Work on what became HTML5 began in 2004 under the WHATWG (Web Hypertext Application Technology Working Group), formed in part because the official standards process at the time was moving toward strict XML-based markup, disconnected from how browsers and real websites actually behaved. HTML5 took a different approach, favoring practical compatibility with the existing web.

Guiding Design Principles

- Support existing content — HTML5 was designed to work with the billions of pages already on the web, not to require a clean break from them;
- Degrade gracefully — a browser encountering an unfamiliar element should still display its content sensibly;
- Prioritize users — when design decisions conflict, users come before authors, which come before implementers, which come before theoretical purity;

- Be a living standard — unlike HTML4, HTML5 is continuously updated rather than released as a fixed, versioned specification.

The Major Themes of HTML5

The features introduced by HTML5, many of which we have already used throughout this book, cluster around four broad themes:

- Semantics — elements such as `<header>`, `<nav>`, `<article>`, and `<footer>`, covered in Chapter 9, which describe structure instead of relying on generic `<div>` elements;
- Native multimedia — `<audio>` and `<video>`, covered in Chapter 6, removing the need for plugins;
- Richer forms — the input types and validation attributes covered in Chapter 8, which used to require custom JavaScript;
- New APIs — programming interfaces such as Geolocation and Web Storage, which extend what a web page can do without leaving the browser, covered later in this chapter.

A Living Standard

Unlike earlier HTML versions, HTML5 is maintained as a living standard, meaning it is continuously revised rather than published as a series of discrete, numbered releases. In practice, this means new elements and APIs are added incrementally over time, and "HTML5" today refers to the current state of HTML itself, rather than a version that was finalized once and left unchanged.

Conclusions

HTML5 is less a single new technology than a set of goals, practical compatibility, native semantics, native multimedia, richer forms, and new APIs, that have shaped essentially every element covered in this book so far.

In the next section, we will look at the handful of structural elements HTML5 introduced that have not yet been covered in detail: `<main>` and `<details>/<summary>`, alongside a complete recap of the semantic elements from Chapter 9.

10.2. New Structural Elements

Chapter 9 already introduced the major layout-level semantic elements HTML5 added. This section recaps them briefly and completes the picture with two elements not yet covered: `<details>/<summary>`, a disclosure widget built entirely from HTML.

Recap: Layout Semantics

- `<header>`, `<nav>`, `<main>`, `<footer>` — page-level structure (Section 9.2);
- `<article>`, `<section>`, `<aside>` — content-level grouping (Section 9.2);
- `<figure>`, `<figcaption>` — captioned content (Section 6.1);
- `<mark>`, `<time>`, `<abbr>`, `<cite>` — inline text semantics (Section 4.2).

`<details>` and `<summary>`

The `<details>` element creates a disclosure widget, a section of content that can be expanded or collapsed by the user, entirely without JavaScript. Its first child, `<summary>`, provides the visible, always-present label.

```
1 <details>
2   <summary>What are the admission requirements?</summary>
3   <p>Applicants must submit a completed application form,
4     academic transcripts, and a letter of intent.</p>
5 </details>
```

By default, `<details>` is collapsed; adding the `open` attribute displays it expanded from the start.

```
1 <details open>
2   <summary>Contact Information</summary>
3   <p>Str. Memorandumului nr. 28, Cluj-Napoca</p>
4 </details>
```

<details> is fully keyboard-accessible and correctly announced by screen readers without any additional ARIA, a good example of a native element replacing what once required custom JavaScript and ARIA attributes.

A Common Use: FAQ Sections

A frequently asked questions section is a natural fit for <details>, with one widget per question.

```
1 <h2>Frequently Asked Questions</h2>
2
3 <details>
4   <summary>Is prior programming experience required?</summary>
5   <p>No prior experience is required for this course.</p>
6 </details>
7
8 <details>
9   <summary>What software do I need?</summary>
10  <p>A text editor and a modern web browser are sufficient.</p>
11 </details>
```

Common Beginner Mistakes

- Recreating a collapsible section with JavaScript and a <div>, when <details> already provides the same behavior natively.
- Placing <summary> anywhere other than as the first child of <details>.
- Overusing <details> for content that should simply always be visible.

Best Practices

- Use <details> and <summary> for genuinely optional or secondary content, such as FAQs or supplementary notes.
- Reach for native disclosure behavior before reimplementing it with JavaScript.
- Keep the <summary> text descriptive, since it is the only part visible when the widget is collapsed.

Conclusions

Between the semantic elements recapped from Chapter 9 and the disclosure behavior of `<details>` and `<summary>`, HTML5 covers a surprising range of structural and interactive patterns without requiring any scripting at all.

In the next section, we return to multimedia, extending Chapter 6 with HTML5's graphics capabilities: the `<canvas>` and `<svg>` elements.

10.3. Multimedia and Graphics with HTML5

Chapter 6 covered `<audio>` and `<video>` in depth. This section completes the multimedia picture with HTML5's two graphics technologies: `<canvas>`, for drawing bitmap graphics with JavaScript, and inline `<svg>`, for resolution-independent vector graphics.

The `<canvas>` Element

The `<canvas>` element creates a blank drawing surface of a given pixel size. On its own, it renders nothing; content is drawn onto it using JavaScript, most commonly through the Canvas 2D API.

```
1 <canvas id="chart" width="400" height="200">
2   Your browser does not support the canvas element.
3 </canvas>
```

```
1 const ctx = document.getElementById("chart")
2   .getContext("2d");
3 ctx.fillStyle = "#4c6cc3";
4 ctx.fillRect(10, 10, 150, 80);
```

Canvas is well suited to content generated or redrawn frequently, such as charts, games, and image manipulation, since it works with raw pixels rather than individual, addressable shapes.

