MDN Curriculum

The essential skillset for new front-end developers

PDF generated: June 9, 2025

Last major curriculum review: February, 2024

MDN recommends Scrimba

We recommend The Frontend Developer Career Path from our learning partner, Scrimba. This course provides a fun, interactive learning experience that teaches all of the MDN Curriculum Core and more.

Check it out

Contents

Getting started modules

The subjects outlined in these modules are not web development topics, but they are useful for anyone wanting to learn front-end web development to have an understanding of. We don't consider learning these topics as required before moving on to the core modules, hence we haven't used the word "prerequisite". However, we believe students will have an easier time if they spend some time on these topics first.

1. Soft skills

This module provides recommendations of soft skills that students can aim to get better at while learning web development, and which constitute good traits to have when entering the industry. They will help immensely in developing the right attitudes for learning, researching, and collaborating, and increase the chances of getting hired.

1.1 Constant learning mindset

Students should get into the mindset of constant learning. The web is constantly evolving and technologies and trends are always changing, and they need to constantly update their knowledge to keep up.

- Get into the habit of regularly reading technical news, blogs, and browser release notes.
- Engage in reading tasks or small research projects semi-regularly.
- Set aside specific learning time to spend on acquiring new skills.
- Be curious.

Recommended news and information sites:

- MDN
- CSS Tricks
- Dev
- freeCodeCamp
- A List Apart
- Smashing Magazine
- CodeCademy

1.2 Open to embracing failure

A very common issue that causes students and new developers to shy away from experimentation and taking risks (for example when starting new projects or exploring new ideas) is fear of failure. Spend some time learning about the value that can be gleaned from making mistakes, and the lessons that can be learned and applied in the future in similar situations.

Here are some tips to improve this skill:

- Define a safe space/peer group where people are free to ask questions and failure will not be judged harshly.
- Look to your local community and try to find meetup groups with people who can either give you help and advice or are facing the same issues you are and can provide moral support or experiment together with you.
- (For educators) Set up the marking schemes for your assessments so that you can still get a reasonable number of marks even if you didn't get the correct result provided the process is well documented. Award extra marks for innovation.
- Run show 'n' tell or one-on-one sessions part-way through a project with peers and mentors to get feedback and insights into where you are going wrong and get advice on how to get back on the right path.
- Run retrospective meetings to analyze projects, look at what didn't go so well, and talk about how to improve things next time.

1.3 Effective research

Web developers spend a lot of time searching for solutions to problems encountered in their work. Students should learn effective strategies for finding answers, and when to use which methods (e.g. don't jump straight to pestering the senior dev every time you hit a roadblock).

These strategies include:

- Consulting the documentation.
  - When you are stuck with using a tool/product, consult the official documentation first. It is surprising how many people do not think to do this; official docs are often geared towards beginners, so people with experience may not think of them as being suitable for them.
  - Learn about different types of documentation — for example, tutorials, references, and other types — and understand when each type is useful.
- Using search engines effectively (See How to use search like a pro: 10 tips and tricks for Google and beyond).
- Choosing valid information sources:
  - Choose recommended sites such as Stack Overflow and MDN.
  - Check the dates on articles, and consider whether the techniques discussed are out-of-date. For example, does an article on CSS layout talk about modern approaches like grid and flexbox, or does it still present obsolete techniques like multi-column layouts with floats? Does it still talk about hacks for ancient browsers like Internet Explorer or Netscape 4?
- Using social media effectively:
  - Build a network of folks who can help.
  - Join community groups where you can look for answers. For example:
    - The MDN Web Docs community on Discord
    - Scrimba *Course partner*

    - freeCodeCamp
    - CodeNewbie
    - Dev.to
  - Learn to give back as well as take; web developers who reciprocate are much more likely to build strong relationships and keep getting help.
  - When you find a useful answer, write about it. For example, blog about it or share it on a social network. Not only will the process of writing clarify the concepts to you, but you'll also get validation and/or feedback from the community. It could also help you to start making a name for yourself in the industry.
- Making effective use of an experienced contact's time:
  - By "experienced contact", we mean a teacher, mentor, or senior developer.
  - Ask them what communication methods they prefer.
  - Think carefully about what questions to ask them beforehand, consider their time limited and precious.
  - Be sure to do some background research about the topic beforehand and don't ask questions that you can find answers to by searching the web or the official documentation.
  - Timebox the session to say 30 minutes.
  - Prioritize your issues.
  - Set a goal for the session, for example, "try to find a solution to the highest priority issue"; solving the biggest issue may also lead to a fix for other issues.
- Rubber ducking as an effective help mechanism. See also Rubber Duck Debugging.
- Using AI to help with coding issues (for example ChatGPT or GitHub Copilot). You should use AI tools with caution, and familiarize yourself with their strengths and weaknesses:
  - On the plus side, they can speed up research/searches enormously, and help with structuring code and copy.
  - On the other hand, AI tools have no reasoning skills and frequently provide answers that are misleading or just plain wrong. You shouldn't just assume that AI answers are correct, and test them/verify them with other sources.

Notes:

- There is definitely a balance to knowing the right time to ask for help. Web developers shouldn't constantly pester their peers/colleagues, but equally, they shouldn't soldier on and pretend they know what they are doing when they don't.
- Consider the value of saying "I don't know" at the right time.

Resources:

- Learning and getting help

1.4 Collaboration and teamwork

As a professional in the web industry, you are going to have to work with other people on projects, and while brainstorming ideas and proposals. Not everyone is born with an innate ability to work in a team, so it is beneficial to start incorporating some best practices early on and putting work into areas where you think you are lacking.

Recommendations:

- Learn about empathy, humility, conflict resolution, and cooperation. In all engagements, stay polite and respectful and do not use offensive language.
- While working in a team in the real world, you will frequently be expected to do peer reviews. Practice how to deliver feedback constructively and respectfully. When receiving feedback, practice how to not take it personally, and focus on the positives and what you can learn.
- Participate in pair programming, or work in teams on assessments to experience working with other people.
- Practice running projects like a real software project, with a timeline, plan, and responsibilities. Learn about the software development lifecycle. Pick up some basic project planning skills and tools to be able to estimate and plan your work/project.
- As part of the course, blog about your work, learnings, and roadblocks, share your code repositories, get peers to critique your work, and offer updates to fix issues in other people's work.
- Join a Slack channel, Discord, or a similar space, ask peers for help, share resources, and discuss the work to be done. For example:
  - Check out the Frontend Developers Discord server.
  - Our learning partner, Scrimba, provides a strong community and collaboration experience via their Discord server, intending to help their students gain exactly these kinds of skills.
- Practice asking and answering questions. Even if they seem somewhat trivial, always come up with one or two questions to ask when discussing or reviewing peer work. It is essential to practice explaining what you are doing and asking the right questions to find out what you need to know.
- Help each other, rather than waiting for a teacher or senior dev to go around and help everyone. Less able peers will get help more quickly, and more able peers will become mentors and experience the satisfaction that it brings.
- Observe and learn from other experienced folks how to engage in discussions as well as how to approach problem-solving/debugging.
- Join an open-source project to practice the skills you learn, engage with folks in the community, and learn from observing others (see How to Contribute to Open Source Projects – A Beginner's Guide for useful information).

Our learning partner, Scrimba, provides a strong community and collaboration experience via their Discord server, intending to help their students gain exactly these kinds of skills. Check out the following embedded content to learn more.

1.5 Succeeding in job interviews

Technical job interviews can be very demanding, and some have quite specific requirements.

Recommendations:

- Learn effective strategies for job searching. For example:
  - Attend networking events and job fairs to meet potential employers.
  - Keep an inventory of the people you meet and the companies you apply to.
  - Follow up with any promising leads you meet.
- Create a portfolio.
- Build the perfect resumé.
- Get experience — build real projects and contribute to open source.
- Build your online persona.
- Use sites like LinkedIn to help with the above.
- Practice writing answers for coding and design interview questions.
- Build a collection of anecdotes to use for experience-based interview questions.
- Be aware of the attributes that hiring managers look for in a candidate and prepare accordingly:
  - Someone they can get along with.
  - Positive attitude, respectful, empathic, constructive.
  - Open-minded and works well in a diverse team with diverse viewpoints.
  - Graceful when a decision does not go their way, able to align for the greater good.
  - Good communicator and relationship builder.
  - Tenacious, focused, good problem solver.
  - Having a good portfolio.
- Be patient. Even the best candidates will get rejections from multiple job applications before they land the job they want.

Resources:

- Getting hired, Scrimba *Course Partner*
- Technical Interviewing 101: Ultimate Guide to Acing Your Tech Interview, learntocodewith.me (2022)
- 30 Technical Interview Questions and Tips for Answering, Coursera (2023)

1.6 Workflows and processes

An important aspect of technical projects that beginners often miss out on is an idea of the bigger picture of a project. They might learn an individual tool or language and understand what they need to do, but be unaware of all the other codebases, tools, systems, and job roles that go together to deliver an entire web application. It is useful to understand the following at a high level:

- Typical technology combinations and application architectures in common web projects.
- Typical processes for a technical project, including where different tools are used in those processes.
- Typical job roles, and where they are involved in those processes.
- Common work management styles, such as agile and waterfall.

Resources:

- What is a Tech Stack and How Do They Work?, mongodb.com
- Website development team structure: roles and processes, truemark.dev (2017)
- Waterfall vs. Agile vs. Kanban vs. Scrum: What's the difference?, Asana (2022)

1.7 Relevant contextual information

While not essential for understanding the technical topics listed in the curriculum, there is a range of contextual information that can help developers gain a well-rounded and flexible perspective.

Recommendations:

- To understand why things are the way they are, study the relevant historical context. For example:
  - Why was the web designed like it is in terms of data delivery when arguably faster mechanisms exist?
  - Why does the web use a document model with links as a central feature when these days it is largely used to build apps?
  - Why are web standards created like they are, in collaboration, when the process isn't necessarily as efficient as it could be?
- Study general programming concepts where relevant, for example:
  - The purpose of objects, and what they enable in terms of the design of a language like JavaScript and its surrounding APIs.
  - How loops work and why they are needed.

2. Environment setup

This module includes topics related to the setup and usage of the computer system that you will use to implement websites/apps. These topics are not directly related to creating web code, but you will benefit greatly from understanding the operating system you are working with.

