HTML & CSS
Basics Guide

Chapters Overview

Chapter 1 – Foundations of HTML and CSS

1.1 What is HTML and CSS?
1.2 Evolution of HTML (HTML4, XHTML, HTML5)
1.3 Page Structure and the Role of the DOM
1.4 How CSS Styles Are Applied
1.5 Embedding CSS: Inline, Internal, External

Chapter 2 – HTML5 Semantic Structure and Media

2.1 Semantic Layout Elements
2.2 Common Text and Structural Elements
2.3 Embedding Media: Images, Audio, Video, Iframes
2.4 Forms and User Inputs in HTML5

Chapter 3 – Core CSS Concepts and Layout

3.1 CSS Syntax and Selectors
3.2 Specificity and Inheritance
3.3 Typography and Color Styling
3.4 The Box Model Explained
3.5 Display Types and Positioning
3.6 Float and Clear

Chapter 4 – Responsive Design and Modern CSS Techniques

4.1 Mobile-First and Media Queries
4.2 Flexbox: Concepts and Practical Use
4.3 Grid Layout: Fundamentals Only
4.4 Transitions, Transforms, and Basic Animations
4.5 Pseudo-classes and Pseudo-elements
4.6 Custom Fonts with @font-face

Chapter 1 – Foundations of HTML and CSS

1.1 What is HTML and CSS?

HTML (HyperText Markup Language) is the foundational language used to create and structure the content on the web. It provides a way to describe elements like text, images, links, and other media so browsers can render them visually. HTML is not a programming language—it's a markup language designed to annotate content.

CSS (Cascading Style Sheets) is a styling language that controls how HTML elements appear on the screen. CSS defines colors, fonts, layout positioning, spacing, and visual transitions. Without CSS, all web pages would look plain and unstyled.

Why are both necessary?

- HTML answers: *What is on the page?* (e.g., a heading, a paragraph, an image)
- CSS answers: *How does it look?* (e.g., bold, centered, blue, with a shadow)

Together, HTML and CSS allow developers to build web pages that are both **structurally sound** and **visually engaging**.

Real-World Analogy:

Imagine a web page as a house:

- **HTML is the structure**: walls, rooms, doors.
- **CSS is the design**: paint, furniture, lighting.

1.2 Evolution of HTML

Early HTML (HTML 1.0 – 3.2)

In the early 1990s, HTML was very basic—limited to formatting text and adding links. Pages were static, and styling was embedded directly using presentational tags like `<font>` and `<center>`.

HTML 4.01 and XHTML

Released in 1999, **HTML 4.01** added structure and flexibility to the language. It separated content from presentation and introduced support for:

- Tables, forms, and scripting (JavaScript)
- Basic styling through CSS
- The concept of standards compliance

At the same time, **XHTML** (a stricter XML-based version of HTML) emerged. It enforced stricter rules like:

- Proper nesting of elements
- Mandatory closing tags
- Lowercase tag names

However, XHTML was less forgiving and harder to work with in browsers, especially for beginners.

HTML5: The Modern Standard

HTML5, finalized around 2014, replaced both HTML 4.01 and XHTML. Its goals were to:

- Simplify the language
- Support multimedia (video, audio, canvas) without plugins
- Introduce **semantic elements** for better structure
- Improve browser error-handling
- Work better across devices (desktop, tablet, mobile)

HTML5 is **backward compatible** and **flexible**. It lets developers write cleaner, more meaningful code while enhancing accessibility and search engine optimization.

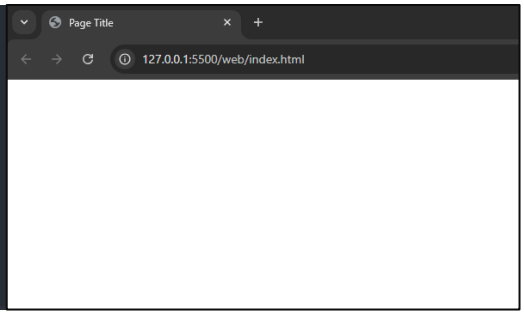
HTML Version	Key Features	Status
HTML 3.2	Tables, forms	Obsolete
HTML 4.01	Structure, CSS	Legacy
XHTML	XML syntax, stricter	Obsolete
HTML5	Modern, semantic, multimedia	Current

