BEGINNER'S
GUIDE

TO

## WEB DEVELOPMENT

CONTENTS

HOW DOES A WEBSITE WORK?

You're probably familiar with the *concept* of a website, but maybe not with how a website actually works. Below, you'll find an overview of each of the components necessary to view and interact with the website you're currently on and virtually all other websites on the internet.

What Is a Website?

A website is a collection of individual documents and files made up of text, graphics, colors, links, and formatting that come together to create a complete user experience. Websites are also usually associated with domain names, like www.codeschool.com, that explain to your computer where all of the files that are necessary to display a website are located.

What Is a Web Browser?

Websites are accessible through web browsers. A web browser is a computer application capable of *downloading* and *displaying* the files and components that make up a website. Popular web browsers like Google Chrome, Mozilla Firefox, and Safari are all able to read and interpret domain names like www.codeschool.com, request the necessary files to display those websites, and render them on your screen as a website.
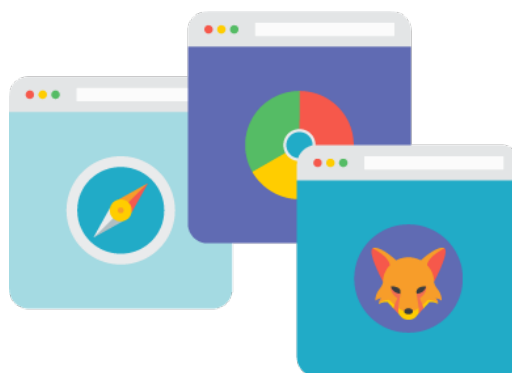
HTML

At a basic level, all websites are written in a language called HTML, or Hypertext Markup Language. HTML is a universal set of instructions that every web browser is capable of understanding.

Text, images, and formatting are the types of content that can be written in HTML. HTML code is stored within documents with the file type *.html* that your web browser uses to know precisely how to display a web page. Collectively, the HTML documents and images used to create a website are sometimes referred to as *assets*.

What Is a Web Server?

Websites and their associated HTML documents and files are stored on computers called web servers. Web servers must be able to receive requests from a user's web browser and send the necessary files back to them to display a website.

Web servers are not unlike your own personal computer in that they're able to separate files and folders just like you do at home, except they're often connected to very fast internet connections and offer large amounts of storage capable of handling hundreds or thousands of simultaneous users. Popular websites like Amazon.com rely on large web servers to handle millions of product descriptions, product images, and purchases every day.

While HTML is the technology used by web browsers to display content to a user, web servers rely on different languages to operate. The languages and technologies used to manage incoming user requests for website files and handle the organization and storage of user data are often called server-side languages.

Combining Everything

When you type a domain name into your web browser, your browser sends a request to a web server where the website's files are located. Your browser downloads these files, usually HTML documents and accompanying images or videos, and renders them on your screen. HTML and other languages used to display the data by your web browser are typically referred to as **"front-end"** technologies in the web-development space because of their user-facing tendencies.

As you enter things like credit card information or submit a form on a site, the data you send back and forth to the web server is managed by server-side languages, sometimes referred to as **"back-end"** technologies. These languages make the organizing of databases easier, as well as manage user requests for new web pages as they navigate a site.
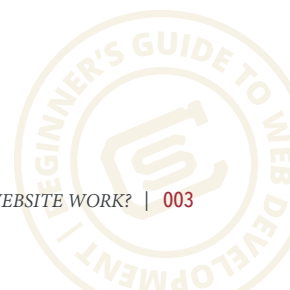
Wix, Squarespace, and Defining Web Development

Even if you're new to the web development space, there's a chance you've heard of services like Wix and Squarespace, which are ready-to-use website building plat-forms. These services use things like pre-made HTML templates and have web servers already set up to handle incoming user requests and manage user data. While they're often the first place beginners go to get a website online and offer a small amount of customizability, we believe there's power in control — learning the tools that make the web work will help you make smarter decisions about how your own website looks and works. You won't be limited to using a one-size-fits-all platform.

WRITING YOUR FIRST WEB PAGE

Once you've got an idea for the kind of web page you want to build, you'll need to start writing some HTML and CSS.

HTML is a markup language that was created solely for making web pages. The idea is that, at their core, web pages are just a bunch of text and images arranged on the screen in different ways, and markup is just a way to attach instructions for how to make those text and images appear.