General resources:

- Windows help and learning, Microsoft (2024)
- macOS User Guide, Apple (2024)
- Official Ubuntu documentation, ubuntu.com (2024)

2.1 Computer basics

- Signing into your computer and connecting it to the internet.
- Using basic system control with keyboard, mouse, and other pointing devices.
- Installing applications.

Resources:

- Installing basic software

2.2 File systems

- Basic explorer/finder usage.
- Standard folder structure.
- File naming best practices for the web — no spaces, lowercase, choosing a reasonable separator like a hyphen or underscore.
- Basic file organization best practices.
- Creating, moving, and deleting files and folders using Explorer/Finder.
- Searching for files and folders.
- Dealing with file extensions (e.g. turning off "Hide extensions for known file types" in Windows, showing dot files (`.env`, etc.)).
- Learning how file types are associated with applications.

Resources:

- Dealing with files

2.3 Browsing the web

- Available web browsers.
- Installing a web browser.
- The difference between a web browser, a website, and a search engine.
- Basic search engine usage.

Resources:

- What is the difference between web page, website, web server, and search engine?
- How to use search like a pro: 10 tips and tricks for Google and beyond, theguardian.com (2016)

2.4 Command line basics

- Understand what the command line is, and what you can do with it.
- Understand how to access the command line on different systems:
  - On Linux and macOS, you've generally got a built-in terminal ready to go.

  - On Windows, the default command prompt is a bit more limited; you are better off installing something like Windows Subsystem for Linux (WSL), PowerShell or Git Bash (part of Git for Windows) if you want the same commands and power available to macOS/Linux.
- Shortcuts (e.g. up arrow to access previous commands, tab for autocomplete).
- Basic commands (e.g. `cd`, `ls`, `mkdir`, `touch`, `grep`, `cat`, `mv`, `cp`).
- Command options/flags.

Resources:

- Command line crash course
- Stack Overflow is a good place to find specific solutions to command line problems, for example How to Batch Rename Files in a macOS Terminal?

Notes:

The command line / terminal is intimidating to newcomers — you just get a blinking cursor, with no obvious signs of what to do next. We are not saying that you should be a command line wizard before you start learning web development, but you should at least understand what it is and know some basics — you will be surprised how often you come across command line usage in web development tooling.

2.5 Code editors

- Learn what code editors are available and what is suitable for your purposes:
  - Binary file editors like Microsoft Word are unsuitable for editing code. You need something that cleanly handles and outputs plain text.
  - Default OS plain text editors can be OK, for example, TextEdit on macOS, or Notepad on Windows, but they also have limitations.
  - You are better off with a fully-fledged code editor like VSCode (multiplatform, free), Sublime Text (multiplatform, not free) or Notepad++ (Windows, free).
  - Integrated Development Environments (IDEs) such as Visual Studio (Windows, not free), NetBeans (multiplatform, free), and WebStorm (multiplatform, not free) tend to have more features than simple code editors but tend to be more complex than what you need at this stage in your learning journey.
- Learn what a basic code editor can do for you:
  - Open and edit code files.
  - Syntax highlighting.
  - Auto-indentation and other simple syntax fixes.
  - Code completion and help.
  - Find and replace, often with the ability to use regular expressions to make the functionality more powerful (e.g. keep a specific string beginning and end, but replace the substring in between).

  - Integration with version control is often provided (see also Version control)
- Customize and enhance your code editor with extensions:
  - Language-specific extensions such as code completion, highlighting, linting, and debugging. This can apply to specific languages such as JavaScript, Python, or Go, or language/framework abstractions such as TypeScript or JSX.
  - GitHub/version control extensions, if not provided by default.
  - Theme and color scheme extensions.
  - Productivity extensions like code snippets and scaffolding generators.
  - AI-powered code suggestion tools such as GitHub Copilot. Be aware that, while useful, AI tools have no reasoning skill, and frequently provide answers that are misleading or just plain wrong. You shouldn't just assume that AI answers are correct, and test them/verify them with other sources.

Core modules

The core modules cover topics that we feel every web developer should have a good grounding in. This includes all the information they need to design and build a basic, accessible web app that follows modern best practices, and manage and deploy their code using appropriate tools.

General resources:

- The Frontend Developer Career Path from Scrimba: Teaches the MDN Curriculum Core with fun interactive lessons and challenges, knowledgeable teachers, and a supportive community. Go from zero to landing your first front-end job!

1. Web standards

This module covers the fundamentals of how the web works at a high level — including the model used for communication, the core technologies involved, how those technologies are created, and how a web browser renders and displays websites to a user.

General resources:

- Resilient Web Design, Jeremy Keith

1.1 How the web works

Learning outcomes:

- Clients and servers and their roles in the web.
- DNS and how it works at a high level.
- TCP/IP and HTTP.
- HTTP syntax at a basic level.
- Common HTTP response codes (e.g. 200, 301, 403, 404, and 500).
- Components of a URL (protocol, domain, and subdomain).

- TLDs and how to register a domain.
- Hosting, how to purchase it, and how to put a website online.

Notes:

- One of the key goals of this section is a high-level understanding of how the web functions behind the code.
- You should also gain a vocabulary to start talking about how the web functions precisely.

Resources:

- Requests and responses, Scrimba *Course Partner*
- How the web works
- How the Web Works: A Primer for Newcomers to Web Development (or anyone, really), freeCodeCamp (2015)
- What is a domain name?
- What is a URL?
- Publishing your website

1.2 The HTML, CSS, and JavaScript triangle

Learning outcomes:

- The purpose of HTML, CSS, and JavaScript.

Notes:

Purposes of the main web authoring technologies:

- HTML is for structure and semantics (meaning).
- CSS is for styling and layout.
- JavaScript is for controlling dynamic behavior.

- Their place in the larger ecosystem, and the fact that they are not the only web technologies.
- Why separating the layers is a good idea.

Notes:

Separation is a good idea for:

- Code management and comprehension.
- Teamwork/separation of roles.
- Performance.

- The fact that in reality, the separation is not always clear.

Notes:

- A prime example is the case of using JavaScript to dynamically update CSS styling on-the-fly in response to app state changes, user choices, etc.
- Often this is done by modifying the `Element.style.x` properties, which results in inline CSS being injected into HTML. A better strategy is to add/change classes on elements to avoid inline CSS.
- Much more severe is the case of JavaScript frameworks that use various HTML-in-JavaScript or CSS-in-JavaScript custom syntax, which results in a lot of mixing of syntax types.

- The nature of this separation — an ideal to aim for where possible rather than an absolute.
- The concept of progressive enhancement.

Notes:

Progressive enhancement is often seen as unimportant, because browsers tend to support new features more consistently these days, and people tend to have faster internet connections. However, you should think about examples relevant to the modern day — cutting down on decoration to make a mobile experience smoother and save on data, or providing a simpler, lower-bandwidth experience for users in developing countries who might still pay for home internet by the megabyte.

This bleeds over into responsive design, which is covered later on in more depth.

Resources:

- Intro to web dev basics, Scrimba *Course Partner*
- The web and web standards
- The Web Standards Model, explainers.dev
- What is Progressive Enhancement, and why it matters

1.3 The web standards model

Learning outcomes:

- How standards bodies operate — for example the W3C, WHATWG, TC39, and Khronos Group.
- The process of standards creation.

Notes:

- The basic principles of the web — interoperable, accessible, collaborative, and not owned by a single corporation.
- This basis means that the web is a unique and exciting industry to get involved in.
- The full W3C standards process is deep and academic. For now, you should understand how different individuals and companies get involved in the standards process, and how the different maturity stages are designed to weed out issues (e.g. interoperability issues, patent issues).

- The lifecycle of web standards features:
  - Experimental: Usually only available in one browser engine as it is developed, sometimes not in a specification yet. Too early to use in production.
  - Stable: Development finished, specified, available across browser engines.
  - Deprecated: Not to be used anymore, may still be in browsers but flagged for deletion.
- The key principles web standards are built on:
  - Open to contribute and use.
  - Not patent-encumbered or controlled by a single private entity.
  - Accessible and interoperable.
  - They don't break the web.

Resources:

- The web and web standards
- About W3C web standards, W3C
- The W3C recommendation track, W3C (2021)
- WHATWG FAQ, WHATWG
- Web Platform Design Principles, W3C (2023)

1.4 How browsers load webpages

Learning outcomes:

- The HTTP request-response model.
- The different kinds of assets that are returned in an HTTP response.

Notes:

The different kinds of downloaded resources to understand are:

- HTML files.
- CSS files.
- JavaScript files.
- Media assets such as images, videos, audio files, PDFs, and SVGs.
- Other kinds of file that the browser can't handle natively and hands off to a relevant app on the device, for example Word documents and PowerPoint slide decks.

- Static versus dynamic files: Some downloaded code files will be static (they exist on the server in the same form as they are downloaded), and some will be dynamic (generated by the server based on varying data).
- How these are assembled to create a web document that is then displayed by the browser.

Notes:

The different stages of rendering a web page:

- A web page is requested (e.g. by clicking a link).
- A DNS lookup is performed to find the location of all the assets to download for the web page.
- The assets start to be fetched. This involves TCP handshakes, TLS negotiation, and HTTP requests and responses.
- A DOM tree is assembled from the downloaded HTML.
- The CSSOM is built from the CSS rules.
- The JavaScript is parsed, interpreted, compiled, and executed.
- The accessibility tree is built for assistive technologies (e.g. screen readers) to hook into.
- The render tree is created from the DOM and CSSOM, to identify visual styles applied to each DOM node.
- Page layout is calculated.
- The styled, laid-out nodes are painted to the screen in the right order, via painting and compositing.

- Why the browser is sometimes seen as a hostile programming environment:
  - Unlike other programming environments, it is much harder to make guarantees about the environment your code will run on.
  - You cannot guarantee a user's OS, browser, language, location, network connection, CPU, GPU, memory, etc.
  - You need to embrace uncertainty and learn to program defensively. This feeds back into and expands upon the concepts looked at around progressive enhancement in 1.2 The HTML, CSS, and JavaScript triangle. This would also be a good place to look at related concepts such as error handling, feature detection, and responsive design.
- The flipside — why the web is an awesome programming environment:

- Its basic state is accessible and linkable. Some of these basics are harder to achieve in other environments.
- App delivery across the web is simple and powerful.
- Updates are easy — in many cases, you can just reload the browser tab. You don't need to worry about constantly downloading and installing large packages.
- The community is vibrant and helpful, and there are lots of great resources available to learn.