Inline SVG

As mentioned in Section 6.1, SVG (Scalable Vector Graphics) is an XML-based vector format. Unlike an image referenced through `<img>`, SVG markup can be written directly inline in an HTML document, making its individual shapes addressable by CSS and JavaScript.

```
1 <svg width="120" height="120" viewBox="0 0 120 120">
2   <circle cx="60" cy="60" r="50"
3       fill="#4c6cc3" stroke="#333" stroke-width="2">
4   </circle>
5 </svg>
```

Because SVG is vector-based, this circle remains crisp at any display size or zoom level, unlike a bitmap image that pixelates when scaled up.

Choosing Between `<canvas>`, SVG, and `<img>`

- `<img>` — for photographs and static graphics that do not need to be manipulated by CSS or JavaScript;
- Inline SVG — for icons, logos, and illustrations that benefit from scaling cleanly or being styled and animated individually;
- `<canvas>` — for dynamically generated or frequently redrawn graphics, such as charts, games, and pixel-level image processing.

A useful distinction: SVG describes a scene as a set of individual, addressable shapes, while canvas produces a flat grid of pixels with no memory of how they were drawn.

Accessibility Considerations

Content drawn on `<canvas>` is invisible to screen readers unless equivalent information is also provided as fallback content or through ARIA; inline SVG, by contrast, can include a `<title>` element and ARIA attributes describing its content directly.

```
1 <svg role="img" aria-labelledby="logo-title" width="60" height="60">
2   <title id="logo-title">UTCN Logo</title>
3   <circle cx="30" cy="30" r="28" fill="#4c6cc3"></circle>
4 </svg>
```

Common Beginner Mistakes

- Using `<canvas>` for static graphics that would be simpler, more accessible, and more maintainable as SVG or a plain image.
- Publishing an SVG or canvas graphic with no text alternative at all.
- Referencing an external `.svg` file with `<img>` when the interactivity or styling benefits of inline SVG are actually needed.

Best Practices

- Default to `<img>` or CSS for static visuals.
- Use inline SVG when individual shapes need to be styled, animated, or scripted.
- Reserve `<canvas>` for graphics that are genuinely generated or redrawn programmatically.
- Always provide a text alternative for graphical content, regardless of which technology renders it.

Conclusions

Between native audio and video, covered in Chapter 6, and canvas and SVG graphics, covered here, HTML5 lets a page handle rich visual and multimedia content without relying on any third-party plugin.

In the final section of this chapter, we will look at the broader family of HTML5 APIs that extend what a page can do, from remembering data between visits to knowing the user's location.

10.4. APIs and Integration with Other Web Technologies

Alongside new elements, HTML5 introduced a family of JavaScript APIs that give web pages capabilities previously reserved for native applications. A full treatment of these APIs belongs to a book on JavaScript; this closing section gives a conceptual overview of the ones most relevant to HTML, and how they fit into the page.

Every API in this section is used through JavaScript, not through HTML markup directly; the goal here is to know what each one is for, not to master its full syntax.

Web Storage: `localStorage` and `sessionStorage`

Web Storage lets a page store key-value data directly in the browser. `localStorage` persists until explicitly cleared; `sessionStorage` is cleared when the browser tab closes.

```
1 localStorage.setItem("theme", "dark");  
2 const theme = localStorage.getItem("theme");
```

This replaced the older, more limited practice of storing small amounts of data in cookies for purposes that had nothing to do with communicating with a server.

The Geolocation API

The Geolocation API lets a page request the user's physical location, with explicit permission requested through the browser each time.

```
1 navigator.geolocation.getCurrentPosition((position) => {  
2   console.log(position.coords.latitude,  
3               position.coords.longitude);  
4 });
```

The Drag and Drop API

HTML5 standardized native drag-and-drop interaction, using the `draggable` attribute together with a set of JavaScript drag events, allowing elements to be picked up and dropped without any external library.

```
1 <div draggable="true" id="task-1">Prepare lecture slides</div>
```

Offline Support: Service Workers and the Cache API

Service workers are a background script that a page can register, capable of intercepting network requests and serving cached responses. Together with the Cache API, they make it possible for a web application to continue functioning, at least partially, without a network connection.

The History API

The History API allows JavaScript to change the URL displayed in the address bar and add entries to the browser's history, without triggering a full page reload. This is the mechanism behind most modern single-page applications, which update the visible content and URL together as the user navigates.

Web Workers

Web Workers run JavaScript on a separate background thread, allowing computationally expensive work to proceed without freezing the page's user interface.

How These APIs Relate to the Rest of This Book

None of these APIs change the HTML markup covered throughout this book; they operate on top of it, reading and modifying the same document structure through the DOM introduced back in Section 2.1. A solid grasp of HTML's structure and semantics, the subject of this entire volume, is what makes working with any of these APIs coherent rather than arbitrary.

Common Beginner Mistakes

- Storing sensitive data in `localStorage`, which is readily accessible to any script running on the page.

- Requesting geolocation or other permissions without a clear, user-facing reason, which tends to result in the permission being denied.
- Reaching for a complex API, such as Web Workers, before confirming that simpler HTML and CSS solutions are insufficient.

Best Practices

- Use each API for the problem it was designed to solve, rather than as a default habit.
- Request sensitive permissions, such as geolocation, only at the point where the user understands why they are needed.
- Treat these APIs as an extension of well-structured HTML, not a replacement for it.

Conclusions

HTML5's APIs, storage, geolocation, drag and drop, offline support, background threads, and more, extend what a web page can do well beyond displaying static content, while still depending entirely on the well-structured HTML document underneath them.

This concludes our chapter on HTML5. Having traced its semantics, multimedia, forms, and APIs back to a common set of goals, you are now equipped to recognize why HTML5 looks the way it does, not just how to use it.

