Client-Side Web Development

Joel Ross and David Stearns

August 16, 2017

Contents

About this Book

This book covers the the skills and techniques necessary for creating sophisticated and accessible interactive web applications. It focuses on the client-side languages, tools, and libraries that professionals use to build the web sites you use every day. It assumes a basic background in computer programming (e.g., one course in Java, and some concepts from the technical foundations of informatics). These materials were developed for the **INFO 343: Client-Side Web Development** course taught at the University of Washington Information School; however they have been structured to be an online resource for anyone who wishes to learn modern web programming techniques.

This book is currently in **alpha** status. Visit us on GitHub to contribute improvements.

Chapter 1

Getting Setup

This course will cover a wide variety of tools and techniques used in modern web development, including different software programs that are used to write, manage, and execute the code for your web application. This chapter explains how to install and use some of the software you will need to utilize.

Note that iSchool lab machines should have all appropriate software already installed and ready to use.

1.1   Web Browser

The first thing you'll need is a web browser for viewing the web pages you make! We recommend you install and utilize **Chrome**, which comes with an effective set of built-in developer tools that will be especially useful in this class.

- You can access the Chrome Developer tools by selecting **View > Developer > Developer Tools** from Chrome's main menu (`Cmd+Option+I` on a Mac, `Ctrl+Shift+I` on Windows). You will almost always want to have these tools open when doing web development, particularly when including interactivity via JavaScript.

Other modern browsers such as **Firefox** or **Microsoft Edge** will also function in this class and include their own versions of the required development tools. Note that different browsers may and will render code in different ways which will be discussed extensively throughout the course.

1.2   Code Editors

In order to write web code, you need somewhere to write it. There are a variety of code editors and IDEs (Integrated Development Environments) that are specialized for web development, providing syntax highlighting, code completion, and other useful functionality. There are lots of different code editors out there, all of which have slightly different appearances and features. You only need to download and use one of the following programs (we recommend **Visual Studio Code** as a default), but feel free to try out different ones to find something you like (and then evangelize about it to your friends!)

Visual Studio Code

Visual Studio Code (or VS Code; not to be confused with Visual Studio) is a free, open-source editor developed by Microsoft—yes, really. It focuses on web programming and JavaScript, though also supports many other languages and provides a number of community-built extensions for adding even more features. Although fairly new, it is updated regularly and has has become my main editors for programming. VS Code is actually a stand-alone web application, so it's written in the same HTML, CSS, and JavaScript you'll learn in this course!

To install VS Code, follow the above link and Click the "Download" button to download the installer (e.g, `.exe`) file, then double-click on that to install the application.

Once you've installed VS Code, the trick to using it effectively is to get comfortable with the Command Palette. If you hit `Cmd+Shift+P`, VS Code will open a small window where you can search for whatever you want the editor to do. For example, if you type in `markdown` you can get list of commands related to Markdown files (including the ability to open up a preview). The `Format Code` option is particularly useful.

For more information about using VS Code, see the documentation, which includes videos if you find them useful. The documentation for programming in HTML, CSS, and especially JavaScript also contain lots of tips and tricks.

Atom

Atom is a text editor built by the folks at GitHub and has been gaining in popularity. It is very similar to VS Code in terms of features, but has a somewhat different interface and community. It has a similar *command-palette* to VS Code, and is arguably even nicer about editing Markdown specifically. The document you are reading was authored in Atom.

Brackets

Brackets is a coding editor authored by Adobe specialized for client-side web developers. It has some intriguing features that are not yet in Visual Studio Code, as well as possibly the nicest interface of this list.

Sublime Text

Sublime Text is a very popular text editor with excellent defaults and a variety of available extensions (though you'll need to manage and install extensions to achieve the functionality offered by other editors out of the box). While the software can be used for free, every 20 or so saves it will prompt you to purchase the full version. This is my application of choice for when I just write to write a plain text file.

1.3 Bash (Command Line)

Many of the software tools used in professional web development are used on the **command-line**: a text-based interface for controlling your computer. While the command-line is harder to learn and figure out, it is particularly effective for doing web development. Command-line automation is powerful and efficient enough to handle the dozens of repeated tasks across hundreds of different source files (split across multiple computers) commonly found in web programming. You will need to be comfortable using the command-line in order to utilize the software for this course.

While there are multiple different **command shells** (command line interfaces), this course is based on the Bash shell, which provides a particular common set of commands common to Mac and Linux machines.

On a Mac you'll want to use the built-in app called **Terminal**. You can open it by searching via Spotlight (hit Cmd (⌘) and Spacebar together, type in "terminal", then select the app to open it), or by finding it in the Applications/Utilities folder.

On Windows, we recommend using **Git Bash**, which you should install along with `git` (see below). Open this program to open the command-shell.

