Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION)

STYLING WITH CSS (WEEKS 4-6)

SELECTORS, PROPERTIES, AND VALUES

CHAPTER 1: INTRODUCTION TO CSS SELECTORS

1.1 What Are CSS Selectors?

CSS selectors define the pattern used to select and style HTML elements. They allow developers to target specific elements on a webpage and apply styles efficiently.

- **Types of CSS Selectors:**
  - **Universal Selector (*)** – Applies styles to all elements.
  - **Type Selector (h1, p, div)** – Targets specific HTML elements.
  - **Class Selector (.class-name)** – Targets elements with a specific class.
  - **ID Selector (#id-name)** – Targets a unique element with a specific ID.
- **Example:**

```
* {
  margin: 0;
```

```
 padding: 0;
} /* Universal Selector */

h1 {
 color: blue;
} /* Type Selector */

.highlight {
 background-color: yellow;
} /* Class Selector */

#main-header {
 font-size: 24px;
} /* ID Selector */
```

These selectors help organize styles efficiently and improve code maintainability.

CHAPTER 2: ADVANCED CSS SELECTORS

2.1 Attribute and Pseudo-Class Selectors

In addition to basic selectors, CSS provides **attribute selectors** and **pseudo-classes** for more precise targeting.

- **Attribute Selectors:**
  - [type="text"] – Selects all input fields with type="text".

- [href^="https"] – Selects all links that start with https.

◆ **Pseudo-Class Selectors:**

- :hover – Applies styles when an element is hovered.
- :nth-child(n) – Targets the nth child of a parent.

◆ **Example:**

```
input[type="text"] {
 border: 2px solid green;
} /* Targets text input fields */

a[href^="https"] {
 color: red;
} /* Targets secure links */

p:nth-child(2) {
 font-weight: bold;
} /* Targets the second paragraph in a parent element */
```

2.2 Pseudo-Elements and Combinators

◆ **Pseudo-Elements:**

- ::before – Adds content before an element.
- ::after – Adds content after an element.

◆ **Combinators:**

- div > p – Selects p elements that are direct children of a div.

- ul li – Selects all li elements inside a ul.

🔹 **Example:**

```
h1::before {
  content: " 🔥 ";
} /* Adds emoji before heading */

h1::after {
  content: " 🔥 ";
} /* Adds emoji after heading */

div > p {
  color: blue;
} /* Styles only direct children paragraphs */
```

Case Study: How Amazon Optimizes CSS Selectors for Performance

Background

Amazon's website uses optimized CSS selectors to ensure fast performance, smooth styling, and maintainability.

Challenges Faced

- Heavy use of multiple stylesheets affected page load time.
- Unoptimized selectors increased CSS parsing time.

Solutions Implemented

- Used **specific selectors** instead of universal selectors for better performance.
- Minimized excessive **ID and class usage** to avoid specificity conflicts.

🔹 **Key Takeaways from Amazon's Success:**

- **Efficient selectors** improve website performance and load speed.
- Using **structured CSS rules** avoids conflicts and improves maintainability.

📝 **Exercise**

✅ Select three elements on a webpage and apply different types of selectors.
✅ Experiment with pseudo-elements (::before, ::after) and create a custom tooltip using CSS.
✅ Analyze a popular website's CSS using browser developer tools and list down the selectors used.

Conclusion

- CSS selectors allow **efficient targeting of elements,** improving style management.
- Using **pseudo-classes and pseudo-elements** enhances user interaction and styling capabilities.
- Optimizing selectors ensures **better performance and maintainability** in web projects.

THE BOX MODEL & DISPLAY PROPERTIES

CHAPTER 1: UNDERSTANDING THE CSS BOX MODEL

1.1 What is the CSS Box Model?

The CSS Box Model is the fundamental layout structure of every HTML element. It defines how elements are displayed, sized, and interact with one another in a webpage.

- **The Four Components of the Box Model:**

1. **Content** – The actual text or image inside an element.
2. **Padding** – The space between the content and the border.
3. **Border** – The outer edge surrounding the padding and content.
4. **Margin** – The space between an element and its neighboring elements.

