Lesson 1: Browser Developer Tools & Supercharging CSS

Codepen Collection for Module 4: https://codepen.io/collection/YyONeO

What are Browser DevTools?

Browser Developer Tools (often referred to as DevTools) are a set of tools built into modern web browsers that allow developers to inspect, debug, and profile web pages. They provide insights into how pages render and perform, and offer a variety of utilities to optimize and diagnose any issues.

Resources:

- Introduction to Web Developer Tools - MDN

How to Open DevTools

- Google Chrome:
  - Right-click on any element and choose "Inspect" from the context menu.
  - Use the keyboard shortcut `Ctrl + Shift + I` (Windows/Linux) or `Cmd + Option + I` (Mac).
- Firefox:
  - Right-click on any element and select "Inspect Element" from the context menu.
  - Use the keyboard shortcut `Ctrl + Shift + I` (Windows/Linux) or `Cmd + Option + I` (Mac).
- Safari:
  - First, enable the "Develop" menu from Preferences > Advanced. Then, from the "Develop" menu, select "Show Web Inspector".
  - Use the keyboard shortcut `Cmd + Option + I`.

Chrome DevTools Keyboard Shortcuts

- Open/Close DevTools: `Ctrl + Shift + I` or `F12`
- Console Panel: `Ctrl + Shift + J`
- Elements Panel: `Ctrl + Shift + C`
- Network Panel: `Ctrl + Shift + E`

(These are just a few. There are many more shortcuts available.)

Resources:

- Chrome DevTools Shortcuts
- HTML Validator: https://validator.w3.org/
- CSS Validator: https://jigsaw.w3.org/css-validator/

The Different DevTools Components

- CONSOLE PANEL: Displays logged messages, run JavaScript, and see errors or warnings.
- NETWORK PANEL: Shows all network requests made by a page, their status, type, initiator, size, and more.
- ELEMENTS PANEL: Inspect and modify the HTML and CSS of a page.
    - Styles Pane: View and edit CSS styles. It shows the styles applied to the selected DOM node, including styles inherited from its ancestors. You can also add, delete, and modify CSS declarations here.
    - Computed Pane: Shows the computed CSS values for the selected DOM node. It helps understand how styles are resolved and applied to an element.
    - Layout Pane: Provides tools to inspect and debug layout issues. It visualizes the box model for the selected element, allowing developers to see padding, borders, and margins.
    - Event Listeners Pane: Lists all the event listeners attached to the selected DOM node. This is useful to understand how elements respond to various events like `click`, `hover`, etc.
    - DOM Breakpoints Pane: Allows developers to set breakpoints on specific DOM mutations. For example, you can set a breakpoint when an element's attributes change, or when a node is removed.
    - Properties Pane: Displays all JavaScript properties of the selected DOM node, including its prototype chain. It's useful for exploring the JavaScript representation of a DOM node.
    - Accessibility Pane: Provides insights into how accessible an element is to screen readers and other assistive technologies. It shows ARIA roles, contrast ratio, and other relevant information.
- Device Mode: Simulate various devices, screen sizes, and resolutions.

Pseudo-classes & Pseudo-elements: Deep Dive

What are Pseudo-classes and Pseudo-elements?

- Pseudo-classes: They are used to define special states of an element that can't be targeted using simple selectors. For instance, `:hover` can be used to style an element when a user's pointer is over it. They are prefixed with a single colon `:`.
- Pseudo-elements: They allow you to style certain parts of a document that aren't represented by any other elements. For example, `::before` and `::after` allow you to insert content before or after an element's content. They are prefixed with a double colon `::`.

Difference Between Pseudo-classes and Pseudo-elements:

- Syntax: Pseudo-classes use a single colon (e.g., `:hover`), while pseudo-elements use a double colon (e.g., `::before`).
- Target: Pseudo-classes target specific states or positions of elements, while pseudo-elements target specific parts of an element.
- Use Cases: Pseudo-classes are often used for user interaction states like hover or focus, whereas pseudo-elements are used to style or add content around or inside an element.

When/Why Do We Use Them?