Resources:

- Populating the page: how browsers work
- Dealing with files
- How browsers work, freeCodeCamp (2018)

2. Semantic HTML

HTML is the technology that defines the content and structure of any website. Written properly, it should also define the semantics (meaning) of the content in a machine-readable way, which is vital for accessibility, search engine optimization, and making use of the built-in features browsers provide for content to work optimally. This module covers the basics of the language, before looking at key areas such as document structure, links, lists, images, forms, and more.

General resources:

- Learn HTML and CSS, Scrimba *Course Partner*
- The basics of semantic HTML, Scrimba *Course Partner*
- Structuring the web with HTML
- Learn HTML, Codecademy

2.1 Basic HTML syntax

Learning outcomes:

- The need for a doctype at the top of HTML documents. Its original intended purpose, and the fact that now it is somewhat of a historical artifact.
- The need to set the language of a document using the `lang` attribute in the opening `<html>` tag.
- The HTML head, and its purpose as a metadata container for the document including key uses:
  - Setting information like character encoding and title.
  - Providing metadata for search engines (e.g. `<meta name="description">`) and social media platforms (e.g. Open Graph Data), and the Search Engine Optimization (SEO) benefits.
  - Linking to icons for use on browsers and mobile platforms.
  - Linking to stylesheets and script files.
- The HTML body and its purpose as a container for the page content.
- The anatomy of an HTML element — element, opening tag, content, closing tag, attributes.

- What void elements (also known as empty elements) are, and how they differ from other elements.

Resources:

- Getting started with HTML
- What's in the head? Metadata in HTML

2.2 Good document structure

Learning outcomes:

- How to create a good document structure with headings and content beneath those headings.
- Using semantic HTML rather than presentational HTML:
  - Some presentational markup should no longer be used at all (e.g. `<big>` and `<font>`); it is deprecated.
  - Some presentational markup has been repurposed to have new semantic meaning (e.g. `<i>` and `<b>`).
  - It is tempting to just use `<div>` elements wherever a block-level container is required, but you should be aware of the other available structural elements and their benefits (such as improved accessibility). Examples include `<main>`, `<section>`, `<article>`, `<header>`, `<nav>`, and `<footer>`.

  > **Notes**:
  >
  > One key point to understand here is the difference between semantic and presentational markup, what these terms mean, and why semantic markup is important to SEO and accessibility.

- The need for heading levels to be used logically, i.e. no skipping levels or using them arbitrarily because you want to achieve a certain font size (that's a job for CSS).
- SEO benefits: for example, keywords are boosted in headings.

> **Notes**:
>
> This conformance criterium doesn't require that you go too deep into strategies for writing SEO-friendly content, although you should understand what this means.

- Accessibility benefits: Assistive technology (AT) such as screen readers use headings and landmarks as signposts to navigate content. HTML documents are very difficult for AT users to use without headings.
- Understanding that HTML needs to be correctly nested. If not, the browser has to guess what you meant your structure to be, and it might not be what you wanted.
- Validating your markup using the HTML validator or another similar tool (for example, view source in Firefox highlights validation errors with a dotted red underline).

Resources:

- HTML text fundamentals
- Document and website structure
- What is accessibility?

2.3 Lists

Learning outcomes:

- The HTML structure for the three types of lists — unordered, ordered, and description.
- Understand that description lists are less commonly used than the other two types, with use cases mainly in areas such as academia and documentation.
- The correct usage for each list type.

Notes:

- Unordered lists are for marking up a list of items where the order doesn't matter, such as a shopping list.
- Ordered lists are for marking up a list of items where the order does matter, such as a set of directions.
- Description lists are for associating a list of terms with descriptions of those terms, for example, product names and descriptions in a shopping cart.

- The broader use cases of lists, such as navigation menus.

Resources:

- HTML text fundamentals > Lists
- Advanced text formatting > Description lists

2.4 Advanced text techniques

Learning outcomes:

- Correct usage of elements for emphasis and importance, such as `<strong>` and `<em>`.
- Understand HTML elements that represent other less common semantic requirements, for example:
  - Quotations.
  - Abbreviations and acronyms.
  - Addresses.
  - Times and dates.
  - Superscript and subscript.
  - HTML entities.
  - Other text markup features such as `<u>`, `<s>`, and `<ruby>`.

> **Notes**:
>
> It is not necessary to have an exhaustive understanding of all the semantic elements HTML offers at this stage, but you should understand that they exist, and how to look them up using MDN if you need them.

Resources:

- HTML text fundamentals > Emphasis and importance
- Advanced text formatting
- A delightful reference for HTML Symbols, Entities and ASCII Character Codes, Toptal (2023)

2.5 Links

Learning outcomes:

- Understand why links are *the* fundamental feature of the web. There is no web without links.
- The `href` attribute.
- Absolute and relative paths, and when to use them.
- Path syntax in detail — slashes, single dot, and double dot.
- Link states and why they are important — `:hover`, `:focus`, `:visited`, and `:active`.
- Inline and block-level links.
- Understanding the benefits of writing good link text, such as better accessibility for screenreader users, and potential positive SEO effects.

Resources:

- Anchor tags, Scrimba *Course Partner*
- Creating hyperlinks

2.6 Media

Learning outcomes:

- The term "replaced element" — what does it mean?
- Images, audio, video:
  - The basics — `<img>`, `<audio>`, and `<video>` tags.
  - Using `src` to point to a resource (Paths are also important here; see 2.5 Links).
  - Using `width` and `height`, for example, to avoid unpleasant jerky updates to the UI once an image has finished loading and is displayed.
  - Video and audio-specific attributes such as `controls` and `muted`.
  - `<sources>`.
  - Optimizing media assets for the web — keep file sizes small.

- Media assets and licensing:
  - Different types of licensing — public domain/CC0, permissive (e.g. CC license, MIT), copyrighted (rights-managed/royalty-free).
  - Searching for appropriately licensed media files to use in projects, e.g. via Google Images, Flickr, and The Noun Project.
  - Complying with license requirements.
- Alternative text ("alt text") for media.

Resources:

- What is multimedia?, Geeks for geeks (2023)
- Multimedia and embedding

2.7 Other interactive elements

Learning outcomes:

- Aside from links, `<button>` and form elements are the main tools for building controls for users to interact with your sites.

> **Notes**:
>
> There are a lot of input types and form features not explicitly mentioned here; the purpose is to get a good general introduction to buttons and form elements, and learn the most common cases. The advanced/specialized cases can be studied on a need-to-know basis, as part of a web developer's constant learning throughout their career.

- `<button>`:
  - Button types — `button`, `submit`, and `reset`.
  - Why reset buttons are nearly always a bad idea.
- Common `<input>` types — `text`, `number`, `file`, `checkbox`, `radio`, `password`, `search`, and `submit`.
- Common attributes — `name` and `value`.
- Client-side validation basics — `required`, `min`, `max`, `minlength`, `maxlength`, and `pattern`.

> **Notes**:
>
> Make sure to understand that client-side form validation is really a usability enhancement, to be used alongside server-side form validation. It is not a substitute for it.

- Making forms accessible — Correct semantics, `<label>`, and the `for` attribute.

Notes:

- Going back to the argument for semantic HTML (see also 2.2 Good document structure), you should understand why it is important to use the right element for the right job. For example, use a `<button>` to submit your form, and not a `<div>` programmed to behave like a `<button>`.
- Understand the features programmed into `<button>`s and form elements by the browser by default, and how important they are. Examples include keyboard accessibility, focus outlines, and semantic meaning for AT to identify the elements and communicate their meaning.

- Form states and why they are important — `:focus`, `:readonly`, `:disabled`, etc.
- `<textarea>`.
- `<select>` and `<option>`.
- `<form>` elements:
  - Form submission: what happens when a form is submitted.
  - The difference between submission methods — `GET`, `POST`, etc.

Resources:

- Input tags, Scrimba *Course Partner*
- Web forms — Working with user data

2.8 HTML tables

Learning outcomes:

- What tables are for — structuring tabular data.
- What tables are not for — layout, or *anything else*.
- Basic tables — `<table>`, `<tr>`, and `<td>`.
- `colspan` and `rowspan`.
- Better table structuring for accessibility — `<th>`, `<thead>`, `<tbody>`, `<tfoot>`, `<caption>`, and the `scope` attribute.

Resources:

- HTML tables

2.9 Debugging HTML

Learning outcomes:

- Use the HTML validator to see if you have any markup errors. This is always a good place to start if you are experiencing unexpected behavior.
- View source is a useful tool for getting a quick look at the source markup of a page.

- Using the DOM inspector in your browser DevTools to dive deeper into your markup:
  - Add and remove elements and attributes on the fly to see what effect it has.
  - Add and remove classes on the fly to see if the associated CSS is applied as expected.

Resources:

- Debugging HTML
- Firefox Docs > Examine and edit HTML, Firefox Source Docs
- Chrome DevTools > Get started with viewing and changing the DOM, developer.chrome.com (2019)

3. CSS fundamentals

CSS enables you to add style to your web pages including color, text, positioning and layout, and animation. In our first CSS module, we cover all the fundamental language mechanics you need to understand how CSS works.

General resources:

- Learn HTML and CSS, Scrimba *Course Partner*
- Write your first lines of CSS!, Scrimba *Course Partner*

3.1 Basic CSS syntax

Learning outcomes:

- The purpose of CSS — style, layout, and provide other visual enhancements to web pages (such as animation).
- Key CSS syntax:
  - Rules.
  - Selectors.
  - Declarations.
  - Properties (including custom properties).
  - Values (including shorthand values).
  - At-rules and descriptors.
- Default browser styles — understand that the browser provides default CSS styling to HTML elements so that it is in some way usable even with no user-defined styles at all:
  - Understand also therefore that HTML has nothing to do with styling.
  - Use this to reinforce the idea of separating semantics and structure (semantic HTML) from presentation (CSS), and not using presentational markup.
  - Study CSS resets, first to prove that browser styles exist and show what a page looks like when they are removed, but also as a technique for providing a blank canvas for developers to build styles on top of.

- Applying CSS to an HTML document — inline styles, internal stylesheets, external stylesheets:
  - Why external stylesheets are usually the best option.