- **Example:**

Consider the following CSS:

```
.box {
  width: 200px;
  height: 100px;
  padding: 20px;
  border: 5px solid black;
  margin: 10px;
}
```

Here's how the element's total size is calculated:

- **Width:** 200px (content) + 20px (left padding) + 20px (right padding) + 5px (left border) + 5px (right border) = 250px
- **Height:** 100px (content) + 20px (top padding) + 20px (bottom padding) + 5px (top border) + 5px (bottom border) = 150px

This demonstrates how margins, padding, and borders affect the total size of an element.

CHAPTER 2: CONTROLLING BOX SIZING AND SPACING

2.1 Box Sizing Property

By default, the total width and height of an element **include only the content**, while padding and border **add extra space**. The box-sizing property allows us to change this behavior.

- **Common Values for box-sizing:**
  - **content-box (default):** Only the content area is counted in width/height.
  - **border-box:** Includes padding and border in the width/height calculation.
- **Example:**

```
.box1 {
  width: 200px;
  box-sizing: content-box; /* Default behavior */
}

.box2 {
  width: 200px;
```

```
 box-sizing: border-box; /* Includes padding and border */
}
```

Using border-box prevents unexpected element growth when adding padding and borders.

2.2 Controlling Margins and Padding

- **Margins affect spacing between elements**, while **padding affects spacing inside an element**.
  - margin: auto; centers an element horizontally.
  - padding: 0 20px; applies 20px padding only to the left and right.
- **Example:**

```
.container {
 width: 50%;
 margin: auto; /* Centers the container */
 padding: 10px 20px; /* Adds internal spacing */
}
```

CHAPTER 3: CSS DISPLAY PROPERTIES

3.1 Block, Inline, and Inline-Block Elements

Every HTML element has a **default display property** that determines how it behaves in the document flow.

- **Common Display Values:**
  - **Block (display: block;)** – Takes up the full width (e.g., , <p>, <h1>).

- **Inline (display: inline;)** – Stays within the same line without breaking (e.g., , <a>).
- **Inline-Block (display: inline-block;)** – Behaves like inline but allows width/height adjustments.

◆ **Example:**

```
.block {
  display: block;
  background-color: lightblue;
}

.inline {
  display: inline;
  background-color: lightgreen;
}

.inline-block {
  display: inline-block;
  width: 100px;
  background-color: lightcoral;
}
```

3.2 Using Display: Flex and Grid

◆ **Flexbox (display: flex;)** is used for arranging elements in a row or column.

- **Grid (display: grid;)** is used for defining rows and columns for complex layouts.

- **Example:**

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

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

Case Study: How Google Uses the Box Model for Responsive Layouts

Background

Google's search results page is a prime example of **efficient use of margins, padding, and box sizing** to create a user-friendly experience.

Challenges Faced

- Ensuring content remains readable on **small and large screens**.
- Optimizing spacing to **reduce clutter** while maintaining a clean layout.

Solutions Implemented

- Used **border-box sizing** to prevent unexpected element growth.
- Applied **consistent padding and margin values** to balance spacing.
- Utilized **display: flex** to arrange search results dynamically.

🔹 **Key Takeaways from Google's Success:**

- **Proper use of spacing and box-sizing** creates visually appealing layouts.
- **Flexbox and Grid help maintain structure** across various screen sizes.

📝 Exercise

✅ Experiment with different **box-sizing** properties and observe how padding and borders affect an element's total size.
✅ Convert an existing webpage layout using **display: flex** or **display: grid**.
✅ Inspect a popular website using browser **developer tools** and analyze how margins and padding are applied.

Conclusion

- The **CSS Box Model** determines how elements are sized and spaced on a webpage.
- Using **box-sizing: border-box;** prevents layout shifts when adding padding and borders.

- **Margins and padding control spacing,** affecting element positioning and layout design.
- The **display property** dictates how elements behave within the document flow.
- **Flexbox and Grid Layouts** simplify complex arrangements for modern, responsive designs.

CSS GRID & FLEXBOX FOR RESPONSIVE LAYOUTS

CHAPTER 1: INTRODUCTION TO RESPONSIVE LAYOUTS

1.1 The Need for Responsive Design

Responsive design ensures that web pages adapt to different screen sizes, making them accessible across devices like desktops, tablets, and mobile phones.