- Pseudo-classes:
    - User Interaction: To change the appearance of links on hover (`a:hover`), or to style an input field when it's focused (`input:focus`).
    - Structural: To style specific children of a parent, like the first child (`:first-child`), or every odd child (`:nth-child(odd)`).
    - Form Feedback: To provide visual feedback on form elements, like styling invalid fields (`:invalid`).
- Pseudo-elements:
    - Styling Decorations: To add decorative elements without altering the HTML, like adding icons or indicators using `::before` and `::after`.
    - Highlighting Content: To style the first line or first letter of a text block differently using `::first-line` and `::first-letter`.
    - Custom Markers: For custom list markers using `::marker`.

Benefits:

- Cleaner HTML: They can reduce the need for additional HTML elements, keeping your markup cleaner.
- Enhanced Styling: They offer advanced styling capabilities, allowing for interactive and dynamic effects based solely on CSS.

- Semantics: Using pseudo-elements can keep the HTML semantically correct by avoiding the addition of non-semantic divs and spans.

Common Pseudo-classes:

- `:hover`: Style an element when a user hovers over it.
- `:focus`: Style an element when it gains focus (e.g., an input field).
- `:first-child`: Selects the first child element of its parent.
- `:last-child`: Selects the last child element of its parent.
- `:nth-child(n)`: Selects the nth child element of its parent.
- `:not(selector)`: Selects every element that is not the specified element/selector.

Common Pseudo-elements:

- `::before`: Insert content before the content of an element.
- `::after`: Insert content after the content of an element.
- `::first-letter`: Selects the first letter of a block-level element.
- `::first-line`: Selects the first line of a block-level element.
- `::marker`: Style list item markers.
- `::selection`: Style the portion of an element that's selected by a user.

Resources:

- Pseudo-classes - MDN
- Pseudo-elements - MDN
- https://www.w3schools.com/css/css_pseudo_classes.asp
- https://www.w3schools.com/css/css_pseudo_elements.asp

Sass up your Styles: The Awesome CSS Preprocessor

Introduction to Sass: Sass (Syntactically Awesome Style Sheets) is a preprocessor scripting language that gets compiled into Cascading Style Sheets (CSS). It introduces a range of features not available in vanilla CSS, such as nested rules and mixins.

Difference between Sass and CSS:

- Sass offers features that don't exist in standard CSS such as nesting and mixins.
- Sass files have `.scss` (Sassy CSS) or `.sass` extensions and need to be compiled to `.css`.

How Does Sass Work?: Writing: You write your styles using Sass. This can be done using either the `.scss` (Sassy CSS, which is closer to regular CSS) or `.sass` syntax

(which omits braces and uses indentation to separate code blocks). Compiling: Sass code does not run in browsers directly. Instead, it must be compiled into standard CSS. This compilation can be done through command line tools, build tools, or applications specifically designed for this purpose. Output: Once compiled, you get a regular `.css` file, which you can link in your HTML file. This CSS file is what the browser reads and applies to your webpage.

Setting Up Sass:

There are several ways to install Sass in your system. There are many applications that will get you up and running with Sass in a few minutes for Mac, Windows, and Linux. You can read more about them here: https://sass-lang.com/install/

You can also check out this tutorial by Coder Coder: How I setup VS Code for a beginners front-end workflow

Sass in CodePen: You can use Sass directly in CodePen. When creating a new pen, you can select Sass (or its variant SCSS) as your preprocessor for the CSS panel. CodePen will automatically compile your Sass into CSS. You don't need any command line commands; it's all handled within the CodePen environment.

Features of Sass: Variables: Store reusable values like colors, font sizes, or spacing.

```
$primary-color: #3498db;
body {
  background-color: $primary-color;
}
```

Nesting: Nest CSS rules within each other.

```
nav {
  ul {
    list-style: none;
    li {
      display: inline-block;
    }
  }
}
```

Mixins: Write reusable styles.

```
@mixin transition($property: all) {
  transition: $property 0.3s ease-in-out;
}
.box {
```

```
  @include transition(opacity);
}
```

Functions: Similar to mixins but they return a value that can be used in a property.

```
@function set-text-color($bg-color) {
  @if (lightness($bg-color) > 50) {
    @return #000; // Return black for light backgrounds
  } @else {
    @return #fff; // Return white for dark backgrounds
  }
}
```

Resources:

- Sass Official Website
- Sass Mixin Documentation
- Sass Guide - MDN
- W3Schools Sass Tutorial
- FreeCodeCamp Sass Course
- CodePen Collection: Module 4

Lesson 2: Responsive Design

Introduction: Responsive design ensures that websites look and function well on a variety of devices and window sizes. As more users access the web on mobile devices, it's become crucial to design flexibly.