Resources:

- What is CSS?
- Getting started with CSS
- How CSS is structured
- How CSS works

3.2 Selectors

Learning outcomes:

- Basic selectors — element type, class, ID:
  - IDs are unique per document — you should use an ID to select one specific element.
  - You can have multiple classes per element, and these can be used to layer on styles as required.
  - IDs and classes should be used sparingly where they make sense for selections, but you shouldn't use them for everything — keep your HTML as clean and uncluttered as possible.
- Selector lists.
- Attribute selectors.
- Combinators.
- Pseudo-classes and pseudo-elements.

Resources:

- CSS Selectors
- CSS classes, Scrimba *Course Partner*
- CSS Selectors – Cheat Sheet for Class, Name, Child Selector List, freeCodeCamp (2022)

3.3 The box model

Learning outcomes:

- Block and inline elements.
- The different boxes that make up an element and how to style them:
  - `width` and `height`.
  - `margin`.
  - `border`.
  - `padding`.
- The alternative box model accessed via `box-sizing: border-box`, and why this is easier to understand (and how it differs from) the standard box model.

- Margin collapsing.
- Basic display values, and how they affect box behavior — `block`, `inline`, `inline-block`, and `none`.

Resources:

- The box model
- Box Model, web.dev (2019)

3.4 Handling conflicts in CSS

Learning outcomes:

- Understand how rules can conflict in CSS.
- Inheritance.
- The cascade.
- The concepts that govern the outcome of CSS conflicts:
  - Specificity.
  - Source order.
  - Importance.

Resources:

- Cascade, specificity, and inheritance

3.5 Values and units

Learning outcomes:

- Understand that property values can take many different types, and what these types represent:
  - Numbers, lengths, and percentages.
  - Ems and rems.
  - Colors.
  - Images.
  - Positions.
  - Strings and identifiers.
  - Functions.
- Understand what absolute and relative units are, and the difference between them.

Resources:

- CSS values and units

3.6 Sizing

Learning outcomes:

- Intrinsic size.

- Setting absolute and percentage sizes.
- `min-`/`max-width` and `min-`/`max-height`.
- Viewport units.

Resources:

- Sizing items in CSS
- Handling different text directions > Logical properties

3.7 Backgrounds and borders

Learning outcomes:

- Basic background styling — colors and images.
- Background image size, repeat, position, and attachment.
- Background gradients:
  - The general concept of what a background gradient is.
  - Linear gradients.
  - (Radial, conic, and repeating gradients are more advanced and in-depth coverage is not required at this stage.)
- Accessibility considerations of backgrounds — ensure good contrast.
- Border basics — `border-width`, `border-style`, `border-color`, and border shorthand (e.g. `border-top` and `border`).
- `border-radius` for rounded corners.

Resources:

- Border and border-radius, Scrimba *Course Partner*
- Backgrounds and borders

3.8 Overflow

Learning outcomes:

- Understand what overflow is.
- Control overflow with `overflow` properties.

Resources:

- Overflowing content

3.9 Styling form elements

Learning outcomes:

- Basic styling of easy-to-style form elements, like `<input type="text">`.
- Using CSS resets to overcome `<input>` font styling inheritance and box styling default differences.

- Understand that not all form elements are easy to style, and why:
  - System styles are applied to some form elements, making consistent styling difficult across browsers.
  - More complex form elements have internal (shadow DOM) elements that define the structure of their inner workings. These are often impossible to access and style individually.
- Using `appearance: none` to work around system styling for `<input>` types like `search`, `checkbox`, and `radio`.
- Mitigating issues with difficult-to-style types such as `datetime-local`, `color`, etc.

Notes:

Conforming to this curriculum module doesn't require having foolproof, conclusive answers to every possible form styling problem. Some form elements are difficult to style, as the resources make clear. However, you should at least be able to handle a wide range of form styling needs and understand the issues around some of the more difficult styling issues.

Resources:

- Images, media, and form elements
- Styling web forms
- Advanced form styling

3.10 Debugging CSS

Learning outcomes:

- Use the HTML validator to see if you have any invalid markup on your page — this could be causing your CSS to not apply as desired.
- Use the CSS validator to check for badly-formed CSS code. A missing semi-colon can cause a whole section of CSS declarations to not apply.
- Use browser developer tools to inspect the CSS that is applied to HTML elements on a page.
- Modify the applied CSS to figure out what changes are needed to get what you want. This includes enabling and disabling declarations, modifying values, and adding new declarations.
- Use layout inspection tools to inspect the box model, grids, flexbox, and other layout features (see also CSS Layout).
- Use responsive design mode tools to check responsive layouts (see also 5.5 Responsive design specifics).

Resources:

- Debugging CSS
- Handling common HTML and CSS problems
- Firefox > Examine and edit CSS, Firefox Source Docs
- Firefox > Responsive design mode, Firefox Source Docs

- Chrome > View and change CSS, developer.chrome.com)

4. CSS text styling

This module focuses specifically on CSS font and text styling, including loading custom web fonts and applying them to your text.

4.1 Text and font styling

Learning outcomes:

- `color`.
- Font family, font stacks, web safe fonts.
- `font-size`, `font-weight`, and `font-style`.
- `text-align`, `text-transform`, and `text-decoration`.
- `text-shadow`.
- `line-height`.

> **Notes**:
>
> There are several other font and text styling properties, and students should be encouraged to explore more of them as part of their constant learning.

Resources:

- Fundamental text and font styling
- Text and typography, web.dev (2021)
- Web-safe fonts, Scrimba *Course Partner*

4.2 Styling lists and links

Learning outcomes:

- Spacing list items, for example, with `margin` or `line-height`.
- `list-style` properties.
- Understand why default link styles are important for usability on the web — they are familiar and help users recognize links.
- Styling link states: `:hover`, `:focus`, `:visited`, and `:active`:
  - Understand why these are necessary for usability and accessibility.
- Creating a navigation menu with lists and links.

Resources:

- Styling lists

- Styling links

4.3 Web fonts

Learning outcomes:

- Understand that web fonts allow developers to go beyond the web safe font set and use custom fonts on their web apps.
- Basic setup — the `@font-face` at-rule, and `font-family` and `src` descriptors.
- Using a web font with the `font-family` property.
- Other descriptors — `font-weight`, `font-style`, etc.
- Using an online service to find web fonts and generate web font code, for example, Font Squirrel and Google Fonts.
- Usability implications of web fonts — using several of them can increase page download size.

Resources:

- Web fonts
- Fonts Knowledge, Google Fonts
- All about the CSS font-family property, explainers.dev

5. CSS layout

Our final core CSS module looks deep into another crucial topic — creating layouts for modern websites. This module looks at floats, positioning, other modern layout tools, and building responsive designs that will adapt to different devices, screen sizes, and resolutions.

5.1 CSS layout basics

Learning outcomes:

- Understand that normal flow is the default way a browser lays out block and inline content.
- Properties such as `display`, `float`, and `position` are intended to change how the browser lays out content.

Resources:

- Introduction to CSS layout
- Normal flow

5.2 Floats

Learning outcomes:

- Understand the purpose of floats — for floating images inside columns of text, or possibly other fun techniques like drop caps and floating inset information boxes.

- Understand that floats used to be used for multiple-column layouts, but this is no longer the case now better tools are available (see 5.4 Modern layout for details).
- Using the `float` property to create floats.
- Clearing floats using `clear`, and the `display: flow-root` value.

Resources:

- Floats
- All About FLoats, CSS-Tricks (2021)

5.3 Positioning

Learning outcomes:

- Understand that `static` positioning is the default way elements are positioned on the page.
- Relative positioning:
  - Understand that relatively positioned elements remain in the normal flow.
  - Final layout position can be modified using the `top`, `bottom`, `left`, and `right` properties.
- Absolute positioning:
  - Absolute (and fixed/sticky) positioning takes elements completely out of the normal flow to sit in a separate layer.
  - `top`, `bottom`, `left`, `right`, and `inset` have different effects on absolutely-positioned elements than on relatively-positioned ones.
  - Setting the positioning context of a positioned element by positioning an ancestor element.
- Fixed and sticky positioning:
  - Understand how these differ from absolute positioning.
- Understand what z-index is, and how to control the stacking of positioned elements with the `z-index` property.

Resources:

- Positioning
- Aside: Position: relative & absolute, Scrimba *Course Partner*
- Stacking context

5.4 Modern layout

Learning outcomes:

- In general, gain an understanding of modern CSS layout techniques.
- Understand that, for basic placement tasks, the below tools could be overkill. Learn simple old-school techniques and where they are still effective:
  - Margins and padding for spacing.
  - Auto margins for horizontal centering tasks (e.g. `margin: 0 auto`).

- Flexbox:
  - Understand the purpose of flexbox — flexibly lay out a set of block or inline elements in one dimension.
    - See We have a problem that flexbox can fix by Scrimba *Course Partner* for a use case example.
  - Understand flex terminology — flex container, flex item, main axis, and cross axis.
  - `display: flex`, and what it gives you by default.
  - Rows and columns, and how to wrap content onto new rows and columns.
  - Flexible sizing of flex items.
  - Justifying and aligning content.
  - Adjusting flex item ordering.
- CSS Grid:
  - Understand the purpose of CSS Grid — flexibly lay out a set of block or inline elements in two dimensions.
  - Understand grid terminology — rows, columns, gaps, and gutters.
  - `display: grid`, and what it gives you by default.
  - Defining grid rows and columns:
    - The `fr` unit.
    - `minmax()`.
  - Defining gaps.
  - Positioning elements on the grid.

Resources:

- Flexbox
- Grids

5.5 Responsive design

Learning outcomes:

- Understand what responsive design is — designing web layouts so that they are flexible and work well across different device screen sizes, resolutions, etc.
- Understand the relationship between modern layout tools such as grid and flexbox, and responsive design.
- Media queries:
  - The mobile-first technique.
  - Understand breakpoints.
  - Using `width` and `height` media queries to create responsive layouts.

- `<meta viewport="">`, and how to use it to get web documents to display appropriately on mobile devices.
  - For the sake of accessibility, never set `user-scalable=no`.

Resources:

- Build a responsive site: Module intro, Scrimba *Course Partner*
- Responsive design
- Beginner's guide to media queries

6. JavaScript fundamentals

JavaScript is a huge topic, with so many different features, styles, and techniques to learn, and so many APIs and tools built on top of it. This module focuses mostly on the essentials of the core language, plus some key surrounding topics — learning these topics will give you a solid basis to work from.