- Note that Windows does come with its own command-prompt, called the *DOS Prompt*, but it has a different set of commands and features. *Powershell* is a more powerful version of the DOS prompt if you really want to get into the Windows Management Framework. But Bash is more common in open-source programming like we'll be doing, and so we will be focusing on that set of commands.

- Alternatively, the 64-bit Windows 10 Anniversary update (August 2016) *does* include a beta version of an integrated Bash shell. You can access this by enabling the subsystem for Linux and then running `bash` in the command prompt. This is currently (May 2017) "beta" technology, but will suffice for our purposes if you can get it running.

This course expects you to already be familiar with basic command-line usage. For review, see The Command Line in the *INFO 201* course reader.

1.4   Git and GitHub

Professional web development involves many different people working on many different files. **git** is a collaborative version control system that provides a set of commands that allow you to manage changes to written code, particularly when collaborating with other programmers.

You will need to download and install the software. If you are on a Mac, `git` should already be installed. If you are using a Windows machine, then installing `git` will also install Git Bash, a command shell (described avove).

Note that `git` is a command-line application: you can test that it is installed by running the command:

```
git --version
```

While `git` is the software used to manage versions of code, **GitHub** is a website that is used to store copies of computer code that are being managed with `git` (think "Imgur for code").

In order to use GitHub, you'll need to create a free GitHub account, if you don't already have one. You should register a username that is identifiable as you (e.g., based on your name or your UW NetID). This will make it easier for others to determine out who contributed what code, rather than needing to figure out who 'LeetDesigner2099' is. This can be the start of a professional account you may use for the rest of your career!

- Note that you can have `git` save your GitHub password on your local machine so you don't have to type it repeatedly. See Authenticating with GitHub from Git.

This course expects you to already be familiar with utilizing Git and GitHub. For review, see Git and GitHub and Git Branches and Collaboration in the *INFO 201* course reader. Note that students in the INFO 343 course will be using GitHub and Pull Requests to turn in programming assignments.

1.5   Node and npm

Node.js (commonly just "Node") is a a command-line runtime environment for the JavaScript programming language—that is, a program that is used to *interpret* and *execute* programming instructions written in JavaScript. Although client-side development usually involves running JavaScript in the browser (see Chapter: JavaScript), Node provides a platform for installing and running a wide variety of "helper" programs that are frequently used in web development.

To install Node, visit the download page and select the installer for your operating system (you probably want the `.msi` for Windows and the `.pkg` for Mac). For this course you will want to install the **latest version** of Node (6.10+), so you should update it if you haven't in a while. Node is a command-line application, so you can test that it is installed and available to your command shell (e.g., Terminal or Git Bash) with:

```
node --version
```

Installing Node also installs an additional command-line program called ***npm***. *npm* is a **package manager**, or a program used to "manage" other programs— think of it as a command-line "app store" for developer tools and libraries. *npm* is the most common way of installing and running a large number of tools used in professional web development. At the time of writing, the *npm* "registry" lists around 500,000 different packages.

Managing packages with npm

You can use the `npm` program to download and install command-line programs by name:

```
npm install -g PACKAGE-NAME
```

For example, you can install the *live-server* utility (a simple program that runs a local web server and will automatically "refresh" the browser when your code changes) using

```
npm install -g live-server
```

- Once the program is done installing, you can run it from the command-line by using the command `live-server`. This program will serve all of the content from the current directory. See Chapter 2 for details.

Importantly, note the included **-g** option. This tells *npm* that the package should be installed **globally**, making it available across the entire computer, rather than just from a particular folder. Because you want to be able to use a command-line utility like *live-server* from any folder (e.g., for any project), command-line utilities are always installed globally with the `-g` option.

It is also possible to omit that option and install a package *locally*. For example:

```
npm install lodash
```

Will download the `lodash` code library (a set of useful JavaScript functions). This package will be placed into a new folder *in the current project directory* called `node_modules/`, and can be imported and used in the current directory's code. (It's called a local install because the package is only available to the "local" project). You will of course need to install local packages once per project.

Because node packages can be very large, and projects can have lots of them, you want to be sure to **not** commit the `node_modules/` folder to version control. Make sure that the folder is listed in your `.gitignore` file!

1.5.1 `package.json`

As projects become large, it is common for them to build up many *dependencies*: packages that must be installed in order for the program to work. In other words, there needs to be a certain set of packages in the project's `node_modules/` folder. *npm* is able to keep track of these dependencies by recording them in a specialized file called `package.json` that can be placed inside the project directory. A `package.json` file is a text file containing a JSON list of information about your project. For example:

```json
{
  "name": "example",
  "version": "1.0.0",
  "description": "A project with an example package.json",
  "main": "index.js",
  "scripts": {
    "test": "jest"
  },
  "author": "Joel Ross",
  "license": "ISC",
  "dependencies": {
    "lodash": "^4.17.4",
    "moment": "^2.18.1"
    },
  "devDependencies": {
    "html-validator": "^2.2.2"
  }
}
```

(You can create one of these files by using the command `npm init` in the current project directory, and then following the instructions to fill in the fields).