- **Why Responsive Design Matters:**
  - Enhances **user experience** by providing consistent layouts.
  - **SEO benefits** as Google prioritizes mobile-friendly websites.
  - Reduces **development time** by avoiding multiple versions of a site.
- **Example:**

A website that is **not responsive** may look perfect on a desktop but broken on a mobile screen. By using **CSS Grid or Flexbox,** we can create adaptable layouts.

CHAPTER 2: CSS FLEXBOX

2.1 What is Flexbox?

Flexbox (Flexible Box Layout) is a one-dimensional layout model that allows elements to be aligned efficiently in a row or column.

- **Key Features of Flexbox:**
  - Distributes space **evenly** among elements.

- Allows **vertical and horizontal alignment** without using float.
- Makes layouts more **adaptive and flexible**.

◆ **Basic Syntax:**

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

This enables flexbox on the container, allowing direct child elements to be positioned dynamically.

2.2 Main Flexbox Properties

◆ **justify-content** – Controls horizontal alignment.

- center – Centers elements in the flex container.
- space-between – Distributes elements with equal space between them.
- space-around – Spaces elements evenly with padding around each.

◆ **Example:**

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

This will center all items inside the flex container horizontally.

◆ **align-items** – Controls vertical alignment.

- stretch (default) – Stretches items to fit the container height.

- center – Aligns items in the middle of the container.

◆ **Example:**

```
.container {
 display: flex;
 align-items: center;
}
```

This centers elements **vertically** within the flex container.

CHAPTER 3: CSS GRID

3.1 What is CSS Grid?

CSS Grid Layout is a **two-dimensional system** that allows developers to control both rows and columns for structured web designs.

◆ **Key Features of CSS Grid:**

- Works **both horizontally and vertically** (unlike flexbox).
- Simplifies complex layouts **without extra divs**.
- Provides **precise control** over element positioning.

◆ **Basic Syntax:**

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

This divides the container into **three equal columns,** each occupying 1fr (fractional unit of available space).

3.2 Common Grid Properties

- **grid-template-columns & grid-template-rows** – Defines the structure of the grid.

```
.container {
 display: grid;
 grid-template-columns: 200px 1fr 1fr;
 grid-template-rows: auto;
}
```

This creates a **three-column layout** where:

- The first column is **fixed at 200px**.
- The other two **adjust based on available space (1fr)**.

- **gap** – Controls spacing between grid items.

```
.container {
 display: grid;
 grid-gap: 10px;
}
```

This adds a **10px gap** between rows and columns.

- **Placing Items in the Grid:**

Elements inside a grid can be **explicitly placed** using grid-column and grid-row.

```
.item {
```

```
  grid-column: 1 / span 2;
  grid-row: 1 / span 2;
}
```

This makes the .item **occupy two columns and two rows**.

Case Study: Netflix's Responsive Layout Using Flexbox & Grid

Background

Netflix provides a **seamless viewing experience** across devices, from desktops to smartphones.

Challenges Faced

- Displaying **dynamic content grids** (movie thumbnails).
- Adjusting layout based on **screen size and resolution**.

Solutions Implemented

- Used **Flexbox for navigation bars** to ensure dynamic resizing.
- Used **CSS Grid for content sections**, making movie thumbnails adjust to different screen sizes.
- Implemented **media queries** for mobile, tablet, and desktop versions.

◆ **Key Takeaways from Netflix's Success:**

- **Combining Flexbox & Grid** helps in building dynamic and responsive layouts.
- **Grid is ideal for structuring content sections**, while **Flexbox is great for UI elements like menus and buttons**.

📝 Exercise

✅ Create a **navigation bar** using display: flex; with three menu items evenly spaced.
✅ Build a **grid-based image gallery** where each image takes up an equal portion of the screen.
✅ Use **CSS media queries** to modify the layout for different screen sizes.

Conclusion

- **Flexbox is best for one-dimensional layouts** (either rows or columns), while **Grid is ideal for two-dimensional layouts** (both rows and columns).
- **Flexbox properties like justify-content and align-items** help position elements efficiently.
- **CSS Grid provides greater control** over structure, allowing easy row and column management.
- **Using Flexbox and Grid together** creates powerful and responsive web designs.