General resources:

- Learn JavaScript, Scrimba *Course partner*
- What is JavaScript?
- A first splash into JavaScript

6.1 Variables

Learning outcomes:

- Understand what variables are and why they are so important in programming generally, not just JavaScript.
- Declaring variables with `let` and initializing them with values.
- Reassigning variables with new values.
- Creating constants with `const`.
- The difference between variables and constants, and when you would use each one.
- Understand variable naming best practices. If not explicitly covered, all examples should show good variable naming pratices in action.
- The different types of value that can be stored in variables — strings, numbers, booleans, arrays, and objects.

Resources:

- Storing the information you need — Variables

6.2 Math

Learning outcomes:

- Basic number operations in JavaScript — add, subtract, multiply, and divide.

- Understand that numbers are not numbers if they are defined as strings, and how this can cause calculations to go wrong.
- Converting strings to numbers with `Number()`.
- Operator precedence.
- Incrementing and decrementing.
- Assignment operators, e.g. addition assignment and subtraction assignment.
- Comparison operators.
- Basic `Math` object methods, such as `Math.random()`, `Math.floor()`, and `Math.ceil()`.

Resources:

- Basic math in JavaScript — numbers and operators
- Numbers and dates

6.3 Text

Learning outcomes:

- Creating string literals.
- Understand the need for matching quotes.
- String concatenation.
- Escaping characters in strings.
- Template literals:
  - Using variables in template literals.
  - Multiline template literals.
- String manipulation using common properties and methods such as:
  - `length`.
  - `toString()`.
  - `includes()`.
  - `indexOf()`.
  - `slice()`.
  - `toLowerCase()` and `toUpperCase()`.
  - `replace()`.

Resources:

- Handling text — strings in JavaScript
- Useful string methods

6.4 Arrays

Learning outcomes:

- Understand what an array is — a structure that holds a list of variables.
- The syntax of arrays — `[a, b, c]` and the accessor syntax, `myArray[x]`.
- Modifying array values with `myArray[x] = y`.
- Array manipulation using common properties and methods, such as:
  - `length`.
  - `indexOf()`.
  - `push()` and `pop()`.
  - `shift()` and `unshift()`.
  - `join()` and `split()`.
- Advanced array methods such as `forEach()`, `map()` and `filter()`.

Resources:

- Aside: Intro to arrays, Scrimba *Course Partner*
- Arrays

6.5 Conditionals

Learning outcomes:

- Understand what a conditional is — a code structure for running different code paths depending on a test result.
- `if ... else ... else if`.
- Using comparison operators to create tests.
- AND, OR, and NOT in tests.
- Switch statements.
- Ternary operators.

Resources:

- Making decisions in your code — conditionals

6.6 Loops

Learning outcomes:

- Understand the purpose of loops — a code structure that allows you to do something very similar many times without repeating the same code for each iteration.
- Basic `for` loops.
- Looping through collections with `for ... of`.

Notes:

There are many other types of loop in JavaScript that we haven't listed here. It is not necessary (or useful) to understand all of them at this stage. For now, students need to understand the purpose of loops, and the most common types.

- `break` and `continue`.

Resources:

- Looping code

6.7 Functions

Learning outcomes:

- Understand the purpose of functions — to enable the creation of reusable blocks of code that can be called wherever needed.
- Understand that functions are used everywhere in JavaScript and that some are built into the browser and some are user-defined.
- Understand the difference between functions and methods.
- Invoking a function.
- Return values.
- Understand global scope and function/block scope.
- Passing in arguments to function calls.
- Named and anonymous functions.
- Building your own custom functions:
  - Including parameters.
  - Including return values.
- Callback functions — understand that arguments to functions can themselves be functions, and what this pattern is used for.
- Arrow functions.

Resources:

- Using functions to write less code, Scrimba *Course Partner*
- Functions — reusable blocks of code
- Build your own function
- Function return values
- Arrow function expressions

6.8 JavaScript object basics

Learning outcomes:

- Understand that in JavaScript most things are objects, and you've probably used objects every time you've touched JavaScript.
- Basic syntax:
  - Object literals.
  - Properties and methods.
  - Nesting objects and arrays in objects.
- Using constructors to create a new object.
- Object scope, and `this`.
- Accessing properties and methods — bracket and dot syntax.
- Object destructuring.

Resources:

- JavaScript object basics
- Object destructuring assignment

6.9 DOM scripting

Learning outcomes:

- Understand what the DOM is — the browser's internal representation of the document's HTML structure as a hierarchy of objects, which can be manipulated using JavaScript.
- Understand the important parts of a web browser and how they are represented in JavaScript — `Navigator`, `Window`, and `Document`.
- Understand how DOM nodes exist relative to each other in the DOM tree — root, parent, child, sibling, and descendant.
- Getting references to DOM nodes, for example with `querySelector()` and `getElementById()`.
- Creating new nodes, for example with `innerHTML()` and `createElement()`.
- Adding and removing nodes to the DOM with `appendChild()` and `removeChild()`.
- Adding attributes with `setAttribute()`.
- Manipulating styles with `Element.style.*` and `Element.classList.*`.

Resources:

- Manipulating documents
- DOM Scripting, explainers.dev

6.10 Events

Learning outcomes:

- Understand what events are — a signal fired by the browser when something significant happens, which the developer can run some code in response to.
- Event handlers:
  - `addEventListener()` and `removeEventListener()`
  - Event handler properties.
  - Inline event handler attributes, and why you shouldn't use them.
- Event objects.
- Preventing default behavior with `preventDefault()`.
- Event delegation.

Resources:

- Introduction to events

6.11 Async JavaScript basics

Learning outcomes:

- Understand the concept of asynchronous JavaScript — what it is and how it differs from synchronous JavaScript.
- Understand that callbacks and events have historically provided the means to do asynchronous programming in JavaScript.
- Modern asynchronous programming with async functions and `await`:
  - Basic usage.
  - Understanding async function return values.
  - Error handling with `try ... catch`.
- Promises:
  - Understand that `async`/`await` use promises under the hood; they provide a simpler abstraction.
  - Chaining promises.
  - Catching errors with `catch()`.

Resources:

- Asynchronous JavaScript

6.12 Network requests with fetch()

Learning outcomes:

- Understand that `fetch()` is used for asynchronous network requests, which is by far the most common asynchronous JavaScript use case on the web.
- Common types of resources that are fetched from the network:
  - Text content, JSON, media assets, etc.

    - Data from RESTful APIs. Learn the basic concepts behind REST, including common patterns such as CRUD.
- Understand what single-page apps (SPAs) are, and the issues surrounding them:
    - Accessibility issues behind asynchronous updates, for example, content updates not being announced by screen readers by default.
    - Usability issues behind asynchronous updates, like loss of history and breaking the back button.
- Understand HTTP basics. You should look at common HTTP methods such as `GET`, `DELETE`, `POST`, and `PUT`, and how they are handled via `fetch()`.

Resources:

- Fetching data from the server

6.13 Working with JSON

Learning outcomes:

- Understand what JSON is — a very commonly used data format based on JavaScript object syntax.
- Understand that JSON can also contain arrays.
- Retrieve JSON as a JavaScript object using mechanisms available in Web APIs (e.g. `Response.json()` in the Fetch API).
- Access values inside JSON data using bracket and dot syntax.
- Converting between objects and text using `JSON.parse()` and `JSON.stringify()`.

Resources:

- Working with JSON
- JSON Review, Scrimba *Course Partner*

6.14 Libraries and frameworks

Learning outcomes:

- Understand what third-party code is — functionality written by someone else that you can use in your own project, so you don't have to write everything yourself.
- Why developers use third-party code:
    - Efficiency and productivity: A huge amount of complex functionality is already written for you to use, created in a way that enforces efficient, modular code organization.
    - Compatibility: Reputable framework code is already optimized to work across browsers/devices, for performance, etc. Many frameworks also have systems to output to specific platforms (e.g. Android or iOS) as build targets.
    - Support/ecosystem: Popular frameworks have vibrant communities and help resources to provide support, and rich systems of extensions/plugins to add functionality.

- The difference between libraries and frameworks:
  - A library tends to be a single code component that offers a solution to a specific problem, which you can integrate into your own app (for example, chart.js for creating `<canvas>`-based charts, or three.js for simplified 3D GPU-based graphics rendering), whereas a framework tends to be a more expansive architecture made up of multiple components for building complete applications.
  - A library tends to be unopinionated about how you work with it in your codebase, whereas a framework tends to enforce a specific coding style and control flow.
- Why should you use frameworks?
  - They can provide a lot of functionality and save you a lot of time.
  - A lot of companies use popular frameworks such as React or Angular to write their applications, therefore a lot of jobs list frameworks as requirements for applicants to have.
- Why is a framework not always the right choice? A framework:
  - Can easily be overkill for a small project — you might be better off writing a few lines of vanilla JavaScript to solve the problem or using a tailored library.
  - Usually adds a lot of JavaScript to the initial download of your application, leading to an initial performance hit and possible usability issues.
  - Usually comes with its own set of custom syntax and conventions, which can introduce a significant additional learning curve to the project.
  - May be incompatible with an existing codebase because of its architecture choice.
  - Will need to be updated regularly, possibly leading to extra maintenance overhead for your application.
  - May introduce significant accessibility issues for people using assistive technologies because of its architecture (for example, SPA-style client-side routing), which will need to be considered carefully.
- How to choose? A good library or framework must:
  - Solve your problems while offering advantages that significantly outweigh any negatives that it brings to the table.
  - Have good support and a friendly community.
  - Be actively maintained — don't choose a codebase that has not been updated for over a year, or has no users.

Resources:

- Introduction to client-side frameworks
- Introduction to React, Scrimba *Course Partner*

6.15 Debugging JavaScript

Learning outcomes:

- Understand the different types of JavaScript errors, for example, syntax errors and logic errors.

- Learn about the common types of JavaScript error messages and what they mean.
- Using browser developer tools to inspect the JavaScript running on your page and see what errors it is generating.
- Using `console.log()` and `console.error()` for simple debugging.
- Error handling:
  - Using conditionals to avoid errors.
  - `try ... catch`.
  - `throw`.
- Advanced JavaScript debugging with breakpoints, watchers, etc.