In the next chapter, we turn to best practices and common pitfalls, consolidating the guidance scattered throughout this book into a single, focused discussion.

Chapter 11. Best Practices and Common Pitfalls

11.1. Writing Clean and Maintainable HTML Code

Every technique in this book so far has focused on what HTML can do. This chapter shifts focus to how it is written, covering the habits that keep a codebase readable as it grows, and the pitfalls that most often trip up developers moving from small examples to real projects.

In this section, we will examine:

- consistent formatting and indentation;
- naming conventions for class and id attributes;
- keeping structure, presentation, and behavior separate;
- commenting HTML effectively, without overdoing it.

Consistent Indentation and Formatting

Nested elements should be indented consistently, using either spaces or tabs, but not a mixture of the two within the same project. Most editors, including the one recommended in Section 2.3, can be configured to enforce this automatically.

```
1 <!-- Hard to follow -->
2 <section><h2>Courses</h2>
3 <ul><li>Web Technologies</li>
4 <li>Databases</li></ul></section>
5
6 <!-- Consistent and readable -->
7 <section>
8   <h2>Courses</h2>
9   <ul>
10     <li>Web Technologies</li>
11     <li>Databases</li>
12   </ul>
13 </section>
```

Lowercase Tags and Attributes

While HTML is not case-sensitive, the long-standing convention, inherited from XHTML and followed by every major style guide, is to write tag names and attributes in lowercase.

```
1 <!-- Avoid -->
2 <DIV CLASS="Card">
3
4 <!-- Prefer -->
5 <div class="card">
```

Quoting Attribute Values

Attribute values should always be quoted, even though HTML permits omitting quotes for values with no spaces. Consistent quoting avoids subtle bugs and reads more predictably.

```
1 <!-- Technically valid, but avoid -->
2 <input type=text name=username>
3
4 <!-- Prefer -->
5 <input type="text" name="username">
```

Meaningful class and id Names

As noted in Section 4.4, class and id values should describe purpose, not appearance or position.

```
1 <!-- Avoid: describes appearance, and only for now -->
2 <div class="blue-box-left">
3
4 <!-- Prefer: describes purpose -->
5 <div class="course-card">
```

Separating Structure, Presentation, and Behavior

HTML describes structure, CSS describes presentation, and JavaScript describes behavior; mixing the three together in one place makes a page harder to maintain.

```
1 <!-- Avoid: inline style and inline event handler -->
2 <button style="background: red;" onclick="alert('Hi')">
3   Click me
4 </button>
5
6 <!-- Prefer: styling and behavior kept in their own files -->
7 <button class="alert-button" id="greet-btn">
8   Click me
9 </button>
```

Commenting HTML

Comments, introduced in Section 3.4, are most useful for marking the boundaries of major sections in a long document, or explaining a non-obvious decision. Comments that merely restate what the markup already says add noise rather than clarity.

```
1 <!-- Useful: explains a non-obvious choice -->
2 <!-- Hidden on mobile; duplicated in the mobile nav below -->
3 <nav class="desktop-nav">...</nav>
4 <!-- Not useful: restates the obvious -->
5 <!-- This is a paragraph -->
6 <p>Welcome to our site.</p>
7
```

Common Beginner Mistakes

- Mixing indentation styles within the same file or project.
- Leaving inline style and onclick-style attributes scattered throughout the markup.
- Writing comments that describe what is already obvious from the code itself.
- Letting inconsistent formatting accumulate instead of running an auto-formatter regularly.

Best Practices

- Adopt a consistent indentation style and enforce it with editor settings or an auto-formatter.
- Use lowercase tags and attributes, and always quote attribute values.
- Name class and id values for what they represent, not how they currently look.
- Keep structure, presentation, and behavior in HTML, CSS, and JavaScript respectively.

Conclusions

None of these habits change what a browser renders; they change how quickly the next person, quite possibly you, months later, can understand and safely modify the page. Clean HTML is a courtesy to every future reader of the code.

In the next section, we turn to a different kind of consistency: making sure a page behaves the same way across different browsers and devices.

11.2. Cross-Browser Compatibility Issues

A page is rarely viewed in only one browser. Chrome, Safari, Firefox, and Edge each use a different rendering engine, and mobile browsers introduce further variation. This section looks at why differences arise and how to manage them.

Why Browsers Behave Differently

Each major browser is built on one of a small number of rendering engines: Blink (Chrome, Edge, and most Chromium-based browsers), WebKit (Safari), and Gecko (Firefox). While all three implement the same HTML specification, differences remain in the timing of new feature support, default styling, and edge-case behavior.

Progressive Enhancement and Graceful Degradation

As mentioned in Section 10.1, one of HTML5's guiding principles is that content should degrade gracefully in browsers that do not support a given feature. Two related strategies put this into practice:

- Progressive enhancement — build a functional baseline experience first, then layer on enhancements for browsers that support them;
- Graceful degradation — build the full experience first, and ensure it fails safely, rather than catastrophically, where support is missing.

```
1 <video controls width="640" height="360">
2   <source src="intro.webm" type="video/webm">
3   <source src="intro.mp4" type="video/mp4">
4   <a href="intro.mp4">Download the video</a>
5 </video>
```

Here, a browser with no video support at all still presents a working download link, rather than a blank gap in the page.

Checking Feature Support

Before relying on a newer HTML feature, it is worth checking how broadly it is supported. Sites such as caniuse.com maintain up-to-date compatibility tables across browsers and versions, and the Mozilla Developer Network (MDN) documents browser compatibility directly alongside each feature's reference page.

Testing Across Browsers and Devices

- Test in more than one browser during development, not only at the end of a project;
- Use a browser's built-in device emulation (available in most developer tools) to preview mobile layouts without a physical device;
- Where possible, test on at least one real mobile device, since emulation does not perfectly reproduce touch interaction or on-device performance.