1.3 Page Structure and the Role of the DOM

HTML Document Structure

Every HTML page follows a consistent structure that browsers expect. The basic skeleton includes:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Page Title</title>
  </head>
  <body>
    <!-- Visible content goes here -->
  </body>
</html>
```

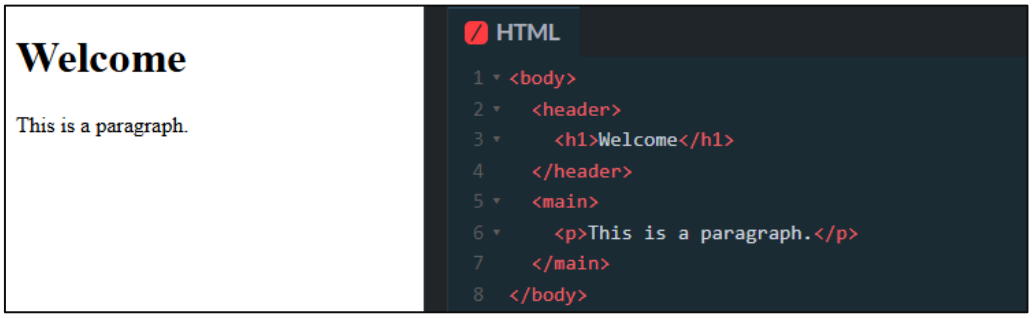


Key sections:

- **`<!DOCTYPE html>`**: Declares the document type as HTML5.
- **`<html>`**: Root element wrapping the entire document.
- **`<head>`**: Contains metadata (like the title, linked CSS, character encoding).
- **`<body>`**: Holds the visible content—text, images, links, etc.

Content Hierarchy

HTML follows a **tree-like hierarchy**. Example:



Here, `<body>` is the parent of `<header>` and `<main>`. Each element can contain other elements, forming a **nested structure**.

What Is the DOM?

DOM (Document Object Model) is the browser's internal representation of the HTML page. It converts the HTML into a **live tree of nodes** that JavaScript and the browser can interact with.

- Elements like `<h1>` or `<p>` become **nodes** in the DOM.
- You can use **JavaScript** to change the DOM dynamically (e.g., hide/show content, change styles, update text).
- CSS also interacts with the DOM, applying styles based on selectors.

Think of the DOM as a **live map of your page** that can be queried, modified, and updated in real time.

Why DOM Matters

- Enables **dynamic interactivity** (buttons, forms, effects)
- Helps browsers **render and update** content efficiently
- Forms the foundation for **JavaScript-driven applications** like single-page apps (SPAs)

1.4 How CSS Styles Are Applied

CSS (Cascading Style Sheets) is used to control the appearance of HTML content. It allows you to define how elements should look—colors, fonts, spacing, layout, and more—separately from the HTML structure.

The Cascade

The "Cascading" part of CSS refers to **how styles are applied when there are multiple conflicting rules**. The browser follows these rules in order:

1. **Source order** – Later styles override earlier ones if they have the same specificity.
2. **Specificity** – More specific selectors override less specific ones.
3. **Importance** – Styles marked `!important` override normal rules (though best avoided).
4. **Origin** – Browser default < Author-defined < User overrides

---

CSS Selectors

Selectors target HTML elements for styling. Examples:

Selector	Description
`p`	All `<p>` tags
`.note`	All elements with class `note`
`#main`	Element with ID `main`
`div > p`	`<p>` elements directly inside a `<div>`
`a:hover`	Links when hovered by the mouse

Style Properties

Each rule consists of a **property** and a **value**:

- `color` and `font-size` are **properties**
- `navy` and `24px` are **values**

How Browsers Apply CSS

1. Parse HTML into the DOM tree
2. Load and apply CSS to elements using selectors
3. Compute the final visual style using the cascade
4. Render the page on screen

Why CSS Is Powerful

- It keeps content and design **separated**
- Allows **global style changes** by editing one file
- Enables **responsive and dynamic designs**

Without CSS, every webpage would look like plain black text on a white screen.

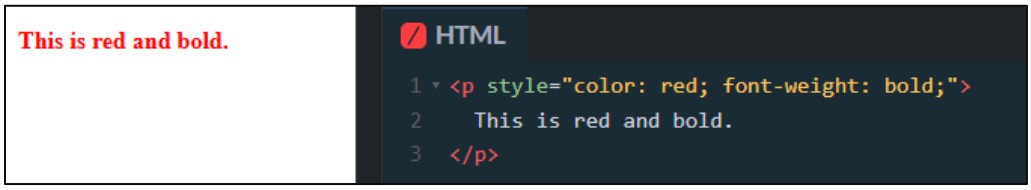
1.5 Embedding CSS: Inline, Internal, External

There are **three main ways** to apply CSS styles to a webpage. Each has its ideal use depending on the situation.

---

1. Inline CSS

Style is added directly to an HTML element using the `style` attribute.



Pros:

- Quick to apply
- Useful for one-time changes or testing

Cons:

- Breaks separation of content and style
- Harder to maintain
- Repetition across multiple elements

2. Internal CSS

Style rules are written inside a `<style>` block in the `<head>` of the HTML document.

Pros:

- Better than inline for styling entire pages
- Keeps style centralized within the same file

Cons:

- Doesn't scale well across multiple pages
- Still mixes style with structure

3. External CSS

Style rules are placed in a separate `.css` file and linked to the HTML using a `<link>` tag.

Pros:

- **Best practice** for large projects
- Clean separation of structure and style
- Reusable across multiple HTML pages
- Easy to maintain and update

Cons:

- Requires extra file load (minimal issue with caching)

Summary Table

Method	Use Case	Maintainability
Inline	Small, quick changes	Low
Internal	One-page projects or tests	Medium
External	Full websites, scalable apps	High (preferred)

 # Best Practice:
Use **external CSS** for long-term projects. Reserve **internal** for quick prototypes and **inline** only for rare exceptions.

2.1 Semantic Layout Elements

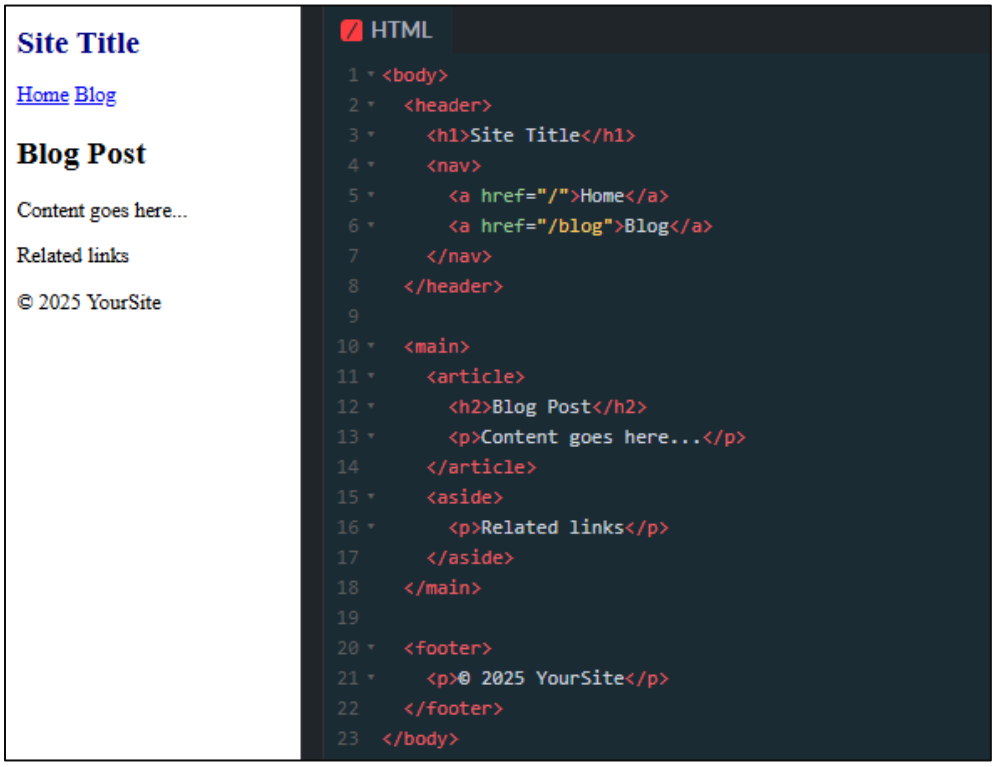
What Are Semantic Elements?

Semantic elements clearly describe their **meaning** both to the browser and the developer. Instead of using generic containers like `<div>`, semantic elements have **descriptive names** that improve readability, accessibility, and SEO.

Common Semantic Layout Elements

Element	Purpose
`header`	Introductory content or navigation for a section
`nav`	Major blocks of navigation links
`main`	Central content unique to the page
`section`	Thematic grouping of related content
`article`	Independent, reusable content (e.g., blog post)
`aside`	Related side content (e.g., ads, sidebars)
`footer`	Bottom section with closing info or navigation

Sample Semantic Page Layout



Why Semantic Elements Matter

- **Improves accessibility**: Screen readers can identify sections clearly
- **Boosts SEO**: Search engines better understand content purpose
- **Enhances maintainability**: Code becomes self-descriptive
- **Future-friendly**: Designed to work well across platforms and devices

Before HTML5, developers relied heavily on `<div id="header">`, `<div id="footer">`, etc. Semantic elements now eliminate this need.

2.2 Common Text and Structural Elements

In addition to layout containers, HTML provides a variety of elements to structure and format the content itself—such as headings, paragraphs, lists, quotes, and inline text enhancements.

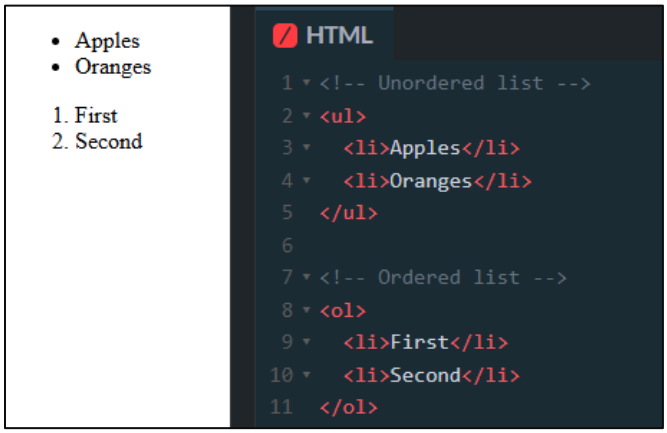
Text Content Elements

Element	Use Case
`h1–h6`	Headings, from most to least important
`p`	Paragraphs of text
`br`	Line breaks (use sparingly)
`hr`	Thematic breaks (horizontal lines)

Headings should follow a **hierarchical order**. Use one `<h1>` per page, typically for the main title.

Lists

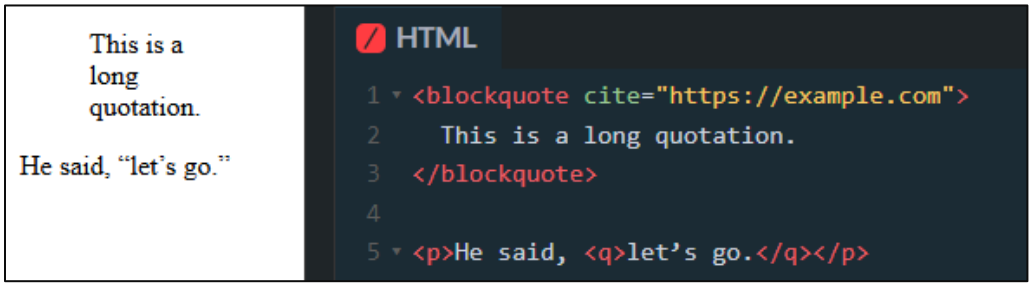
HTML supports two main types of lists:



- `ul` = bullet points
- `ol` = numbered items
- `li` = list item

Blockquotes and Quotes

Use block-level and inline quoting for cited content:

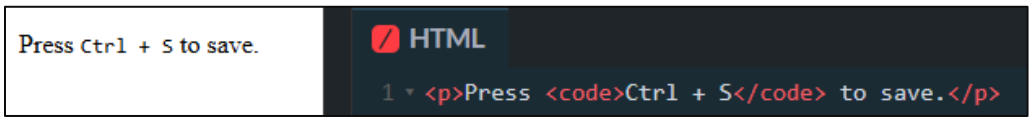


- `blockquote`: for long quotes
- `q`: for inline quotations
- `cite`: to attribute a source

Emphasizing Text

Element	Meaning
`strong`	Strong importance (usually bold)
`em`	Emphasis (usually italic)
`mark`	Highlighted or marked text
`code`	Inline code or commands

Example:



Grouping Content

- `div`: Generic block-level container
- `span`: Generic inline container

These are still useful when semantic elements don't apply, often used with classes for styling.

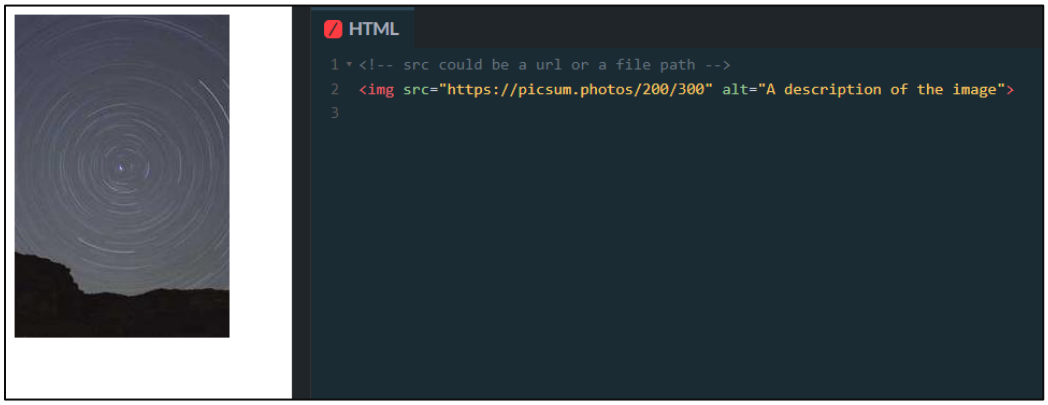
Semantic and properly structured content **improves clarity**, helps **machines understand your content**, and makes your pages more **accessible and responsive**.

2.3 Embedding Media: Images, Audio, Video, Iframes

Modern web pages are no longer just text—they often include **images**, **multimedia**, and **embedded content** from other platforms. HTML5 introduced standard ways to support all of these natively.

Images

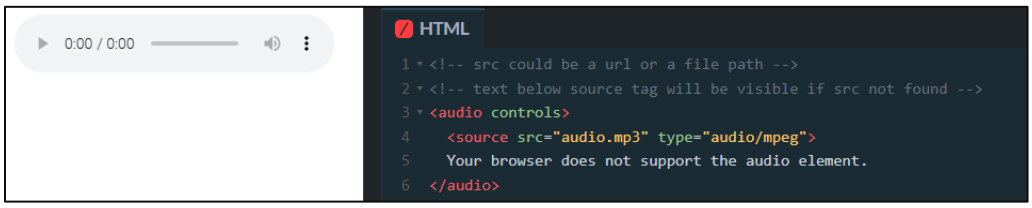
The `<img>` tag embeds images.



- `src`: Path to the image file
- `alt`: Text alternative (important for accessibility and SEO)
- `width` and `height`: Can be set via CSS or as attributes

Audio

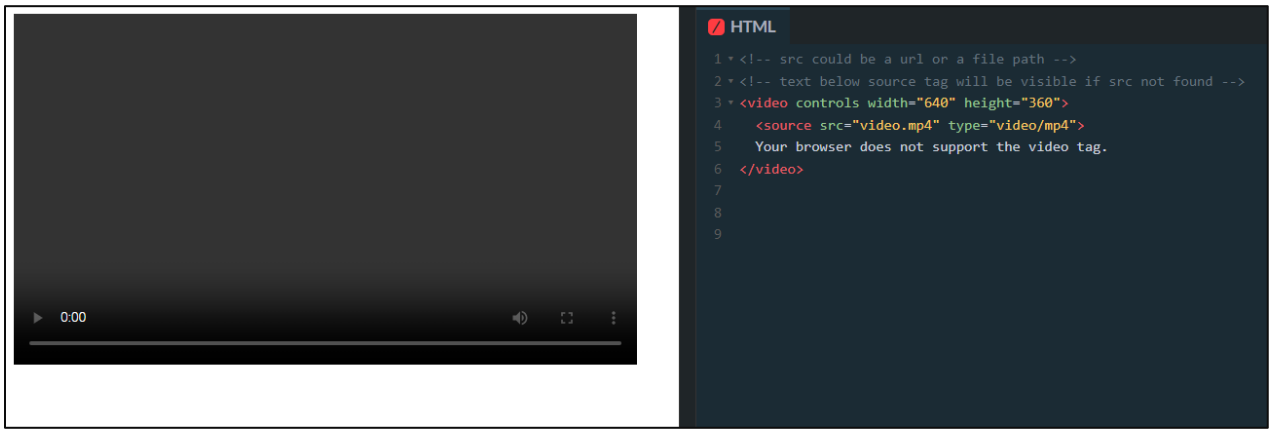
HTML5 introduced the `<audio>` tag for playing sound files.



- `controls`: Shows play/pause UI
- Multiple `<source>` tags allow fallback for different formats
- Common formats: `mp3`, `ogg`, `wav`

Video

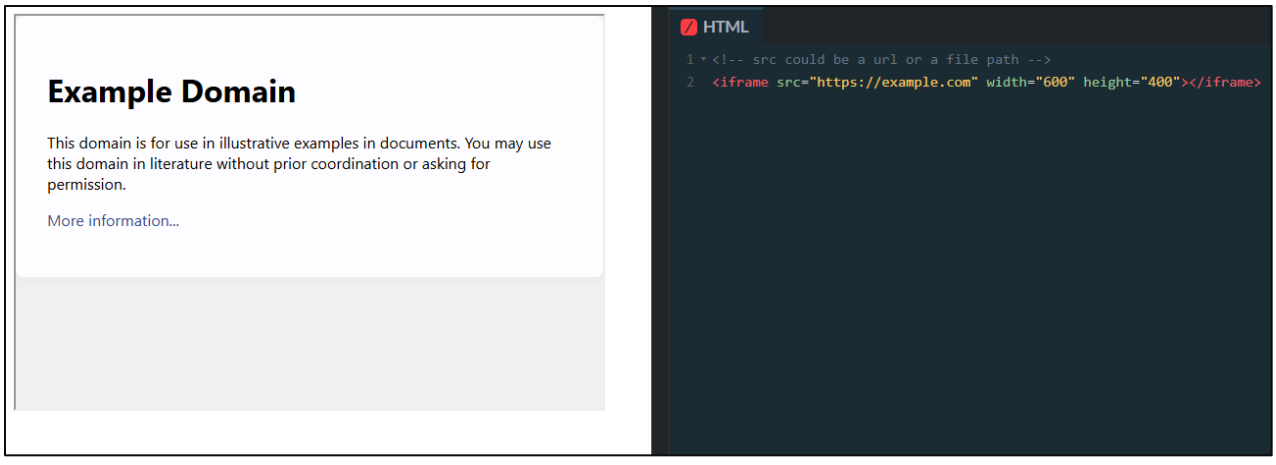
The `<video>` tag supports embedded videos without plugins.



- `controls`: Shows playback controls
- You can also use `autoplay`, `loop`, and `muted` attributes
- Common formats: `mp4`, `webm`, `ogg`

Iframes

Use `<iframe>` to embed external pages, like maps, videos, or widgets.



- Can be styled with CSS

- Use responsibly—iframes may affect performance or pose security risks
- Often used for embedding YouTube, Google Maps, or other third-party tools

---

Accessibility Note

Always include:

- `alt` text for images
- Text fallback inside `<audio>` or `<video>` tags
- Meaningful content around embedded media

Rich media makes your site more engaging but must be used with care to ensure speed, usability, and compatibility.

2.4 Forms and User Inputs in HTML5

Forms are essential for collecting user input—such as logins, search queries, registrations, and feedback. HTML5 introduced **new input types** and **built-in validation** to improve usability and interactivity.

Basic Form Structure

- `action`: Where the form data will be sent
- `method`: `get` (for URL-based queries) or `post` (for secure submission)
- `label`: Associates text with form controls (important for accessibility)

New HTML5 Input Types

HTML5 introduced many **semantic input types** to improve form behavior:

Input Type	Purpose
`text`	Generic single-line text
`email`	Validates email format

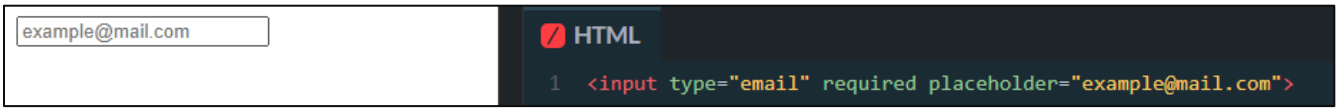
`url`	Validates URL format
`number`	Numeric input, with min/max
`tel`	Phone number (no validation)
`date`	Date picker
`range`	Slider input
`color`	Color picker
`search`	Optimized for search fields
`checkbox`	On/off toggles
`radio`	Multiple-choice options
`file`	File upload control
`password`	Obscured text input

Validation and Attributes

HTML5 allows validation without JavaScript:

- `required`: Field must be filled
- `min`, `max`: Limits for number/date inputs
- `pattern`: Regular expression check
- `placeholder`: Hint text inside fields
- `autocomplete`: Helps browsers remember user input

Example:



Grouping Inputs

- `<fieldset>`: Groups related fields visually and semantically
- `<legend>`: Title for the fieldset group

Accessibility Tips

- Always use `<label>` with `for` attributes
- Use `aria-*` attributes when needed for screen readers

- Test form usability with keyboard navigation

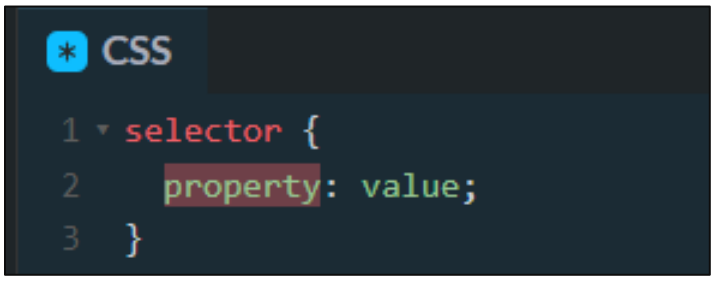
HTML5 makes forms smarter, safer, and easier to build—often **without needing JavaScript** for basic validation.

Chapter 3 – Core CSS Concepts and Layout

3.1 CSS Syntax and Selectors

Understanding how to **target elements** with CSS is fundamental to styling a web page. The combination of **selectors**, **properties**, and **values** allows precise control over the visual presentation of content.

Basic CSS Rule Structure



Example:

- **Selector**: Targets HTML elements
- **Property**: The style you want to change (e.g. `color`)
- **Value**: The setting for that property (e.g. `navy`)

Common Selectors

Selector	Targets
`p`	All `<p>` elements
`.className`	All elements with that class
`#idName`	The element with that ID
`div p`	All `<p>` inside `<div>`
`ul > li`	Direct children `<li>` inside `<ul>`
`a:hover`	Links when hovered

Group Selectors

Apply styles to multiple elements at once:

```css
h1, h2, h3 {
  font-family: sans-serif;
}
```

Universal Selector

Targets everything:

```css
* {
  margin: 0;
  padding: 0;
}
```

Useful for CSS resets.

Attribute Selectors

Target elements based on attribute values:

```css
input[type="text"] {
  border: 1px solid #ccc;
}
```

Pseudo-classes

Style elements based on their state:

Selector	Use Case
`:hover`	When hovered
`:focus`	When focused (e.g., form input)
`:first-child`	First child of a parent
`:nth-child(2)`	Second child, etc.

Best Practices

- Keep selectors **simple and clear**
- Use **class selectors** for reusable styles
- Avoid using IDs for styling (too specific)
- Structure CSS from general to specific for easier overrides

3.2 Specificity and Inheritance

When multiple CSS rules apply to the same element, the browser must decide **which style takes precedence**. This is controlled by **specificity**, the **cascade**, and whether a property is **inherited**.

Understanding Specificity

Specificity is a scoring system based on how a rule is written:

Selector Type	Specificity Score
Inline styles	1000
ID selectors	100
Class selectors	10
Element selectors	1

Example:

CSS

```css
/* Specificity: 11 (10 + 1) */
p.note {
    color: green;
```

.0

If multiple rules conflict, the one with the **higher specificity** wins.

How Specificity Works

Hello!

HTML

```html
<p id="message" class="note">Hello!</p>
```

CSS

```css
p       { color: blue; }      /* Specificity: 1 */
.note   { color: red; }       /* Specificity: 10 */
#message { color: green; }    /* Specificity: 100 */
```

The final color will be **green**, because `#message` has the highest specificity.

The Cascade

If two rules have **equal specificity**, the **last one defined** wins.

Hello!

HTML

```html
<p id="message" class="note">Hello!</p>
```

CSS

```css
p { color: blue; }
p { color: red; }  /* This wins */
```

The `!important` Declaration

Overrides all specificity rules:

Hello!

HTML

```html
<p id="message" class="note">Hello!</p>
```

CSS

```css
p       { color: blue !important; }   /* Specificity: 1 */
.note   { color: red; }               /* Specificity: 10 */
#message { color: green; }            /* Specificity: 100 */
```

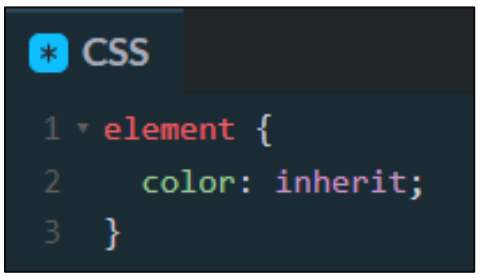
Use sparingly—it can make debugging harder and break expected behavior.

Inheritance in CSS

Some properties **naturally inherit** from parent elements:

Inherited by default	Not inherited
`color`	`margin`
`font-family`	`padding`
`line-height`	`border`

To force inheritance:



Or to prevent it:

Best Practices

- Use **classes** for styling instead of IDs
- Minimize the use of `!important`

- Organize styles from **generic → specific**
- Rely on **inheritance** where practical to reduce repetition

3.3 Typography and Color Styling

Typography plays a key role in making web content **readable**, **accessible**, and **visually consistent**. CSS gives full control over fonts, text size, spacing, alignment, and color.

Font Families

Set the typeface using the `font-family` property:

- Always provide **fallback fonts**
- Use **generic families**: `serif`, `sans-serif`, `monospace`, `cursive`, `fantasy`

You can also import fonts from services like Google Fonts:

Font Sizes and Units

Common units:

Unit	Description
`px`	Fixed pixel size
`em`	Relative to parent font
`rem`	Relative to root (`html`) font size
`%`	Percentage of parent

Use `rem` for scalability across screen sizes.

Text Styles

Property	Function
`font-weight`	Boldness (`normal`, `bold`, `600`)
`font-style`	Italic, normal
`text-transform`	Uppercase, lowercase, capitalize
`text-align`	Left, right, center, justify
`line-height`	Space between lines
`letter-spacing`	Space between characters

Example:

Text Decoration and Shadow

- `text-decoration`: `underline`, `none`, `line-through`
- `text-shadow`: Adds visual depth to text

Color Styling

Set color using:

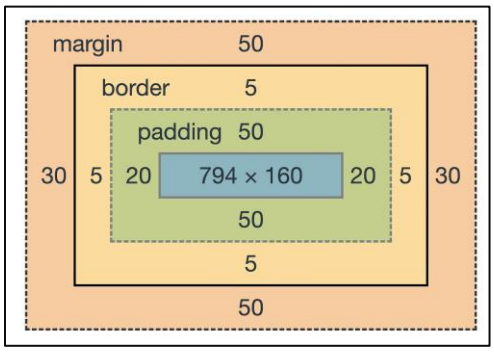
- Named colors: `red`, `navy`, `gray`
- Hex values: `#ff5733`
- RGB: `rgb(255, 87, 51)`
- HSL: `hsl(14, 100%, 60%)`

Use good contrast for readability and accessibility.

3.4 The Box Model Explained

Every HTML element is rendered as a **box**, and CSS allows control over its dimensions, spacing, and borders. Understanding the **box model** is fundamental to layout and spacing in web design.

Parts of the Box Model



1. **Content**: The actual content (text, image, etc.)
2. **Padding**: Space between content and border

3. **Border**: Visible line around the element
4. **Margin**: Space between this element and others

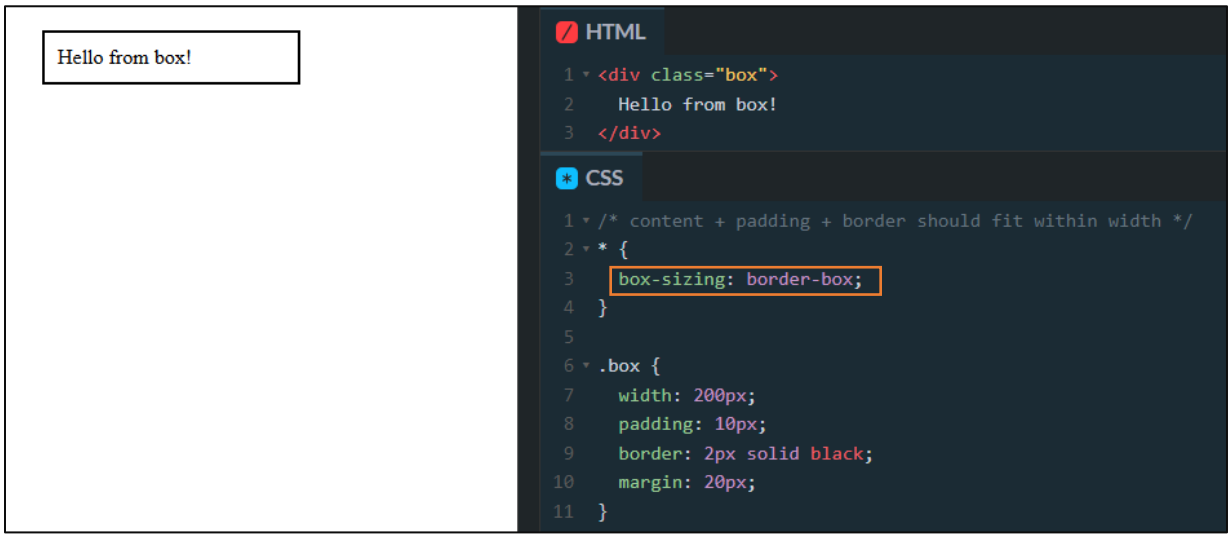
Controlling Box Dimensions

- `margin`: Outside spacing
- `border`: Outline thickness and style

Box-Sizing Property

By default, total width = **content + padding + border**

To make the element's `width` include everything, use:



This makes layouts more predictable and easier to manage.

Shorthand Properties

```css
element {
  margin: 10px 20px;      /* top/bottom 10px, left/right 20px */
  padding: 5px;           /* all sides 5px */
  border: 1px solid gray; /* width, style, color */
}
```

Margin Collapse

Vertical margins between block elements **may collapse** into a single margin. Horizontal margins do **not** collapse.

Example:

First Paragraph

Second Paragraph