Resources:

- What went wrong? Troubleshooting JavaScript
- Control flow and error handling > Exception handling statements
- The Firefox JavaScript debugger, Firefox Source Docs
- Chrome > Console overview, developer.chrome.com (2019)
- Chrome > Debug JavaScript, developer.chrome.com (2017)

7. Accessibility

Access to web content such as public services, education, e-commerce sites, and entertainment is a human right. No one should be excluded based on disability, race, geography, or other human characteristics. This module discusses the best practices and techniques you should learn to make your websites as accessible as possible.

General resources:

- Let's learn to make the web accessible, Scrimba *Course Partner*

7.1 Accessibility basics

Learning outcomes:

- Understand the point of accessibility — increased usability for everyone, better SEO, and a wider target audience. Also, be aware of the legal requirements.
- Understand that accessibility should be considered from the start of a project, and not bolted on at the end.
- Understand the Web Content Accessibility Guidelines (WCAG) conformance criteria.
- Use semantic HTML, aka "The right element for the right job", because the browser provides so many built-in accessibility hooks. Good examples are `<a>` and `<input>` elements.
- Accessible best practices:
  - Alt text (see also 2.6 Media).
  - Good link text (see also 2.5 Links).

    - `<label>`s for form elements (see also 2.7 Other interactive elements).
    - Mobile browsers that provide specific usability advantages for specific `<input>` types such as `number` or `tel`.
    - Making tables accessible with `<th>`, `<thead>`, `<tbody>`, `<tfoot>`, `<caption>`, and the `scope` attribute (See also 2.8 HTML tables).
    - Using simple plain language, steering clear of slang and abbreviations where possible, and providing definitions where it is not possible.
- Understand the purpose of audio transcripts and text tracks (captions, subtitles, etc.) in making audio and video content accessible (we are not expecting mastery in creating them; that is an advanced topic in its own right).
- Keyboard accessibility:
    - Understand why apps need to be keyboard accessible — many people have difficulty using a mouse or other pointing device.
    - Understand built-in browser keyboard controls.
    - Understand when `accesskey` and `tabindex` are appropriate to use.

Resources:

- What is accessibility?
- HTML: A good basis for accessibility
- Handling common accessibility problems
- Understanding the Web Content Accessibility Guidelines

7.2 Accessible styling

Learning outcomes:

- Text sizing and layout:
    - Make sure your text is well laid out, consistent, and legible.
    - Consider providing large-type interfaces for those with visual impairments.
    - See also 4.1 Text and font styling.
- Color contrast:
    - Use an online tool such as WebAIM's Color Contrast Checker or the TPGi Colour Contrast Analyzer to check whether your color contrast conforms to the relevant WCAG conformance criteria.
    - Be mindful of those with color blindness or visual impairments; provide high-contrast modes to suit.
- `:focus` and `:hover` styles:
    - These are important cues for mouse and keyboard users.
    - See also 4.2 Styling lists and links.
- Sensible animation usage — use animation subtly and provide controls to turn it off:

  - Consider the needs of those with certain cognitive disabilities.
  - The `prefers-reduced-motion` media query was created specifically to help with this.
- Best practices for hiding content so that it doesn't become inaccessible. For example, `display: none` makes content unreadable by screen readers, so it needs to be used carefully.

Resources:

- CSS and JavaScript accessibility best practices > CSS
- Inclusive design principles, inclusivedesignprinciples.org
- Accessibility Design Guide, wiki.mozilla.org (2023)

7.3 Accessible JavaScript

Learning outcomes:

- Understand that there is such a thing as too much JavaScript. A simpler approach is usually more accessible, and often better for everyone.
- Understand the value of unobtrusive JavaScript:
  - If possible, use JavaScript as a usability enhancement, which isn't essential for the app to function.
  - A good example is client-side validation of form inputs.
- Use events sensibly so you don't lock out specific control types. For example, mouse-specific events such as `mouseover` and `mouseout` could lock out keyboard or touchscreen users.

Resources:

- CSS and JavaScript accessibility best practices > JavaScript
- Mobile accessibility
- Validating input, W3C (2019)

7.4 Assistive technology

Learning outcomes:

- Screen readers and other assistive technology (AT) types:
  - What they are used for, and who uses them.

> **Notes**:
>
> - The aim here is not to master the usage of all assistive technology types (there are many that we have not listed below), but to be aware of their existence and the types of people who use them, and also to appreciate how and why accessibility best practices work.
> - It is also a good idea for web developers to use screen readers or other types of assistive technology, to get an idea of what the web experience is like for users of those technologies.

    - The importance of source order.
    - The accessibility layer in browsers, and how assistive technologies hook into it.
    - Setting up screen readers and using them to test websites on desktop and mobile.
    - Other assistive technology such as:
        - Large text or braille keyboards.
        - Alternative pointing devices such as trackballs, joysticks, and touchpads.
        - Screen magnifiers.
        - Voice recognition software.
        - Switch controls.
- Auditing tools such as the Firefox Accessibility Inspector, the ANDI bookmarklet, Wave, and Google Lighthouse accessibility audits.

Resources:

- Handling common accessibility problems > Accessibility tools
- Mobile accessibility
- How People with Disabilities Use the Web, W3C (2017)
- WebAIM accessibility tooling articles, WebAIM

7.5 WAI-ARIA

Learning outcomes:

- Understand the purpose of WAI-ARIA — to provide semantics to otherwise non-semantic HTML, so that AT users can make sense of the interfaces being presented to them:
    - The first rule of ARIA: "If you can use a native HTML element or attribute with the semantics and behavior you require already built in, instead of re-purposing an element and adding an ARIA role, state or property to make it accessible, then do so."
    - In other words, using semantic HTML is an ideal, which is not possible at all times. WAI-ARIA is a bridging technology for such cases.
- The basic syntax — roles, properties, and states.
- Landmarks and signposting.
- Enhancing keyboard accessibility.
- Announcing dynamic content updates with live regions.

Resources:

- ARIA, Scrimba *Course Partner*
- WAI-ARIA basics
- ARIA Authoring Practices Guide (APG). W3C

8. Design for developers

The idea of this module is to (re-)introduce developers to design thinking. They may not want to work as designers, but having some basic user experience and design theory is good for everyone involved in building websites, no matter what their role. At the very least, even the most technical, "non-designer" developer should understand design briefs, why things are designed as they are, and be able to get into the mindset of the user. And it'll help them make their portfolios look better.

In addition, front-end developers are often tasked with doing various bits of design work on projects. Clients and employers often assume that they can do it because they are involved with the visual elements of the website. Historically, "web developer" used to be more of a hybrid designer/developer role than it is today.

General resources:

- Learn UI Design Fundamentals, Scrimba *Course Partner*
- The Shape of Design, Frank Chimero
- Designing for the Web, Mark Boulton
- Design for web, Prisca Schmarsow + other contributors. Highlights include:
  - Design trampoline: Learn design theory basics, Anna Riazhskikh
  - Web typography made simple, Hannah Boom
- Practical Typography, Matthew Butterick
- Web Style Guide, Patrick J. Lynch and Sarah Horton
- Visual design rules you can safely follow every time, Anthony Hobday
- 16 little UI design rules that make a big impact, Adham Dannaway

8.1 Basic design theory

Learning outcomes:

- UI design fundamentals:
  - Contrast.
  - Typography.
  - Visual Hierarchy.
  - Scale.
  - Alignment.
  - Use of whitespace.
- Color theory.
- Use of images.

Resources:

- Fundamental text and font styling

8.2 User-centered design

Learning outcomes:

- Understand that everything we do is for the user.
- Intro to user research/testing, and user requirements.
- Design for accessibility — consider the target audience and what additional needs they may have. Design for those from the very start.
- Understand what design patterns are, and the common patterns used on the web, for example:
  - Dark mode.
  - Breadcrumbs.
  - Cards.
  - Deferred/Lazy registration.
  - Infinite scroll.
  - Modal dialogs.
  - Progressive disclosure.
  - Progress indication on forms/registration/setup.
  - Shopping cart.

Resources:

- Accessibility overview
- Inclusive design principles, inclusivedesignprinciples.info

8.3 Design briefs

Learning outcomes:

- Speaking design language, to communicate with designers.
- Interpreting design brief requirements to produce an implementation.
- Typical tools designers use to get their message across to developers (e.g. Figma).

9. Version control

Version control tools are an essential part of modern workflows, for backing up and collaborating on codebases. This module takes you through the essentials of version control using Git and GitHub.

Learning outcomes:

- Understand why version control systems are necessary.

Notes:

- Git and associated social coding sites like GitHub have a lot of functionality, and can be intimidating and unfriendly to begin with. This set of conformance criteria does not expect mastery of these tools, but rather an understanding of the basics, and why it is necessary to have some experience here before entering the industry.
- Git is the web industry standard for version control, and has been for some time.

- Understand the difference between Git, and websites like GitHub and GitLab.
- Understand that websites such as GitHub and GitLab enable teamwork and collaboration that isn't so easy just with plain Git.
- Basic setup — installing git, signing up for an account for your chosen social coding site.
- Handling security requirements, like SSH/GPG keys.
- Creating a repo.
- Pushing changes — `add`, `commit`, and `push`.
- Contributing to others' repos:
  - Forking.
  - Creating a new branch.
  - Creating a PR.
  - Review flow.
- Using GitHub pages to publish a sample project.
- Good housekeeping:
  - Regularly update local repos so that they are in sync with their remote counterparts. This includes pulling remote changes to your local repo, and installing package updates (e.g. with `npm install` or `yarn`). Always do this before you start working on a local repo.
  - Use `.gitignore` to ignore all the stuff you don't want to commit. Examples include dependencies, dev source files, and OS-level admin files like `.DS_Store`.
  - Delete branches you have finished with.
- Handling merge conflicts.

Resources:

- Git and GitHub intro, Scrimba *Course Partner*
- Git and GitHub

Extensions modules

These "extension" modules constitute useful additional skills to learn as web developers start to expand their knowledge and develop specialisms.

1. Transform & animate CSS

Animations are a vital part of a good user experience. Subtle usage can make page designs more interesting and appealing, and also enhance usability and perceived performance.

Learning outcomes:

- Understand why CSS transforms and animation are needed.
- A caveat — overuse can negatively affect usability and accessibility.
- Common transforms — scaling, rotation, and translation.
- 3D transforms, and how 3D positioning/perspective is handled on the web.
- Transitions.
- Animations.