POSITIONING ELEMENTS & OVERLAYS

CHAPTER 1: UNDERSTANDING CSS POSITIONING

1.1 The Importance of Positioning in Web Design

Positioning in CSS determines how elements are placed on a webpage. Proper positioning improves **layout structure, usability, and responsiveness**, allowing elements to be aligned, stacked, or layered dynamically.

- **Why CSS Positioning is Important:**
  - Helps **align content** properly within a section.
  - Allows elements to **overlap or remain fixed** while scrolling.
  - Used for **pop-ups, sticky headers, and floating buttons**.
- **Example:**

A **floating chat button** remains fixed at the bottom right of a page using positioning.

```
.chat-button {
  position: fixed;
  bottom: 20px;
  right: 20px;
  background-color: blue;
  color: white;
  padding: 10px 15px;
  border-radius: 50%;
}
```

This ensures the chat button remains **fixed** regardless of scrolling.

CHAPTER 2: CSS POSITIONING TYPES

2.1 Different Types of Positioning

- **static (default)** – Elements follow the normal document flow.
- **relative** – Position is adjusted **relative to its normal position**.
- **absolute** – Positioned **relative to the nearest positioned ancestor**.
- **fixed** – Stays fixed on the screen even while scrolling.
- **sticky** – Switches between **relative and fixed** based on scrolling.
- **Example:**

```
.relative-box {
  position: relative;
  top: 20px;
  left: 50px;
} /* Moves 20px down and 50px right */


.absolute-box {
  position: absolute;
  top: 30px;
  right: 10px;
} /* Positioned relative to the nearest positioned ancestor */
```

2.2 Using Z-Index for Layering Elements

- **z-index** controls the stacking order of elements. A higher z-index means an element appears **above** others.

- **Example:**

```
.modal {
 position: absolute;
 z-index: 10;
} /* This will appear above elements with a lower z-index */
```

CHAPTER 3: OVERLAYS AND BACKGROUND EFFECTS

3.1 Creating Full-Screen Overlays

Overlays are **semi-transparent layers** that appear above content, commonly used for pop-ups or modal windows.

- **Steps to Create an Overlay:**
  - Use position: fixed; to **cover the entire viewport**.
  - Apply background-color: rgba(0, 0, 0, 0.5); for a **semi-transparent effect**.
  - Use **high z-index values** to keep it above other elements.

- **Example:**

```
.overlay {
 position: fixed;
 top: 0;
 left: 0;
 width: 100%;
```

```
 height: 100%;
 background-color: rgba(0, 0, 0, 0.5);
 z-index: 1000;
}
```

3.2 Creating a Pop-up Modal with CSS

A modal window is a **floating element that appears above the main content**.

- **Example:**

```
.modal {
 position: fixed;
 top: 50%;
 left: 50%;
 transform: translate(-50%, -50%);
 background: white;
 padding: 20px;
 box-shadow: 0 4px 10px rgba(0, 0, 0, 0.3);
 z-index: 9999;
}
```

This centers the modal and ensures it appears above other content.

Case Study: Facebook's Use of Overlays for UX Enhancement

Background

Facebook uses overlays effectively for pop-ups, chat windows, and media previews.

Challenges Faced

- Ensuring pop-ups do not obstruct important content.
- Allowing users to **dismiss overlays easily** while keeping them visually engaging.

Solutions Implemented

- Used **fixed positioning** to create floating chat windows.
- Applied **semi-transparent overlays** for notifications and modals.
- **Enabled smooth animations** for opening and closing pop-ups.

🔹 **Key Takeaways from Facebook's Success:**

- **Positioning and layering (z-index) help create effective UI elements**.
- **Well-designed overlays improve user experience without disrupting content**.

📝 Exercise

✅ Create a **fixed navigation bar** that remains at the top while scrolling.
✅ Build a **modal pop-up** using position: fixed; and z-index.
✅ Experiment with **overlays and background effects** by creating a lightbox effect for images.

Conclusion

- **CSS positioning (static, relative, absolute, fixed, sticky)** controls how elements are placed.
- **Using z-index properly** ensures correct layering of elements in overlays and modals.
- **Overlays and pop-ups** enhance UI but must be used strategically to avoid disrupting user experience.
- **CSS properties like position: fixed; and transform: translate();** help create dynamic floating elements.