```html
<p style="margin-bottom: 20px;">First Paragraph</p>
<p style="margin-top: 30px;">Second Paragraph</p>
```

Result: Combined vertical margin = 30px, not 50px.

Mastering the box model is key to precise layout control and spacing across your page.

3.5 Display Types and Positioning

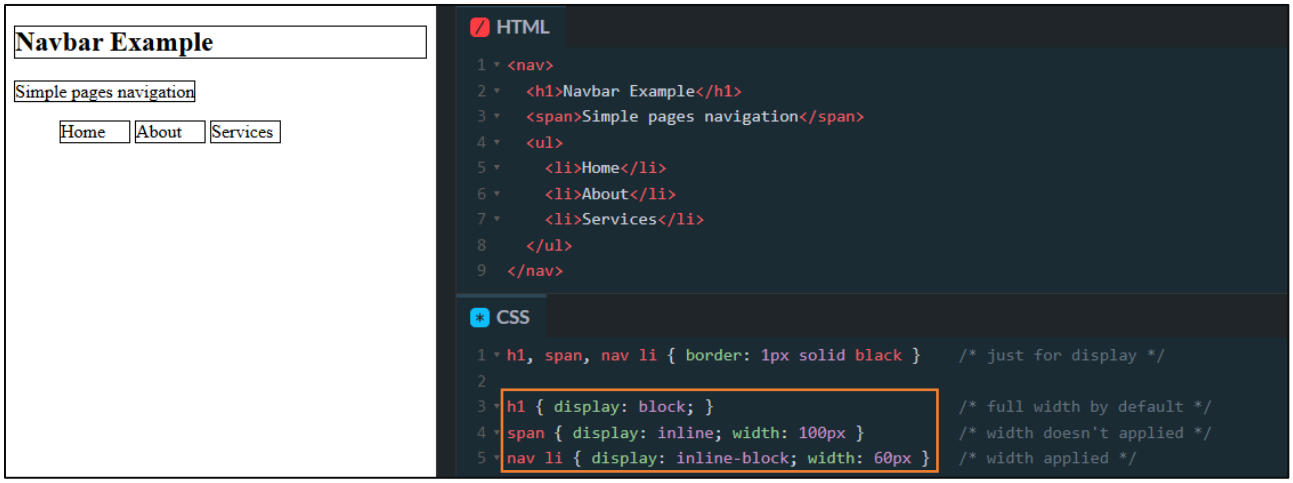
To control how elements flow and stack on a page, you must understand the **display** and **position** properties. These define how elements behave in the layout and how they interact with other content.

Display Property

The `display` property determines how an element is rendered in the document flow.

Value	Description
`block`	Starts on a new line, takes full width
`inline`	Flows with text, cannot have width/height
`inline-block`	Like `inline`, but supports width and height
`none`	Removes element from layout

Examples:



Visibility vs. Display

- `display: none;` removes the element from the flow
- `visibility: hidden;` hides it visually but leaves the space it occupied

Position Property

Controls how an element is **placed** on the page.

Value	Behavior
`static`	Default (in normal flow)
`relative`	Offset from its normal position
`absolute`	Positioned relative to nearest non-static ancestor
`fixed`	Positioned relative to the viewport
`sticky`	Scrolls until a point, then sticks in place

Example:

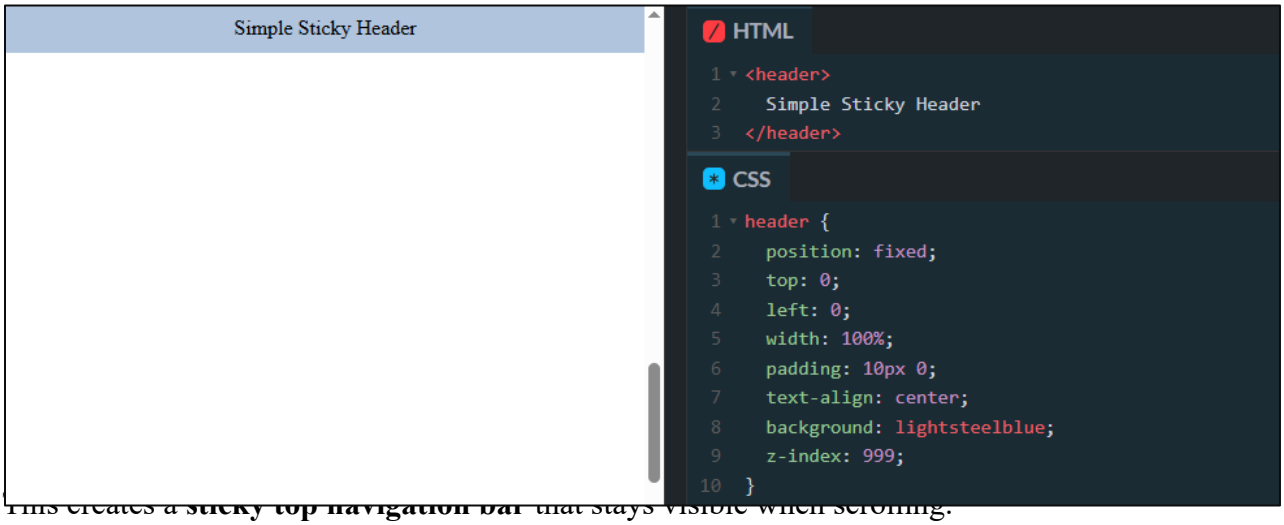
Z-Index

Used with `positioned` elements to control **stacking order**.

Higher `z-index` means it appears **on top** of lower values.

Combining Position and Display

Example layout snippet:



This creates a **sticky top navigation bar** that stays visible when scrolling.

Understanding `display` and `position` allows you to:

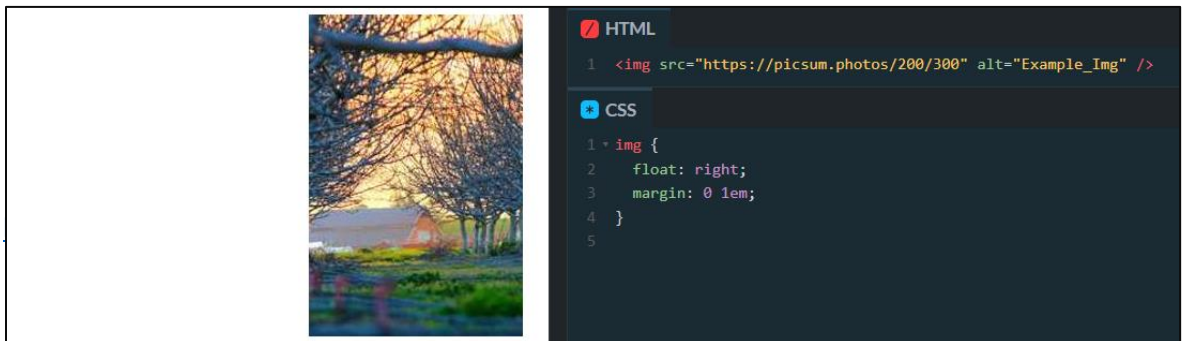
- Build flexible layouts
- Overlay content (modals, dropdowns)
- Handle responsiveness effectively

3.6 Float and Clear

Before modern layout systems like Flexbox and Grid, web developers relied heavily on the **float** property to build multi-column layouts. Though no longer the go-to technique, `float` is still useful in some cases, especially for **wrapping content around images**.

Using Float

The `float` property removes an element from normal flow and positions it to the **left or right** of its container.



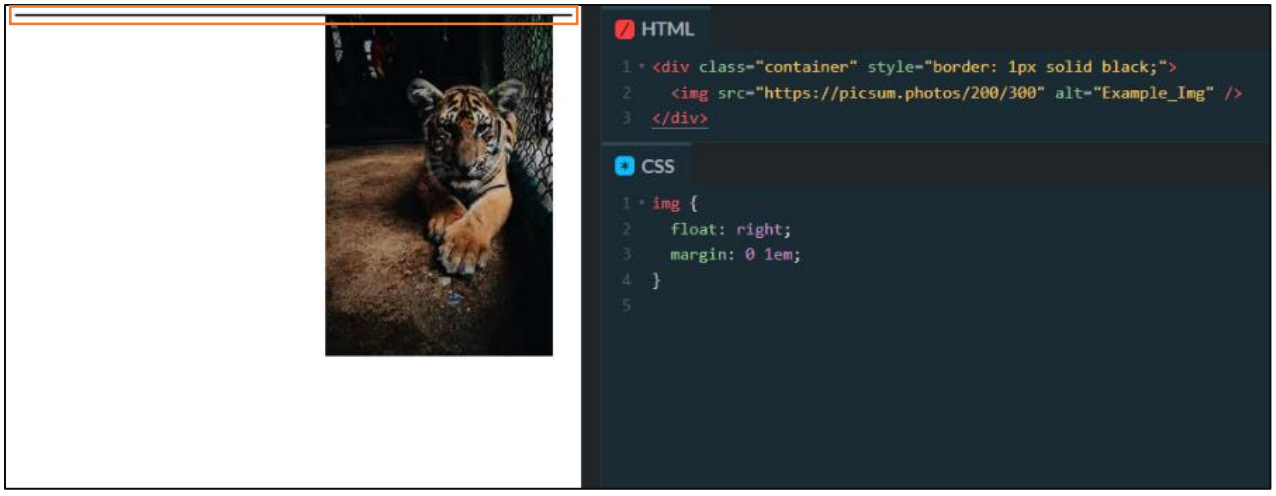
In this example, the image will float to the right, and surrounding text will wrap around it.

Common Float Use Cases

- Wrapping text around an image
- Building basic multi-column layouts (legacy)
- Aligning content side-by-side in older designs

The Float Problem

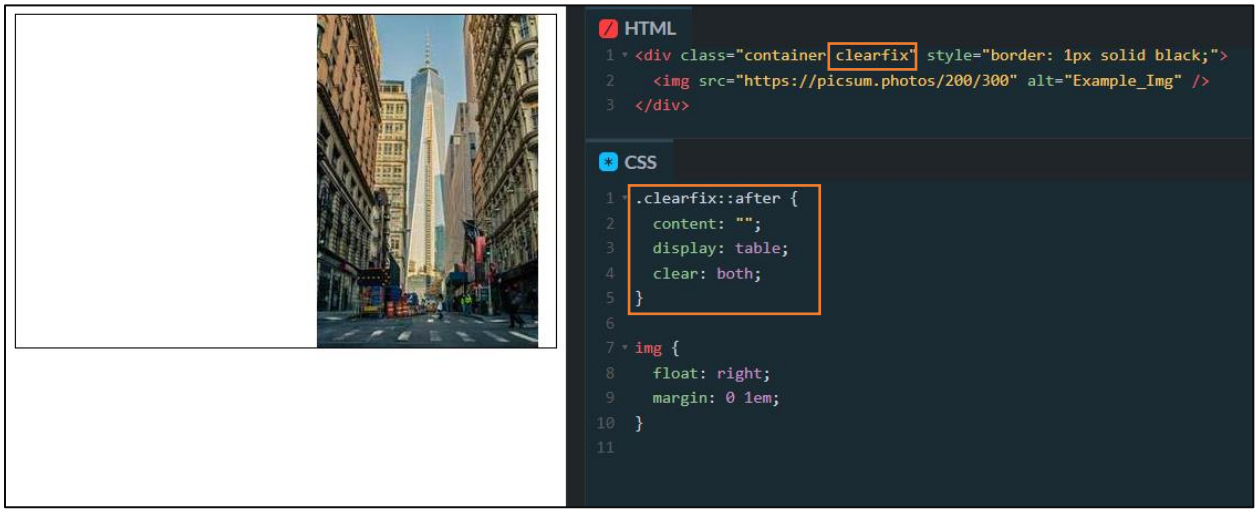
When an element is floated, its parent container may **collapse in height** because floated children are removed from the flow.



The `.container` may appear to have no height unless cleared.

Clearing Floats

To fix layout collapse, apply a **clearfix** to the container:



```html
<div class="container clearfix" style="border: 1px solid black;">
  <img src="https://picsum.photos/200/300" alt="Example_Img" />
</div>
```

```css
.clearfix::after {
  content: "";
  display: table;
  clear: both;
}

img {
  float: right;
  margin: 0 1em;
}
```

The `clear` Property

Stops an element from appearing next to floated elements.