Notice that there are two packages listed under `"dependencies"`: `lodash` and `moment` (the `^4.17.4` indicates which version of `lodash`). You can use `npm` to

automatically install all of packages listed under `"dependencies"` (as well as `"devDependencies"`) using the command:

```
npm install
```

Thus using `npm install` without any arguments means "install all of the requirements that have been listed for this project". This is a good first step *any time* you download a project or checkout a repository from GitHub.

When installing specific packages, you can have *npm* add them to the dependencies list by using the `--save` option:

```
npm install --save lodash
```

will install `lodash` locally, and list it in the `package.json` file as a dependency.

Similarly, the `--save-dev` option will instead save the package in the `"devDependencies"` list, which are dependencies needed only for development (writing the program's code) and not for execution (running the program).

You can uninstall packages using `npm uninstall`, or can remove packages from the dependencies lists simply by editing the `package.json` file (e.g., with VS Code).

To sum up, you will use three commands with `npm` to install packages:

1. `npm install -g PACKAGE-NAME` to *globally* install command-line programs
2. `npm install` to *locally* install all of the dependencies for a project you check out
3. `npm install --save PACKAGE-NAME` to *locally* install a new code package and record it in the `package.json` file.

While *npm* is the most popular package maanger (and the one utilized in this course), there are others as well. For example, **Yarn** is a package manager created by Facebook that is compatible with *npm* and is quickly growing in popularity.

Resources

Links to the recommended software are collected here for easy access:

- Chrome
- git (and Git Bash) - GitHub (sign up)
  - optional: Bash on Windows
- Visual Studio Code
- Node.js (and npm)
  - npm documentation

Chapter 2

Client-Side Development

Web development is the process of implementing (programming) web sites and applications that users can access over the internet. However, the internet is a network involving *many* different computers all communicating with one another. These computers can be divided into two different groups: **servers** store ("host") content and provide ("serve") it to other computers, while **clients** request that content and then present it to the human users.

Consider the process of viewing a basic web page, such as the Wikipedia entry on Informatics. In order to visit this page, the user types the web address (`https://en.wikipedia.org/wiki/Informatics`) into the URL bar, or clicks on a link to go to the page. In either case, user's computer is the **client**, and their browser takes that address or link and uses it to create an **HTTP Request**—a *request* for data sent following the **H**yper**T**ext **T**ransfer **P**rotocol. This request is like a letter asking for information, and is sent to a different computer: the **web server** that contains that information.
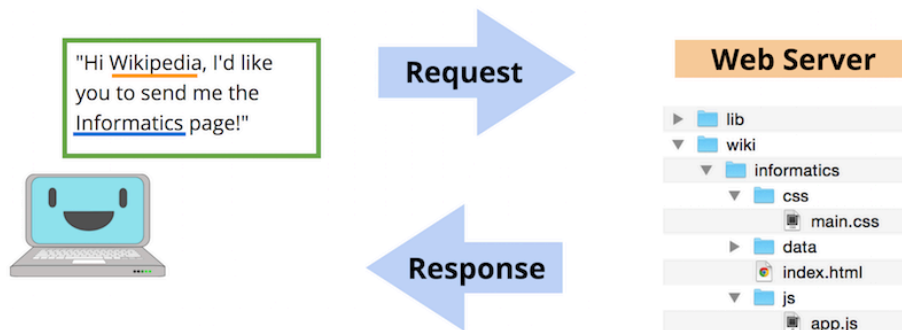

Figure 2.1: A diagram of client/server communication.

The web server will receive this request, and based on its content (e.g., where it was sent to) will decide what information to send as a **response** to the client. In general, this response will be made up of lots of different files: the text content of the web page, styling information (font, color) for how it should look, instructions for respond to user interaction (button clicks), images or other assets to show, and so forth.

The client's web browser will then take all of these different files in the response and use them to *render* the web page for the user to see: it wil determine what text to show, what font and color to make that text, where to put the images, and is ready to do something else when the user clicks on one of those images. Indeed, a web browser is just a computer program that is able to send HTTP requests on behalf of the user, and then render the resulting response.

Given this interaction, **client-side web development** involves implementing programs (writing code) that is interpreted by the *browser*, and so is executed by the client. It is authoring the code that is sent in the server's response. This code specifies how websites should appear and how the user should interact with them. On the other hand, **server-side web development** involves implementing programs that the *server* uses to determine which client-side code is delivered. As an example, a server-side program contains the logic to determine which cat picture should be sent along with the request, while a client-side program contains the logic about where and how that picture should appear on the page.