The Viewport Meta Tag and Mobile Rendering

As introduced in Section 3.2, the viewport meta tag instructs mobile browsers on how to scale a page. Omitting it is one of the most common causes of a page appearing zoomed out and illegible on a phone, even when the underlying HTML and CSS are otherwise correct.

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

Common Beginner Mistakes

- Developing and testing in only one browser throughout a project.
- Using a very new HTML feature without any fallback for browsers that do not yet support it.
- Forgetting the viewport meta tag and being surprised by how a page looks on a phone.
- Assuming that if a page looks correct on a laptop, it will necessarily work correctly for every visitor.

Best Practices

- Test across multiple browsers and at least one real mobile device throughout development, not only at the end.
- Check feature support before depending on something recently introduced.
- Design fallback content for features that may not be universally supported.
- Always include a correctly configured viewport meta tag.

Conclusions

Cross-browser differences are a permanent feature of the web, not a problem to be solved once and forgotten. Designing for graceful degradation, and testing across real browsers and devices, keeps a page usable for everyone, not just whoever happens to be using the same browser as its author.

In the next section, we turn to the tools used to catch these kinds of problems before they reach real users: debugging and validating HTML.

11.3. Debugging and Validating HTML

HTML is forgiving: browsers will attempt to render even badly malformed markup, silently correcting many mistakes. This tolerance is convenient, but it also means errors can go unnoticed for a long time unless they are actively checked for.

Using Browser Developer Tools

As introduced in Section 2.3, every modern browser includes built-in developer tools, opened with F12 or a right-click "Inspect" option. Two panels are especially useful for HTML work:

- The Elements (or Inspector) panel shows the live DOM, letting you see exactly how the browser interpreted the page's markup, including any tags it silently corrected;
- The Console panel surfaces JavaScript errors and warnings, some of which stem from malformed or unexpected HTML structure.

The Elements panel shows the DOM as the browser currently understands it, which is not always identical to the original HTML source, since the browser silently repairs certain mistakes as it parses.

Validating with the W3C Markup Validator

As mentioned in Section 2.2, the W3C Markup Validator (validator.w3.org) checks a document against the official HTML specification and reports errors and warnings, such as unclosed tags, invalid nesting, duplicate id values, or missing required attributes.

```
1 <!-- Errors a validator would catch -->
2  <!-- missing alt -->
3 <p>Welcome
  <div id="box">A</div>
4 <div id="box">B</div> <!-- duplicate id -->
5
```

A validator can check a live URL, an uploaded file, or pasted markup directly, making it easy to check a page before it is even published.

Common Validation Errors

- Unclosed or mismatched tags;
- Missing required attributes, such as alt on `<img>`;
- Duplicate id values within the same page;
- Invalid nesting, such as a `<div>` inside a `<p>`, or a block-level element inside an inline one;

- Using an attribute that does not apply to a given element.

HTML Linters and Editor Integration

Beyond one-off validation, many code editors support linting HTML directly as it is typed, underlining problems immediately rather than waiting for a separate validation pass. This is typically configured once per project and then runs automatically in the background.

Debugging Layout Problems

Many apparent "HTML problems" are actually caused by CSS, but the Elements panel remains the starting point for diagnosing them: selecting an element reveals its box model, applied styles, and computed dimensions, making it possible to trace exactly why an element is positioned or sized the way it is.

Common Beginner Mistakes

- Assuming that because a page looks correct, its markup must also be valid.
- Ignoring validator warnings as unimportant, when they often point to real accessibility or compatibility problems.
- Debugging by editing the live DOM in developer tools without also updating the actual HTML source file.
- Never running a validator at all until a problem is already visible to users.

Best Practices

- Run a validator periodically during development, not only when something appears broken.
- Use the Elements panel to confirm how the browser actually parsed your markup.
- Treat validator warnings as worth investigating, even when they do not visibly break the page.
- Fix errors in the source file, not just in the browser's live copy of the DOM.

Conclusions

Because browsers tolerate invalid HTML, validation and debugging tools are what catch the mistakes a browser quietly papers over. Making them a routine part of development, rather than a last resort, keeps small errors from compounding into larger problems.

In the final section of this chapter, we will look ahead at where HTML is heading, and the trends shaping how it continues to evolve.

11.4. Future Trends in HTML Development

As established in Section 10.1, HTML is maintained as a living standard rather than a series of fixed versions. This closing section looks at some of the directions in which it continues to evolve.

Web Components

Web Components are a set of native browser technologies, custom elements, the Shadow DOM, and HTML templates, that let developers define their own reusable, encapsulated HTML elements, complete with their own internal structure and styling.

```
1 <!-- A hypothetical custom element -->
2 <course-card course-id="web-tech-101"></course-card>
```

Once registered with JavaScript, a custom element like this behaves like any built-in HTML element, and can be used, and reused, the same way throughout a project.

Continued Growth of Native Form Controls

HTML has steadily reduced the amount of custom JavaScript needed for common interface patterns. `<dialog>`, for instance, provides a native modal dialog with built-in focus handling, something that once required a substantial JavaScript library.

```
1 <dialog id="confirm-dialog">
2   <p>Are you sure you want to withdraw from this course?</p>
3   <button value="cancel">Cancel</button>
4   <button value="confirm">Confirm</button>
5 </dialog>
```

This mirrors the pattern seen with `<details>` in Section 10.2: functionality that once required JavaScript and custom ARIA handling becoming available natively, with accessibility handled correctly by default.

Performance-Oriented Attributes

HTML has also grown simple attributes aimed directly at performance. The `loading` attribute on `<img>`, for example, lets the browser defer loading off-screen images until the user scrolls near them.

```
1 
```

Improvements like this reduce reliance on JavaScript libraries for what is fundamentally a loading strategy, consistent with HTML5's original goal of reducing dependence on external tooling for common needs.