```html
<div class="container clearfix" style="border: 1px solid black;">
  <img src="https://picsum.photos/200/300" alt="Example_Img" />
  <h1>Img Title</h1>
</div>
```

```css
.clearfix::after {
  content: "";
  display: table;
  clear: both;
}

img {
  float: right;
  margin: 0 1em;
}

h1 {
  clear: both;
}
```

When to Use Float Today

- **Good for**: Simple image wrapping, side labels
- **Avoid for**: General page layout (use Flexbox or Grid instead)

Floats helped build the early web, but they've largely been replaced by more powerful layout systems.

Chapter 4 – Responsive Design and Modern CSS Techniques

4.1 Mobile-First and Media Queries

Modern websites must adapt to many screen sizes—from phones to desktops. **Responsive Design** ensures your layout adjusts gracefully based on the device. This is achieved using **media queries** and a **mobile-first** approach.

What is Mobile-First?

A **mobile-first** design strategy means:

- You design for **small screens first** (mobile)
- Then use media queries to enhance the layout for **larger screens**

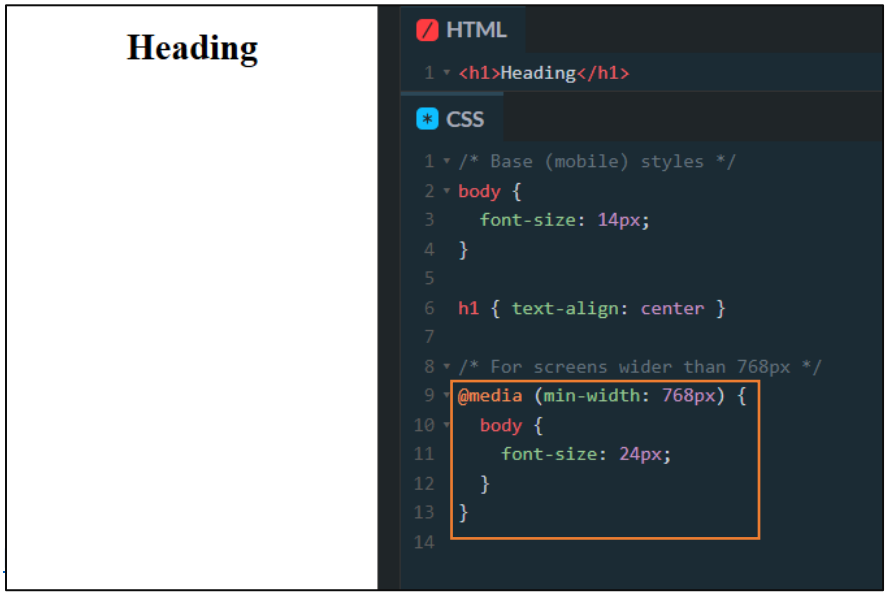
Benefits:

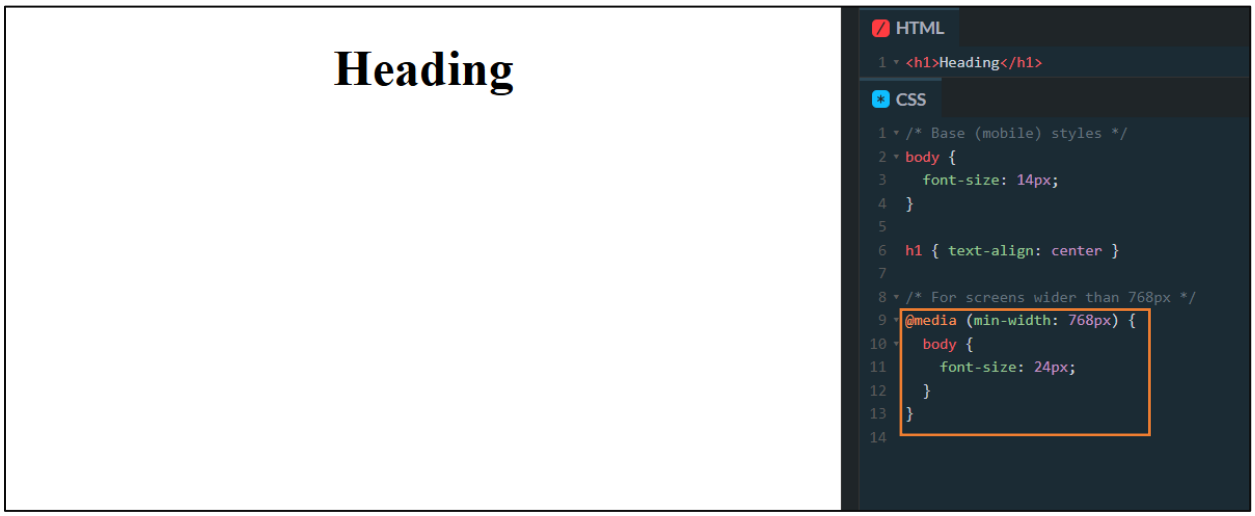
- Faster load times on mobile
- Better accessibility
- Progressive enhancement

What are Media Queries?

Media queries apply CSS rules conditionally, based on the **screen's size**, **orientation**, **resolution**, or other properties.

Example:





Media Query Syntax

```css
@media (media-type) and (condition) {
  /* CSS rules */
}
```

Common conditions:

Condition	Use Case
`min-width`	Style if screen is **wider**
`max-width`	Style if screen is **narrower**
`orientation: portrait`	Phones in vertical mode
`resolution: 300dpi`	High-DPI displays

Breakpoints

Breakpoints are screen widths where layout changes. Common examples:

```css
/* Tablet and up */
@media (min-width: 768px) { ... }

/* Desktop and up */
@media (min-width: 1024px) { ... }
```

Use breakpoints based on **content needs**, not just device types.

Best Practices

- Start with a **flexible base** layout
- Avoid hard-coded widths (use `%`, `vw`, `flex`)
- Use media queries to **enhance**, not duplicate
- Keep CSS organized (e.g. mobile styles first, then queries)

Responsive design isn't optional anymore—it's expected. Media queries give you control over how your site behaves across all devices.

4.2 Flexbox: Concepts and Practical Use

Flexbox (Flexible Box Layout) is a modern CSS layout system designed to arrange items **in one dimension**—either as rows or columns—with automatic resizing, spacing, and alignment.

Why Use Flexbox?

- Align items horizontally or vertically
- Distribute space between elements
- Easily create fluid, responsive layouts
- Replace floats, `inline-block`, and complex margin hacks

Basic Flex Container Setup

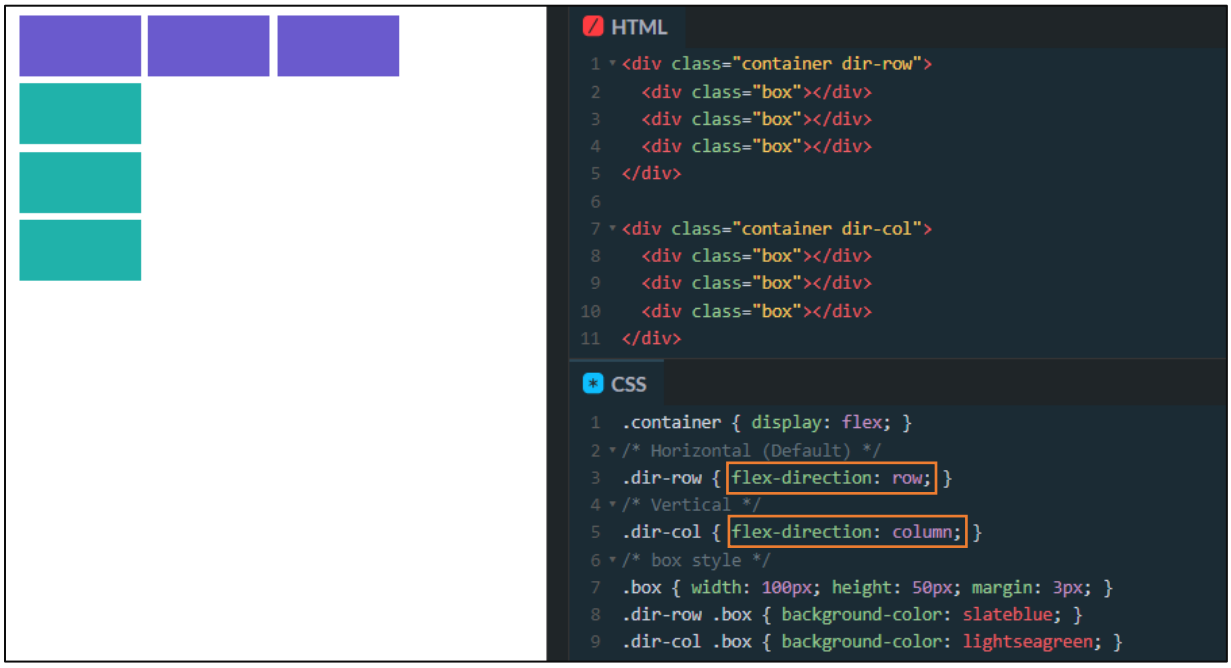
```css
.container {
  display: flex;
}
```

All direct children of `.container` become **flex items**.

Main Concepts

Term	Definition
Main axis	Direction of content flow (`row` by default)
Cross axis	Perpendicular to main axis (usually vertical)
Flex container	Parent element with `display: flex`
Flex item	Direct child of the flex container

Flex Direction



HTML

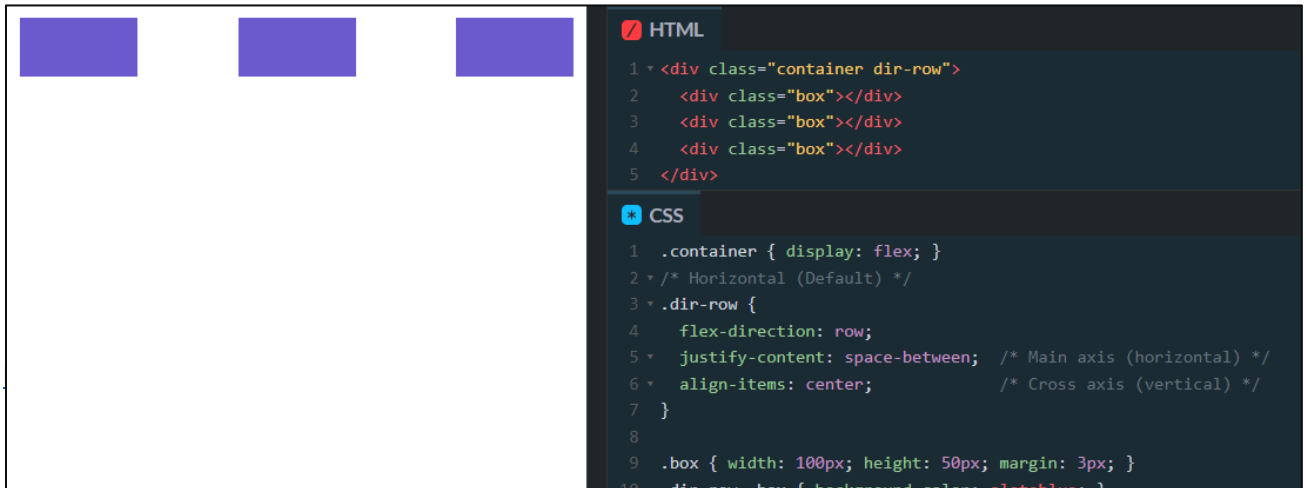
```html
<div class="container dir-row">
  <div class="box"></div>
  <div class="box"></div>
  <div class="box"></div>
</div>

<div class="container dir-col">
  <div class="box"></div>
  <div class="box"></div>
  <div class="box"></div>
</div>
```

CSS

```css
.container { display: flex; }
/* Horizontal (Default) */
.dir-row { flex-direction: row; }
/* Vertical */
.dir-col { flex-direction: column; }
/* box style */
.box { width: 100px; height: 50px; margin: 3px; }
.dir-row .box { background-color: slateblue; }
.dir-col .box { background-color: lightseagreen; }
```

Aligning Items



HTML