For example, take this simple list of Paths on codeschool.com:
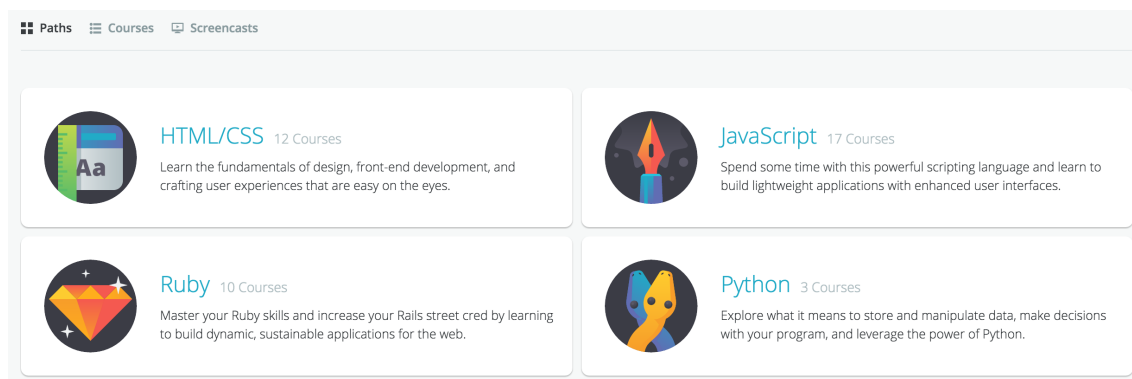
It looks like there's a lot going on, but if we take away all the visual formatting, it will look like this:

Now there's a lot less going on. The only difference between the two is that to generate the screenshot in the second image, we deleted all the CSS code and left just the HTML code in place — HTML code that looks a lot like this:

```
<body>
  <nav>
    <li>Paths</li>
    <li>Courses</li>
    <li>Screencasts</li>
  </nav>
  <main>
    <section>
      <img src="html-css-badge.png" />
      <h2>HTML/CSS 12 courses</h2>
      <p>Learn the fundamentals of design, front-end development…</p>
    </section>
    <section>
      <img src="javascript-badge.png" />
      <h2>JavaScript 17 courses</h2>
      <p>Spend some time with this powerful scripting language and
learn...</p>
    </section>
  </main>
</body>
```

The words "Paths," "Courses," and "Screencasts" are just text, but by putting them inside of those `<nav>` and `<li>` tags, the web browser knows to display them as a bulleted list. The `<img>` tag is used to load an image — in this case, the circular badge icon that represents that learning Path. `<h2>` stands for "heading 2" — you can think of heading tags as outlining the main sections of a page. Finally, `<p>` is short for "paragraph," and any time you need to show some simple text on a web page, it will likely be in a `<p>` tag.

The `<main>` and `<section>` tags are just a way to group that content at an even higher level. For example, each `<img>`, `<h2>`, and `<p>` tag is wrapped in a `<section>` tag to show that those three are all related information. Each of those sections is a part of the main area of the page, so that gets wrapped in a `<main>` tag. There are over 100 specific HTML tags that all have a different purpose, and at least knowing how to find the right tag that fits your specific need is an important part of being a web developer.

Which Browser Should I Use?

The web browser knows how to take all those tags and the content inside them and display them as a web page. But if all browsers have the ability to display web pages, then why isn't there just one instead of multiple? Well, each browser implements the rendering of HTML and CSS a little differently. This is actually a source of

much stress for web developers — they have no way to control which web browser a user will experience their site with, so they spend a lot of time tweaking CSS to ensure it displays equally as close as possible across all different browsers.

Styling the Page

While HTML is a markup language used for arranging content on a page, CSS is a separate language for changing the style of how those tags are displayed. We started this article by removing all the CSS, so let's look at how we can add CSS back in.

It's a best practice to store your CSS code in a separate file from your HTML code and then link to it from your HTML. For example, if you've got a CSS file named **main.css** in the same folder as an HTML file named **index.html**, you'd link to it from the <head> of the index file, like this:

```
 <head>
    <link rel="stylesheet" href="main.css"></link>
  </head>
```

Then, in **main.css**, you can write CSS rules that change the way HTML tags appear in a browser. Remember how our sections originally looked like boxes with an image and text in them, but when we took away the CSS file, those boxes disappeared? We could add them back in like this:

```
section {
  display: inline-block;
  border-radius: 4px;
  max-width: 400px;
  border: 1px solid lightgray;
}
```

This CSS is selecting all of the section tags and applying four properties to those tags.

What about the navigation menu? We want those Paths/Courses/Screencasts links to appear from left to right instead of stacked top to bottom, and we could accomplish that like this:

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

This CSS removes the bullet points from each of the three items in the list, and then the **display: inline** rule tells those list items to display left to right.