This course focuses on *client-side web development*, or developing programs that are executed by the browser (generally as a response to a web server request). While we will cover how client-side programs can interact with a server, many of the concepts discussed here can also be run inside a browser without relying on an external server (called "running locally").

2.1   Client-Side File Types

It is the web browser's job to interpret and render the source code files sent by a server as part of an HTTP response. As a client-side web programmer, your task is to write this source code for the browser to interpret. There are multiple different types of source code files, including:

- `.html` files containing code written in HTML (HyperText Markup Language). This code will specify the textual and *semantic* content of the web page. See the chapter HTML Fundamentals for details on HTML.

- `.css` files containing code written in CSS (Cascading Style Sheets). This code is used to specify styling and *visual appearance* properties (e.g., color and font) for the HTML content. See the chapter CSS Fundamentals for details on CSS.

- `.js` files containing code written in JavaScript. This code is used to specify *interactive behaviors* that the website will perform—for example, what

should change when the user clicks a button. Note that JavaScript code are "programs" that sent over by the web server as part of the response, but are *executed* on the client's computer. See the chapter JavaScript Fundamentals for details on JavaScript.

HTTP responses may also include additional **asset** files, such as images (`.png`, `.jpg`, `.gif`, etc), fonts, video or music files, etc.

2.2 HTTP Requests and Servers

Modern web browsers are able to *render* (interpret and display) all of these types of files, combining them together into the modern, interactive web pages you use every day. In fact, you can open up almost any file inside a web browser, such as by right-clicking on the file and selecting "Open With", or dragging the file into the browser program. HTML files act as the basis for web pages, so you can open a `.html` file inside your web browser by double-clicking on it (the same way you would open a `.docx` file in MS Word):
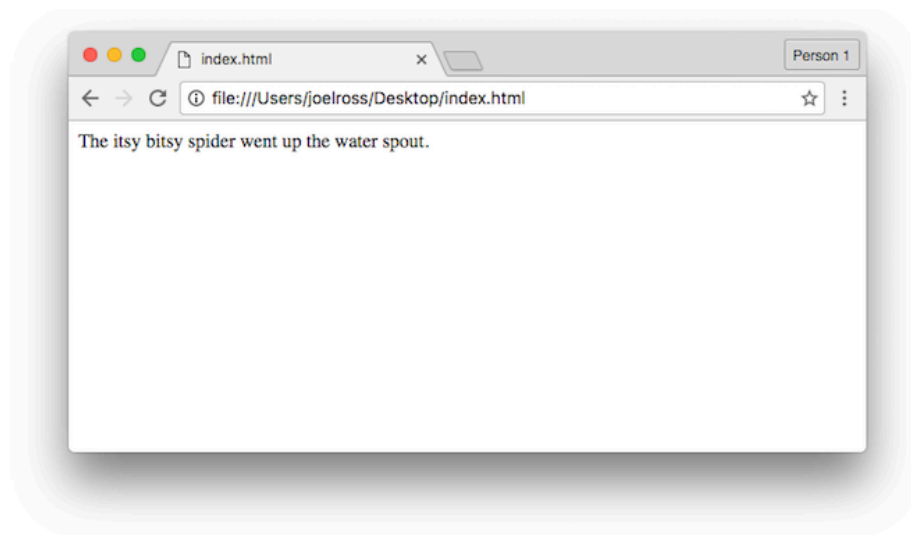

Figure 2.2: An very simple HTML file. See Chapter 3 for source code.

Consider the URL bar in the above browser. The URL (Unform Resource Locator) is actually a specialized version of a **URI (Uniform Resource Identifier)**. URIs act a lot like the *address* on a postal letter sent within a large organization such as a university: you indicate the business address as well as the department and the person, and will get a different response (and different data) from Alice in Accounting than from Sally in Sales.

- Note that the URI is the **identifier** (think: variable name) for the resource, while the **resource** is the actual *data* value (the file) that you want to access.

Like postal letter addresses, URIs have a very specific format used to direct the request to the right resource.
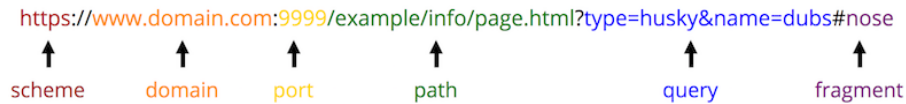
https://www.domain.com:9999/example/info/page.html?type=husky&name=dubs#nose

scheme   domain   port   path   query   fragment

Figure 2.3: The format (schema) of a URI.