Accessibility as a Baseline Expectation

Across the industry, accessibility has shifted from an afterthought to an expectation built into tooling, frameworks, and, as this chapter has shown, HTML itself. New elements increasingly arrive with sensible accessible behavior already built in, reducing the amount of ARIA needed to achieve the same result manually.

A Living Standard, Not a Finished One

The most important trend to internalize is the one introduced back in Chapter 10: HTML is never "finished." Staying current means periodically revisiting documentation such as MDN, rather than treating what this book covers as a permanently fixed body of knowledge.

Common Beginner Mistakes

- Assuming HTML is a static, fully settled technology once learned.
- Reaching immediately for a JavaScript library to solve a problem that a native HTML element may already address.
- Adopting a brand-new feature project-wide before checking its current level of browser support.

Best Practices

- Periodically check whether a native HTML solution now exists for a problem previously solved with JavaScript.
- Follow release notes from browser vendors and resources such as MDN to stay current.
- Continue applying the fundamentals from this book, semantic structure, accessibility, and graceful degradation, as the foundation underneath any new feature.

Conclusions

HTML continues to absorb functionality that once required JavaScript, while steadily raising the baseline for accessibility and performance. This concludes our chapter on best practices and common pitfalls. You now have a working checklist for writing clean, cross-browser compatible, validated HTML, along with a sense of where the language is headed next.

In the next, and final, chapter of this book, we bring everything together in a hands-on project: planning, building, and testing a complete web page from scratch.

Chapter 12. Hands-On Project: Building a Simple Web Page

12.1. Planning Your Web Page Layout

This final chapter departs from the section-by-section format of the rest of the book and instead builds a single, complete project from start to finish: a personal academic portfolio page. Every technique used in this chapter has already been introduced somewhere earlier in the book; the goal here is to see them work together.

In this section, we will examine:

- taking inventory of the content the page needs to present;
- sketching a layout before writing any markup;
- planning semantic structure alongside visual layout;
- deciding what belongs on this page, and what does not.

The Project: A Personal Academic Portfolio Page

Our example page belongs to a fictional student, Ana Popescu, and needs to present who she is, what she has worked on, and how to get in touch with her, on a single page.

Content Inventory

Before writing any HTML, it helps to list, in plain language, every piece of content the page needs:

- A site title and a short navigation menu;
- An introductory section: name, a short bio, and a profile photo;
- A table summarizing coursework and grades;
- A short embedded video, introducing a class project;
- A contact form for visitors to send a message;

- A footer with copyright and social links.

Sketching the Layout

A rough sketch, even a simple text outline, clarifies the page's structure before any code is written:

- Header — site title, navigation;
- Main
- About section — photo, bio;
- Coursework section — table;
- Projects section — video;
- Contact section — form;
- Footer — copyright, links.

This outline maps directly onto the semantic elements from Chapter 9: a `<header>`, a `<main>` containing several `<section>` elements, and a `<footer>`.

Planning Semantics Alongside Layout

Deciding which semantic element each piece of content belongs in is part of planning, not an afterthought to be fixed once the page already works visually.

For each block identified above, we can already commit to an element:

- Site title and navigation → `<header>` containing `<nav>`;
- About, Coursework, Projects, Contact → four `<section>` elements inside `<main>`, each introduced by its own heading;
- Copyright and links → `<footer>`.

Deciding What Not to Include

A focused page is easier to build well than a sprawling one. For this project, we deliberately leave out a blog, a publications archive, and a multi-page site structure, all reasonable additions, but unnecessary for demonstrating the techniques covered in this book on a single page.

Conclusions

A short content inventory and a plain-text layout sketch, done before writing any markup, make the remaining steps in this chapter largely a matter of translation: turning a plan already expressed in semantic terms into actual HTML.

In the next section, we begin implementing this plan, starting with the document's head and overall page skeleton.

12.2. Implementing the Structure in HTML

With the plan from the previous section in hand, we can now build the page's structure, starting from the document head and working outward.

The Document Head

As covered in Chapter 3, the `<head>` carries metadata rather than visible content. For this project, it includes character encoding, the viewport tag from Section 11.2, a title, a meta description, and a link to an external stylesheet, whose contents belong to the companion volume on CSS.

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="UTF-8">
5    <meta name="viewport" content="width=device-width, initial-scale=1.0">
6    <title>Ana Popescu | Student Portfolio</title>
7    <meta name="description"
8          content="Portfolio page for Ana Popescu, Electronics
9                  Engineering student at UTCN.">
10   <link rel="stylesheet" href="styles.css">
11 </head>
```

The lang attribute on `<html>`, mentioned briefly in Chapter 3, should always be set; it helps screen readers choose the correct pronunciation rules and helps browsers offer accurate translation.

The Header and Navigation

Following the plan, the page opens with a `<header>` containing a site title and a `<nav>`, as established in Sections 5.4 and 9.2.

```
1 <body>
2   <header>
3     <h1>Ana Popescu</h1>
4     <nav aria-label="Primary">
5       <a href="#about">About</a>
6       <a href="#coursework">Coursework</a>
7       <a href="#projects">Projects</a>
8       <a href="#contact">Contact</a>
9     </nav>
10  </header>
```

Each navigation link uses a fragment identifier, as covered in Section 5.3, pointing to the id of the matching section further down the page.

The Main Content Skeleton

Inside `<main>`, each section from the plan becomes a `<section>` element, introduced by its own heading and given an id matching the navigation links above.