Resources:

- Using CSS transforms
- Using CSS transitions
- Using CSS animations

2. Custom JS objects

Having a deeper knowledge of how JavaScript objects work is useful as you build confidence with web development, start to build more complex apps, and create your own libraries.

Learning outcomes:

- Object prototypes.
- JavaScript class syntax.
- Defining a constructor.
- Defining properties and methods.
- Defining static properties and methods.

Resources:

- JavaScript object basics
- Object prototypes
- Object-oriented programming
- Classes in JavaScript

3. Web APIs

This module covers common aspects of three of the most common classes of Web APIs that we haven't previously covered in any kind of detail, providing a useful grounding for those who want to go deeper into browser API usage.

General notes:

- This module does not attempt to exhaustively cover the full spectrum of APIs available in web browsers.
- Rather, it provides you with enough information to understand Web APIs in general, and be able to pick up new ones via your own research.

General resources:

- Client-side web APIs

3.1 Video and audio APIs

Learning outcomes:

- General concepts:
  - Understand the different types of video and audio formats.
  - Understand codecs.
  - Understand key functionality associated with audio and video — play, pause, stop, seeking backward and forward, duration, and current time.

  Notes:

  - This set of conformance criteria does not expect a successful student to understand all of the web platform's video and audio-related APIs in detail. There is a lot of functionality in this category, and learning it all upfront would not be practical or particularly useful. Some of the functionality is for niche use cases, and students are encouraged to learn more as part of their constant learning, or when the need arises.
  - For now, you are expected to understand the concepts behind video and audio on the web, the basic core API functionality, and the purpose of some of the more advanced APIs.

- Using the `HTMLMediaElement` API to build a basic custom media player:
  - Understand why you'd do this — your target audience might have specific needs not addressed by the browser defaults.
  - One good example is that some browser default controls are not very keyboard accessible.
  - Another is that you might just want a consistent UI design across browsers.
- Using media streams/`getUserMedia()` to capture video and audio from a local device.
- Handling common errors related to media delivery:
  - Using `<source>` elements to handle multiple formats.
  - Using the correct MIME type.
  - Showing fallback content if the media is not supported.

- Understand the purpose of other video and audio APIs, including the Web Audio API, Media Stream Recording API, and Media Source Extensions API.

Resources:

- Video and audio APIs
- Audio and video delivery
- Audio and video manipulation
- Media type and format guide: image, audio, and video content

3.2 Graphics/animation APIs

Learning outcomes:

- Using timers and `requestAnimationFrame()` to set up animation loops:
  - Basic syntax and usage.
  - Understand why `requestAnimationFrame()` is an improvement over what came before it.
  - Common use cases for animation loops, for example, decorative animations and games.
- Web Animations API:
  - Basic syntax and usage.
  - Understand how the Web Animations API relates to CSS animation properties, and when should you use each one.
  - Common use cases.
- Canvas:
  - Understand conceptually what the `<canvas>` element and associated APIs enable.
  - Basic syntax and usage of the 2D Canvas API.
  - Looping `<canvas>` updates to create a simple animation or game.

Resources:

- Drawing graphics
- Web Animations API
- Canvas tutorial

3.3 Client-side storage

> **Notes**:
>
> The main items of importance to understand in this set of conformance criteria are the general concepts, using Web Storage for simple client-side storage tasks, and how cookies are used in positive and negative ways.

Learning outcomes:

- Understand the concepts of client-side storage:
  - Know the common client-side data storage mechanisms — Web Storage API, cookies, Cache API, and the IndexedDB API.
  - Key use cases — maintaining state across reloads, persisting login and user personalization data, and local/offline working.
  - Understand the negative patterns associated with client-side storage — for example using cookies for tracking/fingerprinting.
- Using cookies to store arbitrary data, normally controlled by HTTP headers.
- Using Web Storage for simple key-value pair storage, controlled by JavaScript.
- Using IndexedDB:
  - Complete client-side transactional database system.
  - Complex, and rarely used directly. You'd be more likely to use a library such as dexie.js.
- Using Cache/Service Workers:
  - Understand the basic ideas behind their usage in Progressive Web Apps (PWAs), and the fundamental use case of making a site work offline.

Notes:

IndexedDB and the Cache API (commonly used with Service Workers) are complex, and constitute huge topics. Exhaustively understanding them at this stage is not necessary, although we would suggest that you gain an understanding of the basics behind how they work.

Resources:

- What is localStorage?, Scrimba *Course Partner*
- Client-side storage
- Using HTTP cookies
- What is a progressive web app?

4. Performance

Performance centers around making your websites as fast as possible, both in real terms (for example small file sizes, quicker loading), and in terms of how performance is perceived (for example indicating progress and getting initial content to a usable state as fast as possible, even if all the content is not yet loaded).

4.1 Performance basics

Learning outcomes:

- Understand the concepts of real performance and perceived performance and the difference between the two.

- Understand key performance concepts:
  - Source order.
  - Critical path.
  - Latency.
  - How a browser renders pages.
- Understand how performance impacts sustainability — good performance can have a positive impact on the planet by reducing energy usage and bandwidth consumption:
  - Energy efficiency:
    - Code performance (see 4.5 JavaScript and performance).
    - Static power draw (idle state).
  - Hardware efficiency (repairability/utilization).
  - Demand efficiency:
    - Spatial (where do you perform computations).
    - Temporal (when do you perform computations).

Resources:

- The “why” of web performance
- What is web performance?
- Perceived performance

4.2 Improving page rendering

Learning outcomes:

- How to reduce page loading times:
  - Use optimal media formats and compression.
  - Remove unnecessary audio from muted video.
  - Use video preload attribute to reduce upfront downloads.
  - Considering using adaptive streaming.
  - Reducing media loading jank with width and height attributes.
  - Be careful about font choices — keep file sizes as small as possible, for example by including only the glyphs, variants, and weights you need.
- How to improve “time to usable”:
  - Minimize initial load by showing only important content initially. Make important interactive features interactive as soon as possible.
  - Additional data and resources can be loaded in the background as users are using the page.
  - Use lazy loading for images and other resources that are not immediately needed.
- How to improve the perceived performance of features:

- Use animations to transition between states rather than making the user wait for the end state.
- Use loading spinners and progress bars to indicate progress, so the user feels like something is happening.
- Use events wisely, e.g. trigger actions on `keydown` rather than waiting for `keyup`.

Resources:

- Perceived performance
- Multimedia: Images
- Multimedia: Video

4.3 Measuring performance

Learning outcomes:

- Understand key metrics for measuring performance, for example first contentful paint, speed index, total blocking time, bounce rate, unique users/page views.
- How to use common performance measurement tools:
  - Google Lighthouse.
  - Pagespeed Insights.
  - WebPageTest.
  - Google Analytics.
  - Browser DevTools.
- How to use Performance Web APIs to create your own performance measurement tools:
  - Performance Timeline API.
  - Navigation Timing API.
  - User Timing API.
  - Resource Timing API.

Resources:

- Measuring performance
- Performance APIs reference
- Google Lighthouse > Performance
- Relevant Firefox DevTools: Network Monitor and Performance Monitor, Firefox Source Docs
- Relevant Chrome DevTools: Inspect network activity (2019) and Analyze runtime performance (2017), developer.chrome.com

4.4 CSS and performance

Learning outcomes:

- Understand techniques for improving CSS performance:

- Only load when needed; optimize with media queries.
- Minimize animation, and force animation on the GPU.
- Minimize repaints.
- Use `will-change` and `contain` appropriately.

Resources:

- CSS performance optimization

4.5 JavaScript and performance

Learning outcomes:

- Understand techniques for improving JavaScript performance:
  - Reduce the amount of JavaScript you use.
  - Only load JavaScript when needed, and remove unused code.
  - Use deferred/async JavaScript appropriately.
  - Compressing, packing, and splitting JavaScript.

Resources:

- JavaScript performance

5. Security and privacy

It is vital to have an understanding of how you can and should protect your data and your user's data from would-be attackers who may try to steal it. This module covers both hardening websites to make it more difficult to steal data, and collecting user data in a respectful way that avoids tracking them or sharing it with unsuitable third parties.

General resources:

- Privacy on the web
- Learn Privacy, web.dev (2023)

5.1 Security and privacy basics

Notes:

- Conforming to all of the criteria in this module is not going to result in a student being a qualified security engineer, but equally it is important for web developers to understand the basics of web security and privacy.
- It is also important for students to understand that a lot of security issues are caused by problems with server-side code, or a combination of client-side and server-side code. A lot of code should present very few security risks, provided the browser is doing its job properly.

Learning outcomes:

- Understand the difference between security and privacy.
- Understand the general HTTP model from a high-level.
- Learn what HTTPS is, and why it is important.
- Same-origin security:
  - Why this is fundamental to the web.
  - Ways of working around it safely, such as Cross-Origin Resource Sharing (CORS).
- How cookies are stored, and their security and privacy implications, such as tracking.
- Learn about situations where security issues generally occur:
  - When asking users to provide sensitive data (such as passwords or credit card data) and transmitting it to a server.
  - When requesting data from a server.
  - When transmitting data between servers (for example, if a server requests data from a web service).
  - When preserving user state by setting a cookie or other mechanisms.
- Learn about common security threats and how to mitigate them:
  - Cross-site scripting (XSS).
  - Cross-site request forgery (CSRF).
  - Clickjacking.
  - Denial of service (DoS).
- Understand the purpose of other important technologies, such as:
  - Content Security Policy (CSP).
  - Permissions-Policy.
  - The web model for user activation of “powerful features” (aka transient activation).

Resources:

- Security on the web
- Website security
- Privacy on the web

5.2 Data protection laws

Learning outcomes:

- Understand fundamental concepts related to user privacy:
  - Personally identifiable information (PII).
  - Confidentiality.
  - Tracking.

    - Fingerprinting.
- Be aware of regional privacy laws, for example:
    - General Data Protection Regulation (GDPR) (EU).
    - Data Protection Act 2018 (UK), gov.uk.
    - California Consumer Privacy Act (2018) (US, CA), ca.gov.
    - Children's Online Privacy Protection Rule (COPPA) (US), ftc.gov.
- Understand how to comply with such laws, in terms of practical implementation.

Notes:

Conforming to the above criteria does not require students to become legal experts in privacy laws, but they should understand the implications of these laws, and how that affects their work.