The parts of this URI format include:

- `scheme` (also `protocol`): the "language" that the computer will use to send the request for the resource (file).

  In the example browser window above, the protocol is `file`, meaning that the computer is accessing the resource from the file system. When sending requests to web servers, you would use `https` (**s**ecure HTTP). *Don't use insecure `http`!*

- `domain`: the address of the web server to request information from. You can think of this as the recipient of the request letter.

  In the browser window example, there is no domain because the `file` protocol doesn't require it.

- `port` (*optional*): used to determine where to connect to the web server. By default, web requests use port 80, but some web servers accept connections on other ports—e.g., `8080`, `8000` and `3000` are all common on development servers, described below.

- `path`: which resource on that web server you wish to access. For the `file` protocol, this is the *absolute path* to the file. But even when using `https`, for many web servers, this will be the *relative path* to the file, starting from the "root" folder of that server (which may not be the computer's root folder). For example, if a server used `/Users/joelross/` as its root, then the *path* to the above HTML file would be `Desktop/index.html` (e.g., `https://domain/Desktop/index.html`).

  Important! If you don't specify a path, most web servers will serve the file names `index.html` file from that server's root folder (i.e., the path "defaults" to `index.html`). As such, this is the traditional name for the HTML file containing a websites home page.

  As in any program, you should always use **relative** paths in web programming, and these paths are frequently (but not always!) relative to the web server's *root folder*.

- **query** (*optional*): extra **parameters** (arguments) included in the request about what resource to access. The leading `?` is part of the query.

- **fragment** (*optional*): indicates which part ("fragment") of the resource to access. This is used for example to let the user "jump" to the middle of a web page. The leading # is part of the query.

Development Servers

As noted above, it is possible to request a `.html` file (open a web page) using the `file` protocol by simply opening that file directly in the browser. This works fine for testing most client-side programs. However, there are a few client-side interactions that for security reasons only work if a web page is requested from a web server (e.g., via the `http` protocol).

For this reason, it is recommended that you develop client-side web applications using a **local development web server**. This is a web server that you run from your own computer—your machine acts as a web server, and you use the browser to have your computer send a request *to itself* for the webpage. Think about mailing yourself a letter. Development web servers can help get around cross-origin request restrictions, as well as offer additional benefits to speed development—such as *automatically reloading the web browser when the source code changes*.

There are many different ways to run a simple development server from the command-line (such as using the Python `http.server` module). These servers, when started, will "serve" files using the current directory as the "root" folder. So again, if you start a server from `/Users/joelross`, you will be able to access the `Desktop/index.html` file at `http://127.0.0.1:port/Desktop/index.html` (which port will depend on which development server you use).

- The address `127.0.0.1` is the IP address for `localhost` which is the domain of your local machine (the "local host"). Most development servers, when started, will tell you the URL for the server's root directory.

- Most commonly, you will want to start the web server from the root directory of your *project*, so that the relative path `index.html` finds the file you expect.

- You can usually stop a command-line development server with the universal `ctrl-c` cancel command. Otherwise, you'll want to leave the server running in a background terminal as long as you are working on your project.

If you use the recommended **live-server** utility, it will open a web browser to the root folder and *automatically reload the page* whenever you **save** changes to a file in that folder. This will make your life much, much better.

Chapter 3

HTML Fundamentals

A webpage on the internet is simply a set of files that the browser *renders* (shows) in a particular way, allowing you to interact with it. The most basic way to control how a browser displays content (e.g., words, images, etc) is by *encoding* that content in HTML.

HTML (**H**yper**T**ext **M**arkup **L**anguage) is a language that is used to give meaning to otherwise plain text, which the browser can then use to determine how to display that text. HTML is not a programming language but rather a *markup language*): it adds additional details to information (like notes in the margin of a book), but doesn't contain any logic. HTML is a "hypertext" markup language because it was originally intended to mark up a document with hyperlinks, or links to other documents. In modern usage, HTML describes the **semantic meaning** of content: it marks what content is the a *heading*, what content is a *paragraph*, what content is a *definition*, what content is an *image*, what content is a *hyperlink*, and so forth.

- HTML serves a similar function to Markdown, but is much more expressive and powerful.

This chapter provides an overview and explanation of HTML's syntax (how to use it to annotate content). HTML's syntax is very simple, and generally only takes someone a few days to learn—though using it effectively can require more practice.

3.1   HTML Elements

HTML content is normally written in `.html` files. By using the `.html` extension, your editor, computer, and browser should automatically know that this file will contain content marked up in HTML.

As mentioned in Chapter 2, most web servers will by default serve a file named **index.html**, and so that filename is traditionally used for a website's home page.

As with all programming languages, `.html` files are really just plain text files with a special extension, so can be created in any text editor. However, using a coding editor such as VS Code provides additional helpful features that can speed up your development process.