```
1 <main>
2   <section id="about">
3     <h2>About</h2>
4     <!-- photo and bio: Section 12.3 -->
5   </section>
6   <section id="coursework">
7     <h2>Coursework</h2>
8     <!-- table: Section 12.3 -->
9   </section>
10  <section id="projects">
11    <h2>Projects</h2>
12    <!-- video: Section 12.3 -->
13  </section>
14  <section id="contact">
15    <h2>Contact</h2>
16    <!-- form: Section 12.3 -->
17  </section>
18 </main>
```

The Footer

The page closes with a `<footer>`, matching the pattern established in Section 9.2.

```
1  <footer>
2    <p>&copy; 2026 Ana Popescu</p>
3    <p>
4      <a href="https://github.com/example">GitHub</a>
5      <a href="https://linkedin.com/in/example">LinkedIn</a>
6    </p>
7  </footer>
8 </body>
9 </html>
```

Checking the Skeleton

At this stage, before adding a single image, table, or form control, the page already has a complete, navigable, semantic structure. Opening it in a browser shows a plain but entirely functional outline: a title, working navigation links, four empty sections, and a footer.

Building the skeleton first, and filling in content afterward, makes it far easier to keep the overall structure clean, rather than retrofitting semantic elements around content that has already been written as a flat pile of `<div>` elements.

Conclusions

The page now has a complete structural skeleton, built directly from the plan in Section 12.1 and the semantic elements from Chapter 9. Every remaining step is a matter of filling in each section's content.

In the next section, we fill in that content: a profile photo, a coursework table, an embedded video, and a contact form.

12.3. Incorporating Multimedia and Forms

With the skeleton from the previous section in place, we now fill each `<section>` with real content, drawing directly on Chapters 6, 7, and 8.

The About Section: Photo and Bio

The about section combines a captioned profile photo, using `<figure>` and `<figcaption>` as covered in Section 6.1, with a short biography.

```
1 <section id="about">
2   <h2>About</h2>
3   <figure>
4     
7     <figcaption>Ana Popescu, Electronics Engineering, 3rd year</figcaption>
8   </figure>
9   <p>
10    I am a third-year Electronics Engineering student at UTCN,
11    focused on embedded systems and web-connected devices.
12  </p>
13 </section>
```

Notice the width, height, and loading attributes on `<img>`, drawn directly from Sections 6.2 and 11.4, reserving the correct layout space and deferring loading until needed.

The Coursework Section: A Table

Coursework is genuinely tabular data, matching the criteria from Chapter 7, and is presented accordingly, complete with a caption and properly scoped headers.

```

1 <section id="coursework">
2   <h2>Coursework</h2>
3   <table>
4     <caption>Selected coursework, current academic year</caption>
5     <thead>
6       <tr>
7         <th scope="col">Course</th>
8         <th scope="col">Semester</th>
9         <th scope="col">Grade</th>
10      </tr>
11    </thead>
12    <tbody>
13      <tr>
14        <th scope="row">Web Technologies</th>
15        <td>5</td>
16        <td>9.5</td>
17      </tr>
18      <tr>
19        <th scope="row">Embedded Systems</th>
20        <td>5</td>
21        <td>9.0</td>
22      </tr>
23    </tbody>
24  </table>
25</section>

```

The Projects Section: An Embedded Video

The projects section embeds a short video using the native `<video>` element from Section 6.3, with multiple source formats and captions from Section 6.4.

```

1 <section id="projects">
2   <h2>Projects</h2>
3   <p>
4     A short walkthrough of my final-year project, a Wi-Fi
5     connected battery monitor:
6   </p>
7   <video controls width="480" height="270"
8     poster="battery-monitor-poster.jpg">
9     <source src="battery-monitor.webm" type="video/webm">
10    <source src="battery-monitor.mp4" type="video/mp4">
11    <track kind="captions" src="battery-monitor-en.vtt"
12      srclang="en" label="English" default>
13    Your browser does not support the video element.
14  </video>
15</section>

```

The Contact Section: A Form

Finally, the contact section uses the form elements and validation attributes from Chapter 8, grouped with `<fieldset>` and correctly labeled throughout.