COLORS, GRADIENTS, SHADOWS & BORDERS

CHAPTER 1: UNDERSTANDING CSS COLORS

1.1 The Role of Colors in Web Design

Colors play a vital role in **enhancing user experience, setting brand identity, and improving readability**. CSS provides multiple ways to define colors.

- **Ways to Define Colors in CSS:**
  - **Named Colors** – red, blue, green
  - **Hex Codes** – #ff5733, #3498db
  - **RGB Values** – rgb(255, 87, 51)
  - **HSL (Hue, Saturation, Lightness)** – hsl(12, 100%, 50%)
- **Example:**

```
h1 {
 color: #ff5733; /* Hex */
}

p {
 color: rgb(34, 45, 67); /* RGB */
}

button {
```

```
 background-color: hsl(200, 80%, 50%); /* HSL */
}
```

1.2 Using Transparency and Opacity

- **opacity** controls the **transparency** of an element.

```
.box {
 background-color: blue;
 opacity: 0.5; /* 50% transparent */
}
```

- **rgba() and hsla()** allow **adjusting opacity** while keeping the background color intact.

```
button {
 background-color: rgba(255, 0, 0, 0.5); /* Red with 50% opacity */
}
```

CHAPTER 2: CSS GRADIENTS

2.1 Linear and Radial Gradients

Gradients create **smooth transitions between two or more colors**.

- **Linear Gradient (Horizontal & Vertical)**

```
.box {
 background: linear-gradient(to right, red, yellow);
}
```

- **Radial Gradient (Circular Spread from the Center)**

```
.circle {
  background: radial-gradient(circle, blue, white);
}
```

2.2 Advanced Gradient Effects

- **Adding multiple color stops**

```
.gradient {
  background: linear-gradient(to right, red, orange, yellow, green, blue);
}
```

- **Repeating Gradients for Patterns**

```
.box {
  background: repeating-linear-gradient(45deg, blue, white 10%);
}
```

CHAPTER 3: CSS SHADOWS & BORDERS

3.1 Box Shadows and Text Shadows

- **Box Shadows for Depth Effects**

```
.box {
  box-shadow: 5px 5px 15px rgba(0, 0, 0, 0.3);
}
```

- **Text Shadows for Typography Enhancements**

```
h1 {
  text-shadow: 2px 2px 5px rgba(0, 0, 0, 0.5);
```

```
}
```

3.2 Borders and Border Radius

- **Customizing Borders**

```
.box {
 border: 3px solid black;
}
```

- **Rounded Corners with border-radius**

```
.button {
 border-radius: 50px;
}
```

Case Study: How Apple Uses Colors & Shadows for Visual Appeal

Background

Apple's website uses **soft gradients, shadows, and borders** to create a clean, modern aesthetic.

Challenges Faced

- Maintaining **consistent brand identity** across devices.
- Creating **depth and focus** using shadows and gradients.

Solutions Implemented

- Used **linear gradients for call-to-action buttons**.
- Applied **subtle box shadows** for product images.

- **Key Takeaways from Apple's Success:**

- **Shadows and gradients create a sense of depth and realism.**
- **Consistent color schemes enhance branding and usability.**

📝 Exercise

✅ Create a **gradient background** that smoothly transitions between three colors.
✅ Apply **box-shadow and text-shadow** to create a 3D button effect.
✅ Design a **rounded border button with hover effects**.

Conclusion

- **CSS colors, gradients, shadows, and borders** enhance visual aesthetics.
- **Linear and radial gradients** create smooth color transitions.
- **Shadows improve depth perception** in UI elements.
- **Borders and border-radius** help create modern, sleek designs.

CSS TRANSITIONS, TRANSFORMATIONS, AND ANIMATIONS

CHAPTER 1: CSS TRANSITIONS

1.1 How Transitions Work in CSS

CSS transitions enable **smooth property changes** over time when elements are hovered or interacted with.

- **Basic Transition Syntax**

```
.button {
 background-color: blue;
 transition: background-color 0.5s ease-in-out;
}

.button:hover {
 background-color: red;
}
```

1.2 Transition Timing Functions

- **ease** – Starts slow, speeds up, then slows down.
- **linear** – Maintains constant speed.
- **ease-in** – Starts slow, then speeds up.
- **Example:**

```
.box {
 transition: transform 1s ease-in-out;
```

}

CHAPTER 2: CSS TRANSFORMATIONS

2.1 Scaling and Rotating Elements

CSS transforms allow **modifying the size, rotation, and position of elements**.