HTML files contain the **content** of your web page: the text that you want to show on the page. This content is then annotated (marked up) by surrounding it with **tags**:

Figure 3.1: Basic syntax for an HTML element.

The **opening/start tag** comes before the content and tell the computer "I'm about to give you content with some meaning", while the **closing/end tag** comes after the content to tell the computer "I'm done giving content with that meaning." For example, the <h1> tag represents a top-level heading (equivalent to one # in Markdown), and so the open tag says "here's the start of the heading" and the closing tag says "that's the end of the heading".

Tags are written with a less-than symbol **<**, then the name of the tag (often a single letter), then a greater-than symbol **>**. An *end tag* is written just like a *start tag*, but includes a forward slash / immediately after the less-than symbol—this indicates that the tag is closing the annotation.

- HTML tag names are not case sensitive, but you should always write them in all lowercase.

- Line breaks and white space around tags (including indentation) is ignored. Tags may thus be written on their own line, or *inline* with the content. These two uses of the <p> tag (which marks a *paragraph* of content) are equivalent:

  ```
  <p>
      The itsy bitsy spider went up the water spout.
  </p>
  ```

```html
<p>The itsy bitsy spider went up the water spout.</p>
```

Taken together, the tags and the content they *contain* are called an **HTML Element**. A website is made of a bunch of these elements.

Some Example Tags

The HTML standard defines lots of different elements, each of which marks a different meaning for the content. Common elements include:

- `<h1>`: a 1st-level heading
- `<h2>`: a 2nd-level heading (and so on, down to `<h6>`)
- `<p>`: a paragraph of text
- `<a>`: an "anchor", or a hyperlink
- `<img>`: an image
- `<button>`: a button
- `<em>`: emphasized content. Note that this doesn't mean *italic* (which is not semantic), but *emphasized* (which is semantic). The same as `_text_` in Markdown.
- `<strong>`: important, strongly stated content. The same as `**text**` in Markdown
- `<ul>`: an unordered list (simil)
- `<li>`: a list item (an item in a list)
- `<table>`: a data table
- `<form>`: a form for the user to fill out
- `<svg>`: a Scalable Vector Graphic (a "coded" image)
- `<circle>`: a circle (in an `<svg>` element)
- `<div>`: a division (section) of content. Also acts as an empty *block* element (followed by a line break)
- `<span>`: a span (section) of content. Also acts as an empty *inline* element (not followed by a line break)

Comments

As with every programming language, HTML includes a way to add comments to your code. It does this by using a tag with special syntax:

```html
<!-- this is a comment -->
<p>this is is not a comment</p>
```

Because that syntax is somewhat awkward to type, most source-code editors will let you comment-out the currently highlighted text by pressing `cmd + /` (or `ctrl + /` on Windows). If you're using a code editor, try placing your cursor on a line and using that keyboard command to comment and un-comment the line.

Comments can appear anywhere in the file. Just as in other languages, they
are ignored by any program reading the file (with a few interesting exceptions),
but they do remain in the page and are visible when you view the page source.

Attributes

The start tag of an element may also contain one or more **attributes**. These
are similar to attributes in object-oriented programming: they specify *proper-
ties*, options, or otherwise add additional meaning to an element. Like named
parameters in R or HTTP query parameters, attributes are written in the for-
mat `attributeName=value`; values of attributes are almost always strings, and
so are written in quotes. Multiple attributes are separated by spaces:

```
<tag attributeA="value" attributeB="value">
   content
</tag>
```

For example, a hyperlink anchor (`<a>`) uses a `href` ("**h**ypertext **ref**erence")
attribute to specify where the content should link to:

```
<a href="https://ischool.uw.edu">iSchool homepage</a>
```

- In a hyperlink, the *content* of the tag is the displayed text, and the *attribute*
  specifies the link's URL. Contrast this to the same link in Markdown:

  ```
  [iSchool homepage](https://ischool.uw.edu)
  ```

Similarly, an image (`<img>`) uses the `src` (**sou**r**c**e) attribute to specify what
picture it is showing. The `alt` attribute contains alternate text to use if the
browser can't show images—such as with screen readers (for the visible impaired)
and search engine indexers.

```
<img src="baby_picture.jpg" alt="a cute baby">
```

- Note that because an `<img>` has no textual content, it is an *empty element*
  (see below).