```
1 <section id="contact">
2   <h2>Contact</h2>
3   <form action="/send-message" method="post">
4     <fieldset>
5       <legend>Send a message</legend>
6
7       <label for="name">Name</label>
8       <input type="text" id="name" name="name" required>
9
10      <label for="email">Email</label>
11      <input type="email" id="email" name="email" required>
12
13      <label for="message">Message</label>
14      <textarea id="message" name="message" rows="5"
15                required></textarea>
16
17      <button type="submit">Send</button>
18    </fieldset>
19  </form>
20</section>
```

Conclusions

Every section of the page is now filled with real content, and every piece of markup traces back to a technique introduced earlier in this book: captioned images, an accessible table, a captioned video, and a properly labeled, validated form.

In the final section of this book, we test the finished page and walk through the process of iterating on it based on what that testing reveals.

12.4. Testing and Iterating on Your Design

A page is not finished the moment its markup is written. This closing section applies the debugging, validation, and compatibility practices from Chapter 11 to the portfolio page built over the previous three sections.

Validating the Markup

The first step is running the complete page through the W3C Markup Validator, as covered in Section 11.3, checking for unclosed tags, missing attributes, and invalid nesting before moving on to anything else.

Testing Keyboard Navigation

With the mouse set aside, the entire page should be usable with the Tab key alone: moving through the navigation links, into the contact form's fields, and to the submit button, in a logical order that matches the visual layout.

If an interactive element cannot be reached or activated using only the keyboard, it is not accessible, regardless of how it looks.

Testing with a Screen Reader

Turning on a screen reader, built into every major desktop and mobile operating system, and navigating the page by its headings and landmarks confirms that the semantic structure from Chapter 9 actually delivers a coherent outline: About, Coursework, Projects, and Contact should each be announced clearly as the user moves through the page.

Testing Responsiveness

Using the device emulation tools from Section 11.2, the page should be checked at a range of screen widths. While the fine details of responsive layout belong to the companion CSS volume, this stage often surfaces HTML-level problems too, such as a table that badly needs the scrollable wrapper from Section 7.3 on narrow screens.

Testing Across Browsers

Finally, the page is opened in at least two different browsers, confirming that the video plays, the form behaves consistently, and nothing depends on behavior specific to a single rendering engine, following the guidance from Section 11.2.

Gathering Feedback and Iterating

Automated checks catch structural problems, but they cannot tell you whether the content itself is clear. Asking a classmate or colleague to complete a simple task, such as finding your contact form and successfully sending a message, often reveals usability issues no validator will ever flag.

- Is the purpose of the page clear within the first few seconds?
- Can a visitor find and use the contact form without hesitation?
- Does anything look broken or behave unexpectedly on their device?

Based on this kind of feedback, the page is revised, and the testing steps above are repeated. This cycle, build, test, gather feedback, revise, does not end when a page is first published; it continues for as long as the page is maintained.

Conclusions

Validating markup, testing with a keyboard and a screen reader, checking responsiveness, and testing across browsers turn a page that merely renders into one that genuinely works for the people who visit it. This concludes our hands-on project, and with it, the main body of this book.

Across twelve chapters, we have gone from a single `<!DOCTYPE html>` declaration to a complete, accessible, validated, multi-featured web page, built entirely from the elements and practices covered along the way. The Conclusion that follows reflects on this journey and points toward what comes next.

Conclusion

A Recap of Key Concepts

This book began with a single question: what is HTML, and why does it matter? Twelve chapters later, that question has a considerably richer answer than it did at the start.

We began with the fundamentals: the structure of an HTML document, the DOCTYPE declaration, and the roles of `<html>`, `<head>`, and `<body>`. From there, we built outward, chapter by chapter:

- text and structural elements, from headings and paragraphs to lists and generic containers;
- hyperlinks and navigation, the feature that turns individual pages into an interconnected web;
- images and multimedia, embedded natively and made accessible through captions, transcripts, and thoughtful use of alt text;
- tables, for presenting genuinely tabular data with proper structure;
- forms, for collecting and validating user input directly in the browser;
- semantic HTML and accessibility, and the ARIA attributes that extend them;
- HTML5 as a coherent whole, tracing its elements and APIs back to a shared set of design goals;
- best practices, cross-browser compatibility, and the tools used to debug and validate real markup;
- and finally, a complete hands-on project, bringing every one of these techniques together on a single page.

If a single idea runs through all twelve chapters, it is this: HTML is not just about making content appear on a screen, it is about describing what that content means. Every choice of element, from `<strong>` instead of `<b>`, to `<nav>` instead of `<div>`, to a properly labeled form field, is a small act of communication aimed

at browsers, assistive technology, search engines, and other developers alike, not just at the person currently looking at the screen.

Next Steps in Web Development

HTML rarely works alone. Having built a solid foundation in structure and semantics, the next steps most readers of this book will want to take are:

- CSS, for controlling the visual presentation of everything built in this book, and the natural companion volume to this one;
- JavaScript, for adding interactivity and dynamic behavior on top of a well-structured HTML foundation;
- responsive and mobile-first design, building on the viewport meta tag and testing practices introduced in Chapters 3 and 11;
- web accessibility in greater depth, extending the ARIA foundations from Chapter 9;
- a server-side language or framework, for generating HTML dynamically and handling the form submissions covered in Chapter 8.

None of these subjects require abandoning what this book has covered; each one builds directly on top of it. A page with confused or missing HTML structure remains difficult to style, script, or make accessible, no matter how advanced the tools layered on top of it.

Additional Resources

Appendix C, immediately following this conclusion, lists a curated set of official references, learning platforms, and tools for continuing to build on the material covered here. Appendix A provides a quick-reference table of the HTML elements used throughout this book, and Appendix B collects the key terms introduced along the way into a single glossary.

Thank you for working through this volume. Web technologies continue to evolve, but the fundamentals covered here, structure, semantics, and accessibility, remain the foundation that everything else on the web is built upon.

Appendices

A. HTML Tag Reference

A quick-reference summary of the HTML elements used throughout this book, organized by category and cross referenced to the section where each was introduced.