Resources:

- Complete guide to GDPR compliance, gdpr.eu

6. Testing

Any codebase past a certain level of complexity needs to have a system of tests associated with it, to make sure that as new code is added, the codebase continues to function correctly and performantly, and continues to meet the users' needs. This module lists the fundamentals that you should start with.

6.1 General testing fundamentals

Learning outcomes:

- Understand the overall purposes of testing:
    - To make sure that the functionality of an app works (and continues to work) for the intended target audience to a chosen level of quality.
    - To make sure that the code and functionality meet chosen standards of quality and conform to set conformance criteria, benchmarks, or guidelines.
- The purpose of common test types:
    - Functional testing (related to unit testing): Checking that the features and functions of the web app are working as expected, e.g. user interactions, forms, navigation, links, and other core functionalities.
    - Usability testing: Evaluating the user-friendliness of the web app. This involves assessing how easy it is for users to navigate the app, complete tasks, and achieve their goals.
    - Compatibility testing (aka cross-browser testing): Ensuring that the web app functions correctly across different browsers, operating systems, and devices.
    - Performance testing: Measuring the responsiveness, speed, scalability, and stability of the web app under different workloads (see also Extension 4: Performance).

- Security Testing: Identifying vulnerabilities and weaknesses in the web app's security mechanisms (see also Extension 5 Security and privacy).
- Accessibility Testing: Ensuring that the web app can be used by people with disabilities, conforming to accessibility guidelines such as the Web Content Accessibility Guidelines (see also Accessibility).

6.2 Functional and compat testing

Learning outcomes:

- Understand that the two are closely related — you will want your web app functionality to work for a target range of users, across a range of target browsers/devices. In addition, such testing can largely be automated using available tools.
- Some accessibility testing can be automated, for example, "do the images all have alt text?".
- A typical process for performing automated tests:
  - Identify who the target audience groups are, and what browsers they are using.
  - Identify the items of functionality that should be tested. These can range from visual ("does the layout look like it should?"), to user-facing functions ("does the search box return a result?"), to lower-level code ("does function `x()` return the expected result?").
  - Some tests will also test that integrated modules are working correctly with other parts of the codebase (integration testing).
  - Write tests to test this functionality, using an appropriate tool set.
  - Run the tests regularly (for example after each commit to the repo) to see if any tests fail.
  - Document the test results.
  - Fix the code and rerun the tests to make sure they pass, and ensure that new errors have not been introduced by the code updates (regression testing).
- Understand the typical toolset used to run automated tests:
  - Use a combination of physical devices and virtual machines (such as VirtualBox) to make the different browser and device combinations available.
  - Use Selenium/WebDriver to run specific tests on installed browsers and return results, alerting you to failures in browsers as they appear. Packages such as webdriver.io and nightwatch.js can be used to control WebDriver via Node.js code (other platform integrations also exist).
  - Playwright is a popular alternative.
  - Write your own functional tests right inside your codebase, using a library like Jest or Mocha.
  - Use a cloud-based service to do automated cross-browser functional testing, such as LambdaTest, Sauce Labs, BrowserStack, or TestingBot. Most of these services have APIs available, allowing you to run tests from a platform like Node.js.
- Understand how to integrate testing with GitHub using continuous integration (CI) tools:
  - Understand the utility of this — you can create a setup to automatically run your test suite each time you commit a change to the codebase.

- Use tools such as CircleCI or Travis CI to do this.

Resources:

- Cross-browser testing

6.3 Usability testing

Learning outcomes:

- Understand how usability testing differs from functional testing — whereas functional testing can largely be automated (it largely looks at whether some code returns an expected result), usability testing tends to require manual testing with real test subjects (e.g. "does the page allow me to find the most important functionality and is it intuitive to use?").
- Some accessibility testing should be done manually as a subset of usability testing — for example, you might be able to tab to the form elements to fill them in, but does the form make sense to an assistive technology user? Can they tell what information it is asking them for?
- A typical process for usability testing:
  - Identify the aspects of the app functionality you want to test (these can be in the form of hypotheses or problem statements you want to test), and what target audience segments would be most appropriate to test them.
  - Recruit some test subjects for you to do the testing with. 5-10 successful sessions should be enough to start to identify recurring usability problems.
  - Write scripts for your test subjects to follow that will test the functionality (for example "navigate to the product page, search for a widget, and add one to your shopping cart").
  - Run the usability tests with your test subjects and record the sessions.
  - Analyze the sessions, and identify key improvements to make.
  - Make the improvements.
  - Measure whether the improvements fixed the problem (for example by looking for changes in key metrics, or by rerunning the usability test with a new cohort of test subjects).
- A typical toolset for running a usability test:
  - Clear instructions are needed: Explain the purpose of the test to the test subject. Instruct them to think aloud while performing the tasks. Observe their interactions and take notes on any issues or difficulties they encounter. Encourage participants to provide honest feedback.
  - Video conferencing software such as Zoom, which includes screen-sharing functionality so you can observe what the test subjects are doing, and the ability to record the session so you can review it later.
  - Eye-tracking software can be useful, to allow you to see what the user looks at on the website as they try to follow your script.

Resources:

- Usability testing 101, nngroup.com (2019)

7. JavaScript frameworks

JavaScript frameworks are commonly used by companies to build web applications. It is therefore beneficial to learn about popular frameworks and use cases (as listed below) for better employment prospects.

Notes:

- Conforming to this set of criteria does not involve becoming a master of a particular framework and learning everything it has to offer. To do so can be limiting — one framework might be the most popular choice now, but in five years everything could change.
- Instead, we want to encourage students to gain a solid understanding of the JavaScript fundamentals frameworks are built on top of (see JavaScript fundamentals), and learn the common features and patterns that frameworks use. This approach is much more flexible and future proof.
- Students are strongly encouraged to start building apps with 2 or 3 popular frameworks so they understand how they do things in general but also the differences between them.
- Advanced and/or framework-specific features can be learned as part of the student's constant learning.

Learning outcomes:

- Understand how to start using frameworks:
  - Link to the relevant JavaScript files, either locally or on a CDN (not usually recommended).
  - Use a dependency manager such as npm to add it to your project.
  - Install a CLI such as Vite and use it to generate a skeleton framework-based application that you can modify. This is the easiest way to get started.
- The fundamentals of a typical framework. The overall architecture will differ, but most frameworks include the following in one form or another, which you should learn about:
  - Component systems:
    - Props.
    - Data binding.
    - Event handlers.
    - Dependency injection.
    - Component lifecycle system.
  - Templating language:
    - Usually a domain-specific language (DSL) such as JSX or handlebars.
    - Includes features such as filtering and conditional rendering.
  - Rendering system (for example, virtual or incremental DOM).
  - State management.

    - Routing.
    - Styling system to apply CSS to your components.
    - Dependency management.
    - Extension system.
- Understand the practicalities of working with a framework. Most framework-related CLIs or development environments include:
    - A local testing server that refreshes on file save.
    - A useful set of extensions for popular code editors (for example VSCode) that provide features like linting and code completion for framework-specific code.
    - An integrated system for writing tests (see also Extension 6 Testing).
    - Facilitation for deployment to a remote server (most deployment tools such as Netlify and Vercel have integrations/tooling for working with both GitHub and popular frameworks).

Resources:

- Learn React, Scrimba *Course partner*
- Framework main features
- Getting started with React
- Getting started with Vue
- Getting started with Ember
- Getting started with Svelte
- Getting started with Angular

8. CSS tooling

Tooling is not just confined to JavaScript frameworks. There are also common CSS tooling types that you'll encounter on your learning journey.

8.1 CSS frameworks

> **Notes**:
>
> The aim here is not to have an exhaustive understanding of any one framework, but rather to understand the pros and cons, what a CSS framework can do, and what general usage looks like.

Learning outcomes:

- Understand the value that CSS frameworks bring — consistency, organization, ready-made components and style guidelines, and built-in best practices.

- Understand the problems with CSS frameworks — they are often overkill (in terms of complexity and file size), new syntax to learn, your sites will start to look like everyone else's, and they can be hard to override if you want to customize them.
- Understand the basics of using popular CSS frameworks such as Bootstrap, Foundation, and Tailwind.
- Understand how they integrate into a web project.
- Understand how to weigh up the burden of adopting a CSS framework (e.g. handling integration and initial learning curve) versus the advantages (how much they can speed up development once you are familiar with them).

8.2 CSS preprocessors

Notes:

The aim here is not to have an exhaustive understanding of any one preprocessor, but rather to understand the pros and cons, what a CSS preprocessor can do, and what general usage looks like.

Learning outcomes:

- Understanding the value of CSS preprocessors — bringing features to CSS that are not available natively (such as loops or if/else structures), and speeding up the writing of your code.
- Understand the problems with CSS preprocessors — for example, they often require learning new syntax.
- Understand the basics of using popular CSS preprocessors such as Sass and PostCSS.
- Understand how they integrate into a web project.
- Understand how to weigh up the burden of adopting a CSS preprocessor (e.g. handling integration and initial learning curve) versus the advantages (how much they can speed up writing CSS once you are familiar with them).

9. Other tooling types

Notes:

This set of criteria doesn't require having an in-depth knowledge of the tools listed below; instead, students are encouraged to learn the basic concepts of why and how they are used (and where in a web app project), test them out, and play with usage examples.

Learning outcomes — understand the purpose and basic usage of other common tooling types you may be required to use in a web project:

- Linters/formatters, for example, ESLint and Prettier:
  - Detecting potential bugs, and how linters can automatically fix them.

    - The need to have coding standards, and how these tools can easily automate conformance to those standards.
    - Integrating them as part of your workflow — as a code editor extension, as part of your test runner, or as part of a CI run.
- Package managers, for example, npm and yarn:
    - Using these as a way to manage project dependencies as a codebase becomes more complex.
    - Finding suitable packages to solve your problems from registries and installing them.
    - Using scripts (for example npm scripts) to make deployment and testing easier.
- Build tools/bundlers, for example, Rollup.js, Parcel, and Vite.js:
    - Using these to manage dependencies, and organize, optimize, and minify your code into build files for improved performance.
- Deployment tools, for example, Simple FTP clients, Netlify and Vercel:
    - Using these to deploy your applications.
    - Integrating tooling with GitHub to provide automatic test deployments on push.
    - Handling custom domains, serverless functions, and form submissions.
    - Handling access control and integration with identity providers such as Okta and Auth0.
    - Integrations with popular frameworks and app platforms.

Resources:

- Understanding client-side web development tools