- **Scaling (scale())**

```
.box {
 transform: scale(1.5);
}
```

- **Rotating (rotate())**

```
.box {
 transform: rotate(45deg);
}
```

2.2 Skewing and Translating Elements

- **Skew (skew())**

```
.box {
 transform: skewX(30deg);
}
```

- **Translating (translate())**

```
.box {
 transform: translate(50px, 20px);
```

```
}
```

CHAPTER 3: CSS ANIMATIONS

3.1 Creating Keyframe Animations

CSS animations allow **more complex movement effects** using keyframes.

- **Basic Animation Syntax**

```
@keyframes slide {
 from { transform: translateX(0); }
 to { transform: translateX(100px); }
}


.box {
 animation: slide 2s infinite alternate;
}
```

3.2 Combining Multiple Animation Properties

- **Delays and Loops**

```
.box {
 animation: slide 3s ease-in-out 1s infinite;
}
```

Case Study: Google's Use of CSS Animations in Material Design

Background

Google's Material Design framework uses **smooth animations for buttons, menus, and cards**.

Challenges Faced

- Creating **fluid, natural motion**.
- Ensuring **animations do not slow down performance**.

Solutions Implemented

- Used **subtle transitions for buttons and input fields**.
- Applied **transformations to create depth effects**.

🔹 **Key Takeaways from Google's Success:**

- **Animations should be subtle and functional.**
- **Overuse of animations can negatively impact UX.**

📝 Exercise

✅ Create a **hover effect** using transition: all 0.3s ease-in-out;.
✅ Design an **animated loader** using keyframes.
✅ Apply **rotate and scale transforms** on an image gallery.

Conclusion

- **CSS transitions** enable smooth property changes.
- **Transformations modify the size, rotation, and position of elements.**
- **Animations create engaging UI effects but should be used wisely.**

- **Material Design by Google is a prime example of effective CSS animations.**

ASSIGNMENT:

BUILD A RESPONSIVE WEBPAGE USING CSS WITH ANIMATIONS AND LAYOUT TECHNIQUES.

Step-by-Step Guide to Building a Responsive Webpage Using CSS with Animations and Layout Techniques

Step 1: Setting Up the Project Structure

Before we start coding, create the necessary files:

1. **Create a project folder** (e.g., responsive-webpage).
2. **Inside the folder, create the following files:**
    - index.html (for the webpage structure)
    - style.css (for styling the page)
    - script.js (optional, for interactivity)

Step 2: Writing the HTML Structure

Open index.html and write the basic structure of the webpage:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Responsive Webpage</title>
  <link rel="stylesheet" href="style.css">
```

```
</head>
<body>

  <!-- Navigation Bar -->
  <nav class="navbar">
    <div class="logo">MyWebsite</div>
    <ul class="nav-links">
      <li><a href="#">Home</a></li>
      <li><a href="#">About</a></li>
      <li><a href="#">Services</a></li>
      <li><a href="#">Contact</a></li>
    </ul>
  </nav>

  <!-- Hero Section -->
  <header class="hero">
    <h1 class="fade-in">Welcome to My Website</h1>
    <p class="slide-in">Creating beautiful, responsive web pages with CSS</p>
    <a href="#" class="btn">Get Started</a>
  </header>

  <!-- Services Section -->
```

```
  <section class="services">
    <h2>Our Services</h2>
    <div class="services-grid">
      <div class="service-card">Web Design</div>
      <div class="service-card">SEO Optimization</div>
      <div class="service-card">App Development</div>
    </div>
  </section>

  <!-- Footer -->
  <footer class="footer">
    <p>© 2024 MyWebsite. All rights reserved.</p>
  </footer>

</body>
</html>
```

Step 3: Styling with CSS

Open style.css and define the styles for different sections.

3.1 Reset and Global Styles

```
* {
  margin: 0;
```

```
  padding: 0;
  box-sizing: border-box;
  font-family: Arial, sans-serif;
}

body {
  background-color: #f8f9fa;
  color: #333;
  text-align: center;
}
```

3.2 Styling the Navigation Bar (Flexbox Layout)