There are also a number of global attributes that can be used on any element.
For example:

- Every HTML element can include an **id** attribute, which is used to give
  them a unique identifier so that we can refer to them later (e.g., from CSS
  or JavaScript). `id` attributes are named like variable names, and must be
  unique on the page.

  ```
  <h1 id="title">My Web Page</h1>
  ```

- For example, the `lang` attribute is used to indicate the language in which
  the element's content is written. Programs reading this file might use that

to properly index the content, correctly pronounce it via a screen reader, or even translate it into another language:

```
<p lang="sp">No me gusta</p>
```

We specify the `lang` attribute for the `<html>` element (see below) to define the default language of the page; that way we don't need to mark the language of every element. **Always include this attribute**.

```
<html lang="en">
```

Empty Elements

A few HTML elements don't require a closing tag because they *can't* contain any content. These tags are often used for inserting media into a web page, such as with the `<img>` tag. With an `<img>` tag, you can specify the path to the image file in the `src` attribute, but the image element itself can't contain additional text or other content. Since it can't contain any content, we leave off the end tag entirely:

```
<img src="picture.png" alt="description of image for screen readers and indexers">
```

Older versions of HTML (and current related languages like XML) required you to include forward slash / just before the greater-than symbol. This "end" slash indicated that the element was complete and expected no further content:

```
<img src="picture.png" alt="description of image for screen readers and indexers" />
```

This is no longer required in HTML5, so feel free to omit that forward slash (though some purists, or those working with XML, will still include it).

3.2 Nesting Elements

Web pages are made up of multiple (hundreds! thousands!) of HTML elements. Moreover, HTML elements can be **nested**: that is, the content of an HTML element can contain *other* HTML tags (and thus other HTML elements):

Figure 3.2: An example of element nesting: the `<em>` element is nested in the `<h1>` element's content.

The semantic meaning indicated by an element applies to *all* its content: thus all the text in the above example is a top-level heading, and the content "(with emphasis)" is emphasized in addition.

Because elements can contain elements which can *themselves* contain elements, an HTML document ends up being structured as a **"tree"** of elements:
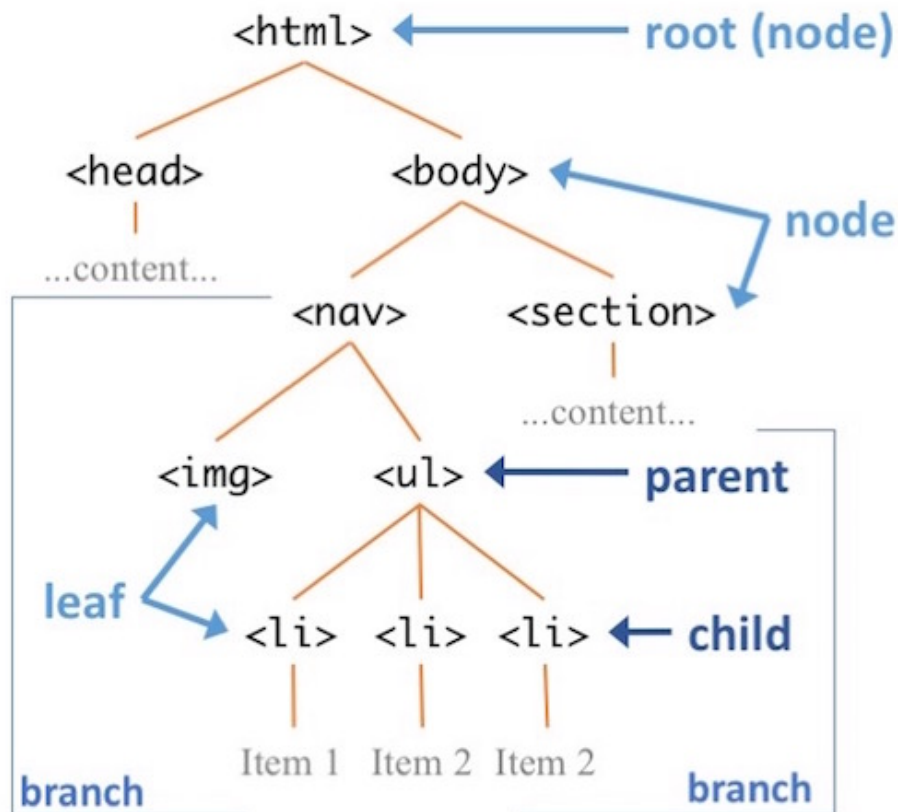

Figure 3.3: An example DOM tree (a tree of HTML elements).

In an HTML document, the "root" element of the tree is always an **`<html>`** element. Inside this we put a `<body>` element to contain the document's "body" (that is, the shown content):

```html
<html lang="en">
  <body>
    <h1>Hello world!</h1>
    <p>This is <em>conteeeeent</em>!</p>
```

```
    <body>
</html>
```

This model of HTML as a tree of "nodes"—along with an API (programming interface) for manipulating them— is known as the **Document Object Model (DOM)**. See Chapter: DOM for details.