```html
<div class="container dir-row">
  <div class="box"></div>
  <div class="box"></div>
  <div class="box"></div>
</div>
```

CSS

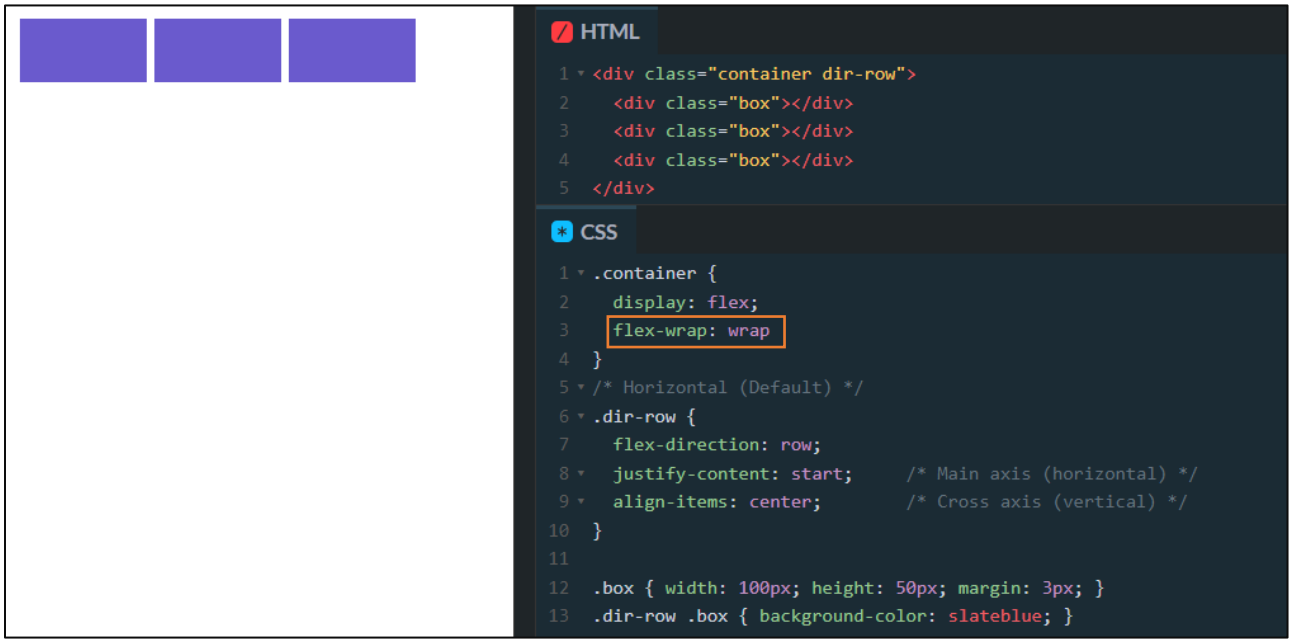
```css
.container { display: flex; }
/* Horizontal (Default) */
.dir-row {
  flex-direction: row;
  justify-content: space-between;  /* Main axis (horizontal) */
  align-items: center;             /* Cross axis (vertical) */
}

.box { width: 100px; height: 50px; margin: 3px; }
```

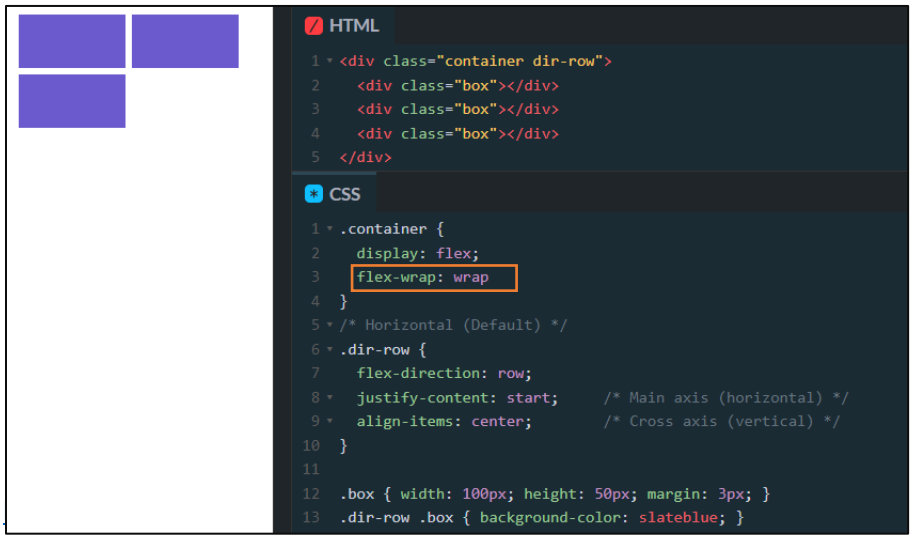
Common values:

- `justify-content`: `flex-start`, `center`, `space-between`, `space-around`, `space-evenly`
- `align-items`: `stretch`, `center`, `flex-start`, `flex-end`

Wrapping Items



Allows flex items to wrap to the next line if space runs out.

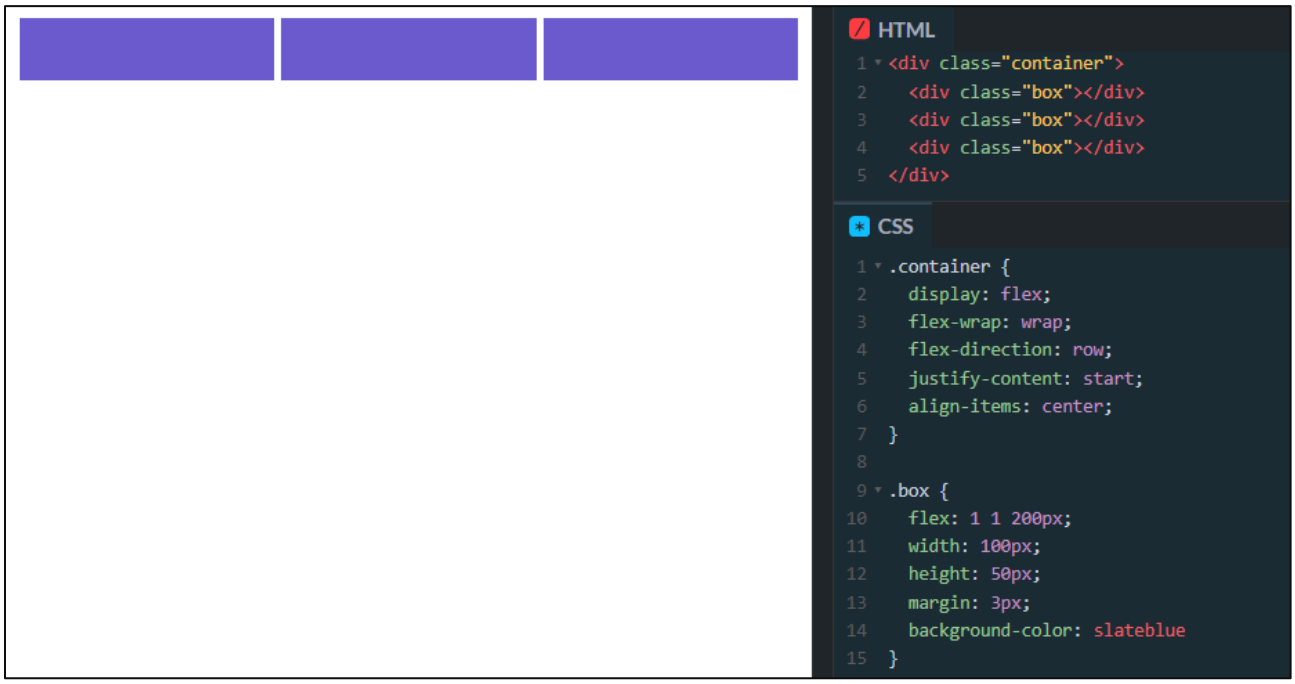


Controlling Flex Items

```css
.box {
  flex-grow: 1;
  flex-shrink: 1;
  flex-basis: 200px;
}
```

- `flex-grow`: how much it can expand
- `flex-shrink`: how much it can shrink
- `flex-basis`: initial size before space distribution
- `flex`: shorthand for above three

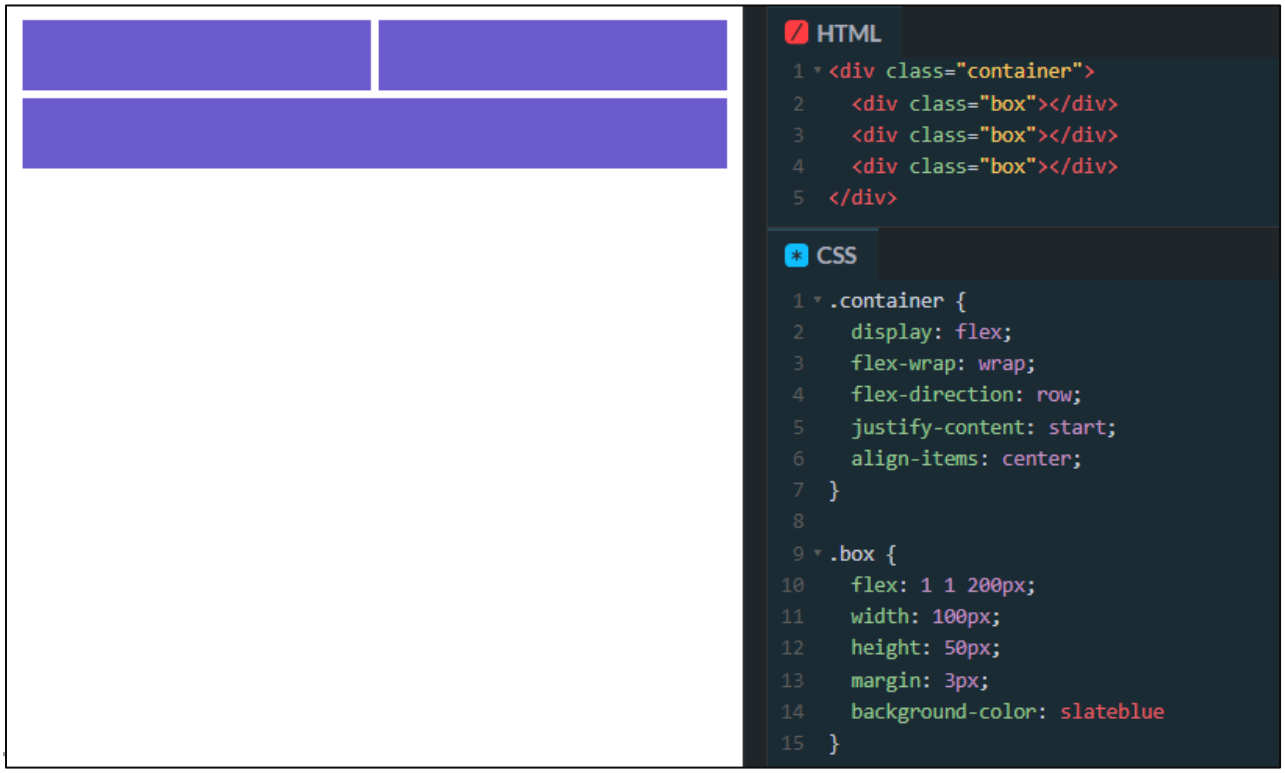
Real-World Example



```html
<div class="container">
  <div class="box"></div>
  <div class="box"></div>
  <div class="box"></div>
</div>
```

```css
.container {
  display: flex;
  flex-wrap: wrap;
  flex-direction: row;
  justify-content: start;
  align-items: center;
}