```
.navbar {
  display: flex;
  justify-content: space-between;
  align-items: center;
  padding: 15px 30px;
  background-color: #333;
  color: white;
}

.logo {
```

```
  font-size: 24px;
  font-weight: bold;
}

.nav-links {
  list-style: none;
  display: flex;
}

.nav-links li {
  margin: 0 15px;
}

.nav-links a {
  color: white;
  text-decoration: none;
  font-size: 18px;
}
```


3.3 Styling the Hero Section with Background Image & Animations

```
.hero {
```

```
  background:
url('https://source.unsplash.com/1600x900/?nature,technology') no-
repeat center center/cover;
  height: 100vh;
  display: flex;
  flex-direction: column;
  justify-content: center;
  align-items: center;
  color: white;
  text-shadow: 2px 2px 5px rgba(0, 0, 0, 0.5);
}

.hero h1 {
  font-size: 50px;
  margin-bottom: 10px;
  opacity: 0;
  animation: fadeIn 2s forwards;
}

.hero p {
  font-size: 20px;
  opacity: 0;
  animation: slideIn 2s forwards;
```

```
}

.btn {
  background: #ff6f61;
  color: white;
  padding: 10px 20px;
  text-decoration: none;
  font-size: 20px;
  border-radius: 5px;
  margin-top: 20px;
  display: inline-block;
  transition: transform 0.3s ease-in-out;
}

.btn:hover {
  transform: scale(1.1);
}
```

3.4 Adding Animations for Hero Section

```
@keyframes fadeIn {
  from {
    opacity: 0;
```

```
  }
  to {
    opacity: 1;
  }
}

@keyframes slideIn {
  from {
    transform: translateY(20px);
    opacity: 0;
  }
  to {
    transform: translateY(0);
    opacity: 1;
  }
}
```

3.5 Creating the Services Section with CSS Grid

```
.services {
  padding: 50px;
  background: #fff;
}
```

```
.services h2 {
  font-size: 32px;
  margin-bottom: 20px;
}

.services-grid {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(250px, 1fr));
  gap: 20px;
}

.service-card {
  background: #ff6f61;
  color: white;
  padding: 20px;
  font-size: 20px;
  border-radius: 8px;
  text-align: center;
  transition: transform 0.3s ease-in-out;
}
```

```
.service-card:hover {
  transform: translateY(-10px);
}
```

3.6 Footer Styling

```
.footer {
  background: #333;
  color: white;
  padding: 15px;
  text-align: center;
}
```

Step 4: Making the Webpage Responsive

To ensure the webpage works well on all screen sizes, add **media queries** in style.css.

```
@media (max-width: 768px) {
  .navbar {
    flex-direction: column;
  }

  .nav-links {
    flex-direction: column;
    padding: 10px 0;
```

```
  }

  .nav-links li {
    margin: 5px 0;
  }

  .hero h1 {
    font-size: 36px;
  }

  .hero p {
    font-size: 18px;
  }

  .services-grid {
    grid-template-columns: 1fr;
  }
}
```

Step 5: Adding a Simple JavaScript Animation (Optional)

To **add a fade-in effect when scrolling,** open script.js and add:

```
document.addEventListener("DOMContentLoaded", function () {
```

```
  const elements = document.querySelectorAll(".service-card");

  function checkScroll() {
    elements.forEach((el) => {
      const position = el.getBoundingClientRect().top;
      if (position < window.innerHeight - 50) {
        el.style.opacity = "1";
        el.style.transform = "translateY(0)";
      }
    });
  }

  window.addEventListener("scroll", checkScroll);
});
```

In style.css, add:

```
.service-card {
  opacity: 0;
  transform: translateY(20px);
  transition: all 0.5s ease-in-out;
}
```

Final Preview of Features Implemented

✓ **Responsive Navigation Bar** (Flexbox)
✓ **Hero Section with Animations** (@keyframes)
✓ **Service Section Using CSS Grid**
✓ **Hover Effects and Transitions**
✓ **Mobile-Friendly with Media Queries**
✓ **Simple JavaScript Animation for Scrolling Effects**

Conclusion

This step-by-step guide **teaches you how to build a modern, responsive webpage** using CSS **Flexbox, Grid, Transitions, and Animations**. By following these steps, you can create engaging and interactive designs that adapt seamlessly across different screen sizes.