Caution! HTML elements have to be "closed" correctly, or the semantic meaning may be incorrect! If you forget to close the `<h1>` tag, then *all* of the following content will be considered part of the heading! Remember to close your inner tags *before* you close the outer ones. Validating your HTML can help with this.

Block vs. Inline Elements

All HTML elements fall into one of two categories:

- **Block elements** form a visible "block" on a page—in particular, they will be on a new line from the previous content, and any content after it will also be on a new line. These tend to be structural elements for a page: headings (`<h1>`), paragraphs (`<p>`), lists (`<ul>`), etc.

```
<div>Block element</div>
<div>Block element</div>
```

Block element
Block element

Figure 3.4: Two block elements rendered on a page.

- **Inline elements** are contained "in the line" of content. These will *not* have a line break after them. Inline elements are used to modify the content rather than set it apart, such as giving it emphasis (`<em>`) or declaring that it to be a hyperlink (`<a>`).

```
<span>Inline element</span>
<span>Other inline element</span>
```

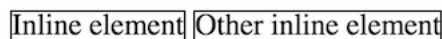| Inline element	Other inline element

Figure 3.5: Two inline elements rendered on a page.

Inline elements go inside of block elements, and it's common to put block elements inside of the other block elements (e.g., an `<li>` inside of a `<ul>`, or a `<p>` inside of a `<div>`). However, it is invalid to to nest a block element inside

of an inline element—the content won't make sense, and probably won't look right.

Some elements have further restrictions on nesting. For example, a `<ul>` (unordered list) is *only* allowed to contain `<li>` elements—anything else is invalid markup.

3.3   Web Page Structure

Now that you understand how to specify HTML elements, you can begin making real web pages! However, there are a few more tags you need to know and include for a valid, modern web page.

Doctype Declaration

All HTML files start with a document type declaration, commonly referred to as the "Doctype." This tells the rendering program (e.g., the browser) what format and syntax your document is using. Since you're writing pages with HTML 5, you can declare it as follows:

```
<!DOCTYPE html>
<html lang="en">
    ...
</html>
```

`<!DOCTYPE>` isn't technically an HTML tag (it's actally XML). While modern browsers will perform a "best guess" as to the Doctype, it is best practice to specify it explicitly. Always include the DOCTYPE at the start of your HTML files!

The

Section {-} In addition to the `<body>` element that defines the displayed content, you should also include a `<head>` element that acts as the document "header" (the `<head>` is nested inside the `<html>` at the same level as the `<body>`). The content of the `<head>` element is *not* shown on the web page—instead it provides extra (meta) information *about* the document being rendered.

There are a couple of common elements you should include in the `<head>`:

- A **`<title>`**, which specifies the "title" of the webpage:

  ```
  <title>My Page Title</title>
  ```

  Browsers will show the page title in the tab at the top of the browser window, and use that as the default bookmark name if you bookmark the page. But the title is *also* used by search indexers and screen readers for

the blind, since it often provides a strong signal about what the page's subject. Thus your title should be informative and reflective of the content.

- A `<meta>` tag that specifies the character encoding of the page:

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

The `<meta>` tag itself represents "metadata" (information about the page's data), and uses an attribute and value to specify that information. The most important `<meta>` tag is for the character set, which tells the browser how to convert binary bits from the server into letters. Nearly all editors these days will save files in the `UTF-8` character set, which supports the mixing of different scripts (Latin, Cyrillic, Chinese, Arabic, etc) in the same file.

- You can also use the `<meta>` tag to include more information about the author, description, and keywords for your page:

```
<meta name="author" content="your name">
<meta name="description" content="description of your page">
<meta name="keywords" content="list,of,keywords,separate,by,commas">
```

Note that the `name` attribute is used to specify the "variable name" for that piece of metadata, while the `content` attribute is used to specify the "value" of that metadata. `<meta>` elements are *empty elements* and have no content of their own.

Again, these are not visible in the browser window (because they are in the `<head>`!), but will be used by search engines to index your page.

  - *At the very least, always include author information for the pages you create!*

- We will discuss additional elements for the `<head>` section throughout the text, such as using `<link>` to include CSS and using `<script>` to include JavaScript.

3.4 Web Page Template

Putting this all together produces the following "template" for making a web page:

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="author" content="your name">
   <meta name="description" content="description of your page">
   <title>My Page Title</title>
```

```
</head>
<body>
   ...
   Content goes here!
   ...
</body>
</html>
```

You can use this to start off every web page you ever create from now on!

Resources

- Getting Starting with HTML
- General HTML 5 Reference
- Alphabetical HTML Tag Reference
- Dive into HTML5 - Free book on HTML
- W3C HTML Validation Service

Also remember you can view the HTML page source of *any* webpage you visit. Use that to explore how others have developed pages and to learn new tricks and techniques!

Chapter 4

Accessible HTML