Philosophy of Responsive Design:

- Fluid Grids: Elements will scale relative to their containing elements.
- Flexible Images: Images behave as fluid grids do.
- Media Queries: Apply styles based on the device characteristics, like its width.

Basics of Responsive Design:

- Adaptive vs. Responsive:
    - Adaptive: Targets specific device sizes.
    - Responsive: Targets ranges of sizes.
- Mobile-First vs. Desktop-First: Starting with mobile styles and then expanding to desktop, or vice-versa.

The Viewport & Other Considerations:

The Viewport:

- The viewport is the currently visible part of a webpage on a device's screen.
- Responsiveness, at its core, revolves around how content adjusts to fit this viewport.

Meta Viewport Tag:

- This tag, `<meta name="viewport" content="width=device-width, initial-scale=1" />`, is crucial for responsive design.
- It instructs the browser to set the width of the page to follow the screen-width of the device and to start with a zoom level of 1.

- Most modern HTML templates (like the ones generated by Emmet abbreviations in editors like VS Code) include this tag by default.
- This tag ensures that your site will be displayed correctly on various devices.

HTML's Natural Responsiveness:

- By default, HTML tries to be responsive. It's CSS that can sometimes restrict this natural behavior.
- avoid setting fixed `height` and `width` in CSS. Such fixed dimensions can lead to issues:
    - Fixed widths can create horizontal scrolling.
    - Fixed heights can make content overflow its container.
- Instead, it's recommended to use properties like `min-height`, `min-width`, `max-width`, or even `padding` to maintain responsiveness.

Rule of Thumb Coding Practices for Responsiveness:

- Avoid using fixed widths; instead, use percentages or relative units.
- Use `max-width` to avoid oversized elements on larger screens.
- Always test your website on multiple devices.

Fixed and Relative Settings:

- Fixed Width: Using exact pixels like `width: 300px;`.
- Relative Width (like `max-width`):

```
.container {
    max-width: 1200px;
    margin: 0 auto;
- }
```

Relative Units Relative units in CSS provide a way to define dimensions as they relate to something else, rather than as absolute values. This makes designs more adaptable to various screen sizes, resolutions, and user preferences.

some of the most commonly used relative units:

`em`: - Relative to: The font-size of the element. - Usage: If the font-size of the current element is 16px, then `1em = 16px`. If you set the font-size of an element to `1.5em`, it would compute to `24px`. - Common Use Cases: Making padding or margins related to the font-size, scaling elements in relation to their font-size.

`rem` (Root `em`): - Relative to: The font-size of the root element (`<html>`), often set to 16px by default in browsers. - Usage: If the root font-size is 16px, then `1rem = 16px`. Changing the font-size of the root element will scale all values set in `rem` throughout the site. - Common Use Cases: Creating consistent spacing and sizing throughout a site, especially when wanting to maintain proportional relationships regardless of local font-size.

`vh` (Viewport Height): - Relative to: The height of the viewport. - Usage: `1vh` is equal to 1% of the viewport's height. - Common Use Cases: Creating full-screen sections, or elements that adapt to the height of the viewport.

`vw` (Viewport Width): - Relative to: The width of the viewport. - Usage: `1vw` is equal to 1% of the viewport's width. - Common Use Cases: Elements that adapt to the width of the viewport.

`vmin` and `vmax`: - Relative to: The viewport's dimensions. - Usage: - `1vmin` is equal to 1% of the viewport's smallest dimension (either width or height). - `1vmax` is equal to 1% of the viewport's largest dimension. - Common Use Cases: Ensuring elements scale proportionally within the viewport, especially when orientation changes.

`%` (Percentage): - Relative to: The parent element's corresponding dimension. - Usage: If a box is inside a container that has a width of 500px, setting the box's width to `50%` would make it 250px wide. - Common Use Cases: Creating fluid layouts, especially in grid or flexbox designs.