Document Structure

Element	Purpose	Section
<!DOCTYPE html>	Declares the document as HTML5	2.4
<html>	The root element of the document	3.1
<head>	Container for metadata, not displayed directly	3.2
<title>	The document's title, shown in the browser tab	3.2
<meta>	Metadata: charset, viewport, description, and more	3.2
<link>	Links an external resource, such as a stylesheet	3.2
<body>	Contains all visible page content	3.3
<base>	Sets a base URL for all relative links on the page	5.2

Text and Structural Elements

Element	Purpose	Section
<h1>–<h6>	Section headings, in order of importance	4.1
<p>	A paragraph of text	4.1
<div>	A generic block-level container	4.1
	A generic inline container	4.4
	Strong importance	4.2
	Stressed emphasis	4.2
<abbr>	An abbreviation, with its expansion in title	4.2
<mark>	Highlighted, contextually relevant text	4.2
<time>	A machine-readable date or time	4.2
 	A single line break	4.2
	An unordered (bulleted) list	4.3
	An ordered (numbered) list	4.3
	A list item	4.3
<dl>, <dt>, <dd>	A definition list: terms and descriptions	4.3

Links, Images, and Multimedia

Element	Purpose	Section
<a>	A hyperlink	5.1
	An embedded image	6.1
<figure>, <figcaption>	A captioned piece of content	6.1
<audio>	Embedded, playable audio	6.3
<video>	Embedded, playable video	6.3
<source>	An alternate media format for audio/video	6.3
<track>	Captions or subtitles for a video	6.4
<canvas>	A blank surface for JavaScript-drawn graphics	10.3
<svg>	Inline vector graphics	10.3

Tables

Element	Purpose	Section
<table>	A table of tabular data	7.1
<tr>	A table row	7.1
<th>	A header cell	7.1
<td>	A standard data cell	7.1
<caption>	A table's title	7.2
<thead>, <tbody>, <tfoot>	Header, body, and footer row groups	7.2

<colgroup>, <col>	Column grouping, for shared column styling	7.2
-------------------	--	-----

Forms

Element	Purpose	Section
<form>	A container for a group of submittable controls	8.1
<input>	A form control; behavior set by its type attribute	8.2
<label>	A caption associated with a form control	8.2
<textarea>	A multi-line text input	8.2
<select>, <option>	A dropdown list of choices	8.2
<optgroup>	Groups related <option> elements	8.2
<button>	A clickable button	8.2
<fieldset>, <legend>	Groups related form controls with a caption	8.3

Semantic Layout Elements

Element	Purpose	Section
<header>	Introductory content for a page or section	9.2
<nav>	A major navigation block	5.4 / 9.2
<main>	The primary content of the page	9.2

<article>	Self-contained, independently distributable content	9.2
<section>	A thematic grouping of content	9.2
<aside>	Tangentially related content	9.2
<footer>	Closing content for a page or section	9.2
<details>, <summary>	A native, togglable disclosure widget	10.2

This table lists the elements this book covers directly; it is not exhaustive. The Mozilla Developer Network (MDN) HTML Element Reference, linked in Appendix C, documents the complete, current set of HTML elements.

B. Glossary of Terms

Key terms used throughout this book, collected here for quick reference.

Accessibility. The practice of designing content so that it can be used by people with a wide range of abilities, including those using assistive technology such as screen readers.

Attribute. A piece of additional information supplied inside an element's opening tag, such as href on <a> or src on (Section 2.2).

ARIA. Accessible Rich Internet Applications; a set of attributes that describe roles, states, and properties for assistive technology when native HTML alone is not enough (Section 9.3).

Block-level element. An element that begins on a new line and occupies the full available width of its container (Section 4.1).

DOM. Document Object Model; the tree-like, in-memory representation a browser builds from an HTML document, and the structure that JavaScript reads and modifies (Section 2.1).

DOCTYPE. A declaration at the very top of an HTML document that tells the browser which version of HTML to use for rendering (Section 2.4).

Element. A complete HTML unit, generally consisting of an opening tag, content, and a closing tag (Section 2.2).

Fragment identifier. A # followed by an id in a URL, instructing the browser to scroll to the matching element (Section 5.3).

Inline element. An element that flows within a line of text, without starting a new line (Section 4.2).

Landmark. A region of a page identified by a semantic element or ARIA role, such as navigation or main content, that assistive technology can jump between directly (Section 9.2).

Progressive enhancement. A design strategy that builds a functional baseline experience first, then layers on enhancements for browsers that support them (Section 11.2).

Relative URL. A URL resolved against the location of the current document, rather than specifying a full address (Section 5.2).

Semantic element. An element whose name describes the meaning of its content, rather than only its appearance (Section 9.1).

SEO. Search Engine Optimization; the practice of making a page easier for search engines to find, understand, and rank (Section 9.4).

Tag. The markup syntax, such as <p> or </p>, used to open or close an element (Section 2.2).

Validation. Checking a document against the official HTML specification to catch structural errors, typically with the W3C Markup Validator (Section 11.3).

Viewport. The visible area of a web page within the browser window; on mobile devices, its scaling is controlled by the viewport meta tag (Section 11.2).

Void element. An element with no closing tag and no content, such as or
 (Section 4.2).

W3C. The World Wide Web Consortium, one of the organizations responsible for developing web standards (Section 1.3).

WHATWG. The Web Hypertext Application Technology Working Group, which develops the HTML Living Standard (Section 10.1).

C. Further Reading and Online Resources

The following resources are useful both while working through this book and afterward, as HTML continues to evolve.

Official Specifications

- WHATWG HTML Living Standard (html.spec.whatwg.org) — the authoritative, continuously updated HTML specification described in Section 10.1;
- W3C (w3.org) — one of the organizations behind core web standards, including earlier HTML and CSS specifications.

Reference and Learning Platforms

- MDN Web Docs (developer.mozilla.org) — comprehensive, example-rich documentation for HTML, CSS, and JavaScript, including up-to-date browser compatibility data;
- web.dev (web.dev) — Google's collection of guides on modern web development, with a strong focus on performance and accessibility;
- freeCodeCamp (freecodecamp.org) — free, project-based courses covering HTML, CSS, and JavaScript from the ground up.

Tools

- W3C Markup Validator (validator.w3.org) — validates HTML documents against the official specification, as covered in Section 11.3;
- Can I Use (caniuse.com) — up-to-date browser support tables for HTML, CSS, and JavaScript features, referenced in Section 11.2;
- Browser developer tools — built into every major browser, covered throughout this book starting in Section 2.3.

Accessibility Resources

- Web Content Accessibility Guidelines, WCAG ([w3.org/WAI/standards-guidelines/wcag](https://www.w3.org/WAI/standards-guidelines/wcag)) — the standard reference for web accessibility requirements;
- WAI-ARIA Authoring Practices Guide ([w3.org/WAI/ARIA/apg](https://www.w3.org/WAI/ARIA/apg)) — detailed patterns for building accessible interactive components, extending Section 9.3;
- The A11Y Project (a11y-project.com) — a community-driven, practical accessibility knowledge base.

As with any list of online resources, addresses and specific content may change over time; searching for the resource by name will generally lead to its current location even if a specific address above becomes outdated.