.box {
  flex: 1 1 200px;
  width: 100px;
  height: 50px;
  margin: 3px;
  background-color: slateblue
}
```



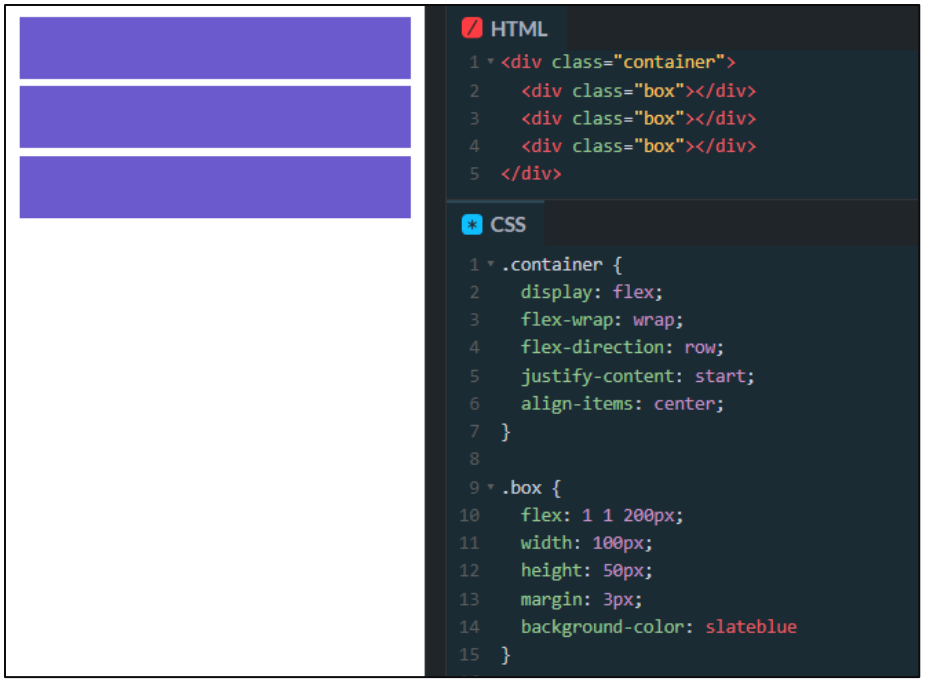
HTML

```html
<div class="container">
  <div class="box"></div>
  <div class="box"></div>
  <div class="box"></div>
</div>
```

CSS

```css
.container {
  display: flex;
  flex-wrap: wrap;
  flex-direction: row;
  justify-content: start;
  align-items: center;
}

.box {
  flex: 1 1 200px;
  width: 100px;
  height: 50px;
  margin: 3px;
  background-color: slateblue
}
```



HTML

```html
<div class="container">
  <div class="box"></div>
  <div class="box"></div>
  <div class="box"></div>
</div>
```

CSS

```css
.container {
  display: flex;
  flex-wrap: wrap;
  flex-direction: row;
  justify-content: start;
  align-items: center;
}

.box {
  flex: 1 1 200px;
  width: 100px;
  height: 50px;
  margin: 3px;
  background-color: slateblue
}
```

Flexbox is ideal for **menus, toolbars, cards, and centering elements**—especially when building responsive UIs.

4.3 Grid Layout: 2D Layout Made Easy

CSS Grid is a powerful layout system designed for creating **two-dimensional layouts**—arranging elements in rows and columns simultaneously. It complements Flexbox (which is 1D) by handling complex page structures more efficiently.

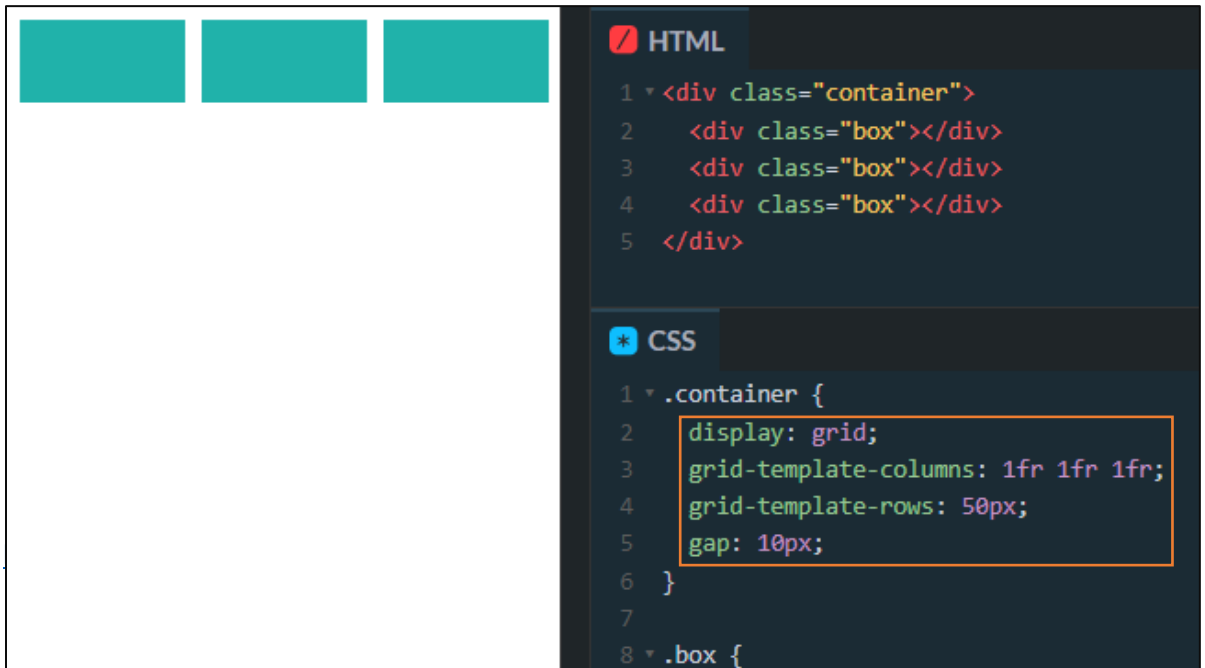
Why Use Grid?

- Create **complex page layouts** with minimal code
- Control both **rows and columns**
- Place items anywhere within the layout
- Build responsive templates and UI sections

This creates a grid with **3 equal-width columns**.

Creating a Grid Container

Use `display: grid;` to create a grid container, the use `grid-template-columns` and `grid-template-rows` to define the number and size of grid tracks:



Property	Description	Example
`grid-template-columns`	Defines the columns in the grid (left to right)	`1fr 2fr 1fr` = 3 columns: 25%, 50%, 25%
`grid-template-rows`	Defines the row heights (top to bottom)	`auto 100px` = 1st row auto height, 2nd is fixed

- `fr` = fraction of available space
- `auto` = adjusts to content
- `px`, `%` = fixed units

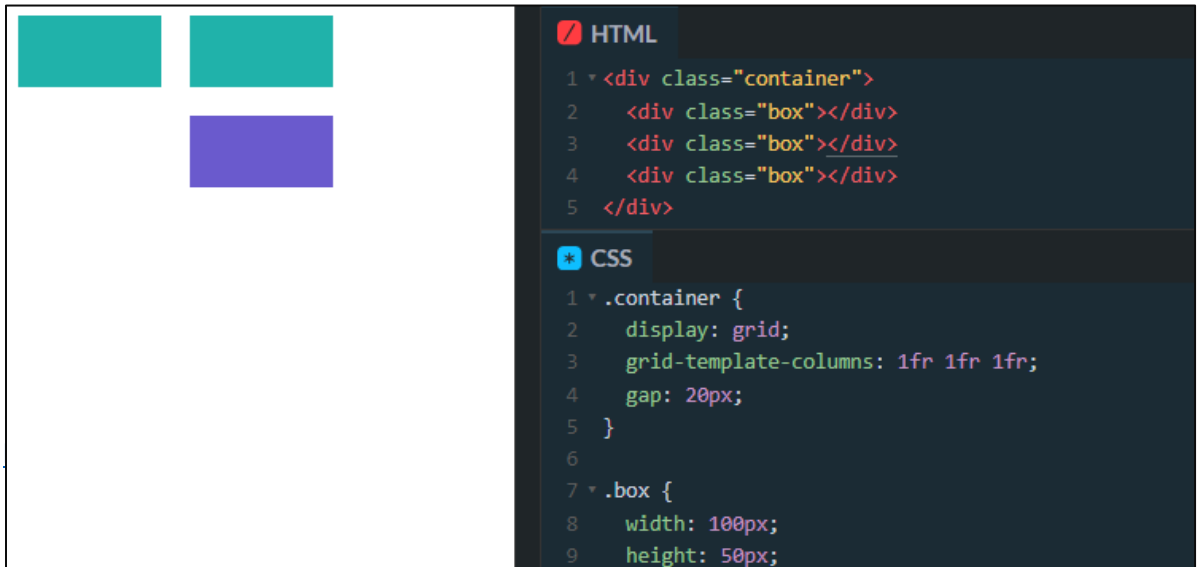
Grid Gaps

When using `gap`, Add space between rows and columns:

Property	Description
`gap`	Shorthand for row & column gaps
`row-gap`	Vertical spacing between rows
`column-gap`	Horizontal spacing between columns

Placing Items

Each direct child of a grid container becomes a **grid item**. You can place them in specific rows/columns:



Property	Description
`grid-column`	Starts at column 2, ends before column 4 (spans 2)
`grid-row`	Starts at row 1, ends before row 2 (spans 1)

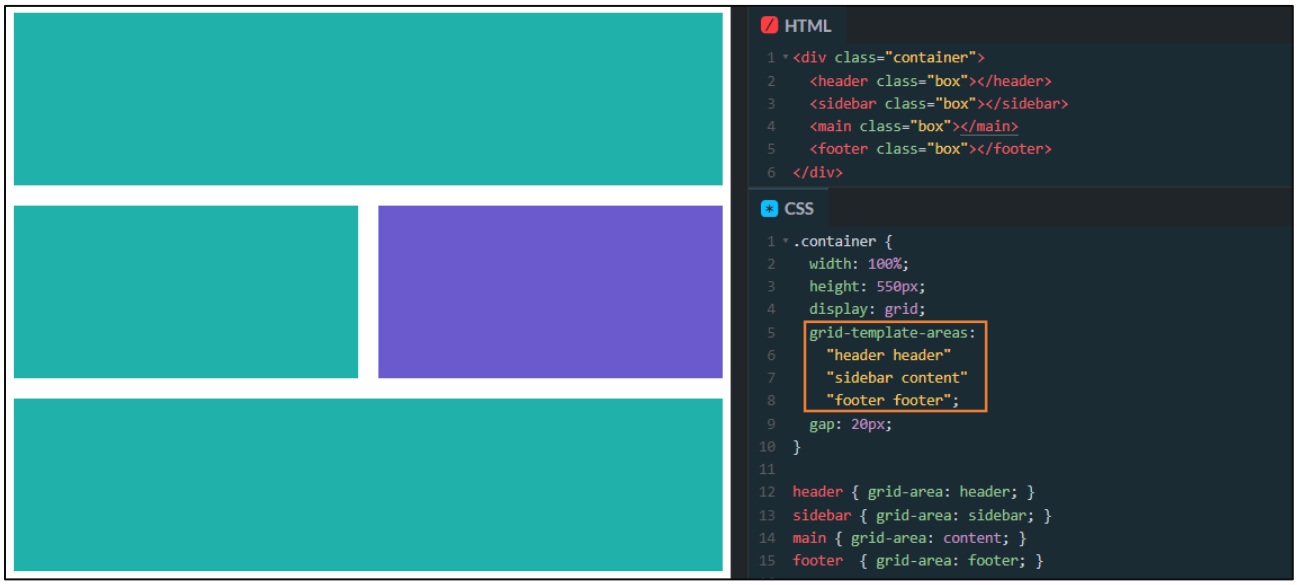
Format: `start / end`

You can also **name lines** or use **span**:

```
css
CopyEdit
grid-column: 1 / span 2; /* span across 2 columns starting from 1 */
```

Grid Areas (Named Layouts)

Define named areas to make layout more readable, then assign areas to items:



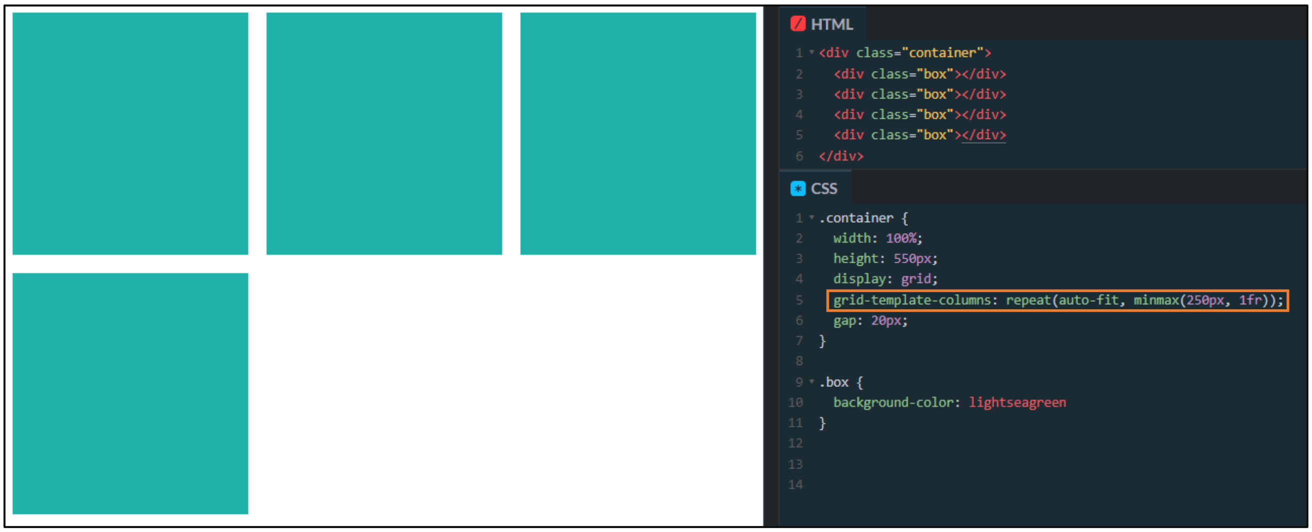
Term	Meaning
`grid-area`	Refers to named layout zone

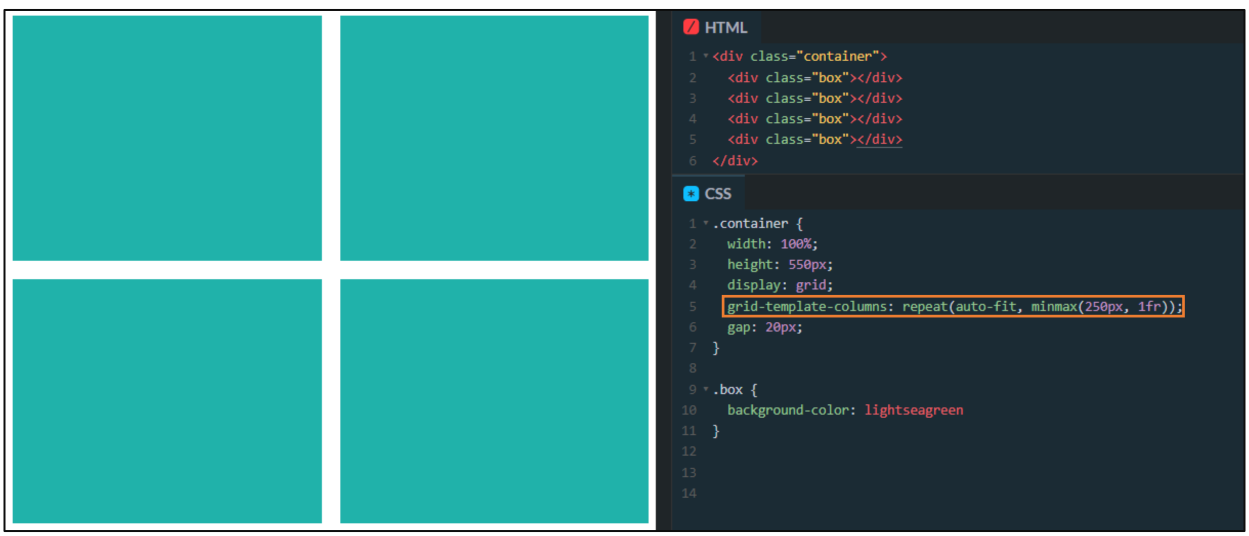
`template-areas`	Declares overall grid map

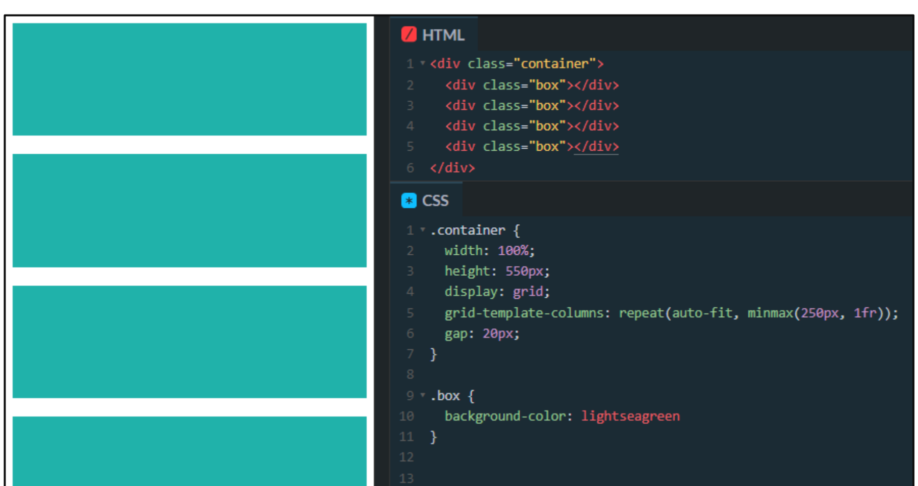
This is **very useful** for responsive page templates.

---

6. Responsive Grid with Auto-Fit







Function	Description
`repeat(n, val)`	Repeats a column n times
`auto-fit`	Fills available space by adjusting column count automatically
`minmax(a, b)`	Each column must be at least `a`, and can grow up to `b`

This layout will:

- Create as many 250px-wide columns as can fit
- Make them responsive by stretching (`1fr`)

Summary

Feature	Best For
`grid-template-columns/rows`	Defining fixed or fluid layout tracks
`grid-column/row`	Precise placement of items
`grid-template-areas`	Semantic and readable layouts
`auto-fit/minmax`	Fully responsive grid systems

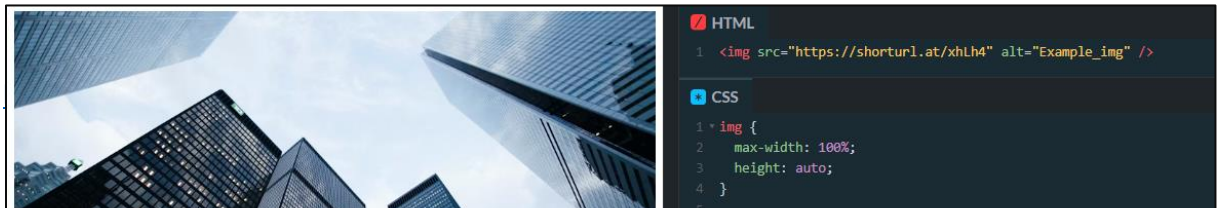
4.4 Responsive Images and Viewport Units

To build fully responsive websites, media like **images** and units like `vw` and `vh` must scale with the screen. This section covers how to handle responsive imagery and use **viewport-based sizing**.

Responsive Images

Basic Technique

Use `max-width` and `height: auto` to make images scale down but not stretch:

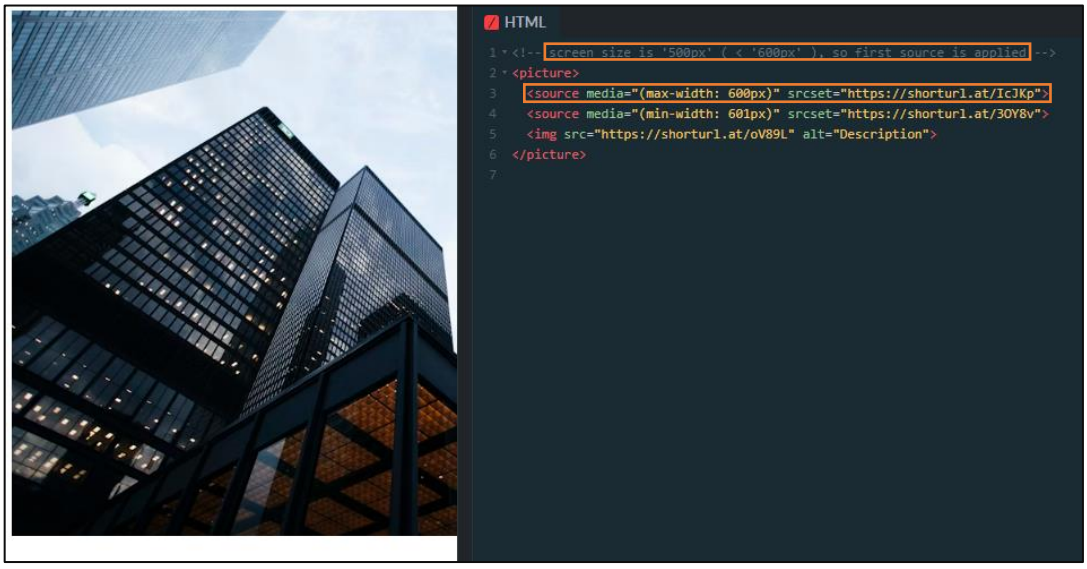


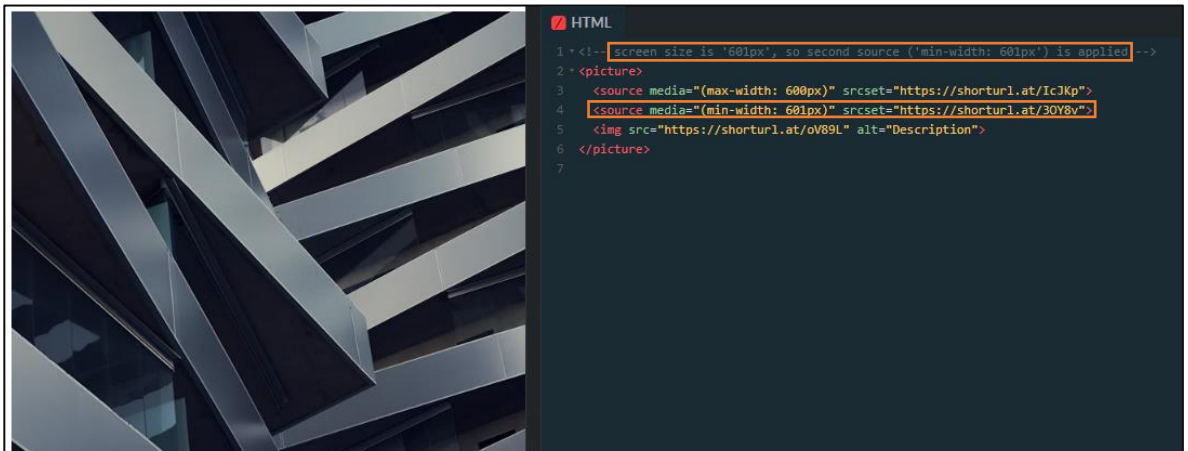
Property	Purpose
`max-width`	Prevents image from exceeding container
`height: auto`	Maintains aspect ratio

This is the **most common method** for responsive images.

`<picture>` Element

Serves **different images** based on screen size or format:





- The browser picks the best matching image
- Use for high-performance, responsive image loading

`srcset` and `sizes`

Use these attributes to provide **multiple resolutions** of the same image:

```HTML
<img
  src="img-default.jpg"
  srcset="img-small.jpg 600w, img-large.jpg 1200w"
  sizes="(max-width: 600px) 100vw, 600px"
  alt="Description">
```

- `srcset`: list of image files with width descriptors
- `sizes`: tells the browser how wide the image will display
- Best for **retina screens** and varying screen widths

Viewport Units

What Are Viewport Units?

Unit	Meaning
`vw`	1% of the **viewport width**
`vh`	1% of the **viewport height**
`vmin`	1% of the smaller dimension
`vmax`	1% of the larger dimension

Example:

```CSS
#hero {
  height: 100vh;   /* full screen height */
  font-size: 4vw;  /* scales with screen width */
}
```

When to Use Viewport Units

- Full-screen sections (`100vh`, `100vw`)
- Large responsive headings
- Mobile-first sections that fill the screen
- Custom breakpoints (instead of px)

Tips

- Combine `vw`/`vh` with **media queries** for best results
- Always test **text in vw**—can be too small on mobile
- Don't forget **`object-fit: cover`** for background-like images:

Responsive images save bandwidth and improve loading speed, while viewport units help build scalable, flexible designs.