Media Queries:

- Media queries in CSS allow you to apply styles based on certain conditions, such as screen width, height, device type, and more.
- Basic Syntax:

```
@media screen and (max-width: 600px) {
    body {
        background-color: lightblue;
    }
    }
```

- This changes the background color to light blue when the screen size is 600px or less.
- Breakpoints: Specific values at which your site's layout or style changes to accommodate different screen sizes.
  Common Breakpoints:
  - Small devices (phones): `@media (max-width: 640px) { ... }`
  - Medium devices (tablets): `@media (max-width: 768px) { ... }`
  - Large devices (desktops): `@media (max-width: 1024px) { ... }`
  - X-Large devices (large desktops): `@media (max-width: 1200px) { ... }`
- Device Types:
  Using the `media` feature, you can target specific device types.
  - All Devices: `@media all { ... }`
  - Screen (most common): `@media screen { ... }`
  - Print: `@media print { ... }` (for print previews and printing pages)
  - Speech: `@media speech { ... }` (for screen readers)
- Width and Height:
  These are the most commonly used features in media queries.
  - Width:
    - max-width: `@media (max-width: 600px) { ... }` (styles for devices with a viewport width up to 600px)
    - min-width: `@media (min-width: 601px) { ... }` (styles for devices with a viewport width of 601px and above)
  - Height:
    - max-height: `@media (max-height: 400px) { ... }`
    - min-height: `@media (min-height: 401px) { ... }`
- Orientation:
  Target devices based on their orientation: portrait or landscape.
  - `@media (orientation: portrait) { ... }`
  - `@media (orientation: landscape) { ... }`

Review this documentation for the @media rule:
https://www.w3schools.com/cssref/css3_pr_mediaquery.php

Resources:

1. MDN Responsive Design Guide
2. CSS-Tricks Media Queries Guide
3. Responsive Web Design Patterns
4. CSS Units: https://www.w3schools.com/cssref/css_units.php

Lesson 3: CSS Animations

Introduction: CSS animations allow us to bring websites and web applications to life by smoothly transitioning between CSS property values. They consist of two main components: `transitions` and `animations`.

Transitions:

Transitions let you change property values smoothly over a given duration.

- Syntax:

```
.element {
    transition: property duration timing-function delay;
}
```

- Example:

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

- In this example, when hovering over `.box`, its `background-color` will change from blue to red over 0.5 seconds with an `ease-in-out` timing function.

Animations:

Animations are more complex and provide finer control over the animation sequence using `keyframes`.

- Keyframes: Define the sequence of frames during the animation.

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

```
        opacity: 1;
    }
}
```

- The above `fadeIn` animation changes the `opacity` from 0 to 1.
- Usage:

```
.element {
    animation: fadeIn 2s;
}
```

- This applies the `fadeIn` animation to the `.element`, and the animation will last for 2 seconds.

CSS Animation: The Building Blocks:

- Animation Name: Refers to the `keyframes` at-rule to use.
- Animation Duration: How long the animation will take.
- Animation Timing Function: Speed curve of the animation.
- Animation Delay: How long to wait before starting the animation.
- Animation Iteration Count: How many times to play the animation.
- Animation Direction: Direction of the animation, e.g., `normal`, `reverse`, `alternate`.
- Animation Fill Mode: Styles applied before or after the animation.
- Animation Play State: Whether the animation is running or paused.

Example of a Basic CSS Animation:

Let's create a simple fade-in animation:

CSS:

```
@keyframes fadeIn {
    from {
        opacity: 0;
    }
    to {
        opacity: 1;
    }
}

.element {
    animation: fadeIn 2s;
```

```
}
```

When applied, `.element` will fade in, gradually becoming fully opaque over 2 seconds.

What Do Keyframes Do?

`Keyframes` in CSS animations define a sequence of frames where styles are applied at specific times. Using `from` and `to` (or percentage values), we can dictate the starting, ending, or intermediate states of an animation.

Best Practices & Accessibility with Animations:

- Avoid Overuse: Too many animations can distract users.
- Performance: Some animations can be resource-intensive. Always test performance.
- Accessibility: Be considerate of users who might be sensitive to motion.

`prefers-reduced-motion`:

For users who have set their devices to reduce motion (often due to motion sensitivity), we can respect their settings using the `prefers-reduced-motion` media query.

```
@media (prefers-reduced-motion: reduce) {
    .element {
        animation: none !important;
        transition: none !important;
    }
}
```

This disables animations and transitions for users who prefer reduced motion.

Resources:

1. MDN CSS animations
2. prefers-reduced-motion: Guide
3. W3Schools Animations Tutorial

4. https://www.w3schools.com/css/css3_transitions.asp
5. https://www.w3schools.com/css/css3_animations.asp