Like HTML tags, there are hundreds of CSS properties and a few dozen different ways to select which HTML tags to apply those rules to, and while a web developer doesn't necessarily need to memorize all those properties, they do need to be familiar with how to research the ones they need and put them into practice.
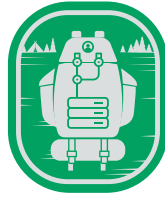
SERVER-SIDE LANGUAGES

Why We Need Server-side Languages

Websites require two key components to function: a client and a web server. Clients, as we've learned, are any web browser or device that's used to view and interact with a website. All of the files and data associated with displaying a website on a client are stored on a web server.

If you've ever purchased something online, many of the processes necessary to complete a transaction happen on the shop's web server. Shopping sites need things like user accounts, up-to-date inventories, product descriptions, the ability to receive shipping and credit card information, and a place for shoppers to review products they've purchased. This information is stored in large databases on web servers, away from users' web browsers.

In order to display products on the screen, you would need a **programming language** that runs on a server. Such a programming language would take care of creating, reading, updating, and deleting products from a database, which is where the products are stored. Languages that do this are referred to as **server-side languages**.

The Shopping Process

To understand some of the functions that server-side languages are able to perform, let's take a closer look at the purchasing process in an online store.

After you've filled your shopping cart and are ready to check out, most shopping sites first prompt you to create or log in to an existing account before you can complete your purchase. When you enter your username and password on the login page, the information you enter is sent to a web server to be validated. Server-side languages are used to compare the login information you provided with their database of existing users, and either confirm or deny that the login information you entered is correct.

By requiring users to log in before they make a purchase, shopping sites are able to enhance the user experience by remembering things like shipping and credit card information to help save time at checkout. These types of information are stored alongside login credentials in the same large databases on the web server.

After you've made a purchase and received your product in the mail, you may
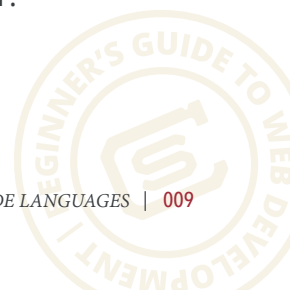
decide that you'd like to leave a product review for future shoppers. Product reviews rely on similar server-side processes to require a user to log in to their account before reviewing a product and also making sure product reviews are posted to the correct product page.

Server-side languages are an integral part of both the online shopping experience and the entire web. Web servers provide a "central hub" where not only HTML documents are stored, but also the processes required to store, manipulate, and distribute data in real time as users interact with a site.

Which Language Do I Start With?

As mentioned previously, most server-side languages provide very similar functionality and are all capable of producing an almost identical end product. While each language has its quirks, we recommend picking a language that sounds interesting and sticking with it rather than trying to find the "best" first language to learn. As you begin to understand the logic employed by one server-side language, you'll be able to use that knowledge to learn other languages and other styles of solving the same types of problems.

Frameworks

Many of the processes performed by a web server can get repetitive. After all, server-side applications are often designed to handle similar requests from thousands of users at a time. To alleviate some of the burden of writing unique code for every task a server-side language must perform, developers began creating something called **frameworks**. Frameworks provide a type of programming shorthand for common, repetitive tasks. Instead of spending time coding functionality from scratch, frameworks make developing applications easier by shortening development time, as well as offering standards and conventions that entire development teams can get behind.

Frameworks are like buying a box of cake mix at the store instead of measuring out and combining all of the individual ingredients necessary to make a cake.

Frameworks in an Online Shop

In keeping with our online shopping example, as databases of products or user accounts on an online store grow, the need for a way to easily update these databases becomes necessary.

A common example of a framework stepping in to handle the heavy lifting of updating a database comes in the form of something called object-relational mapping (ORM). Because the code syntax used to look up a product in a database is often different than the code syntax used to actually send that product information back to a user's web browser, frameworks provide an easy way for both syntaxes to "talk to each other."

Frameworks are built on a language-by-language basis, meaning the code for an ORM in Ruby would look different than the code for an ORM framework in Java, but they would both perform similar functions.

ORMs are just one example of how frameworks can help speed up the process of developing server-side applications.

Popular Server-side Languages

As mentioned above, the goal of a web server is to distribute the correct HTML files to the clients requesting them, maintain databases, and validate user inputs like login credentials. Just like the variety of cars on the road are all capable of bringing you from point A to point B, server-side languages all perform the same core functions, just in varying styles, speeds, and techniques. As the web has grown, so have the number of server-side languages to choose from.

Below are a few popular server-side languages you may have heard of (presented in alphabetical order), as well as the most popular frameworks used to simplify workflows and establish development standards across teams.

C# (pronounced C-Sharp)
C# was developed by Microsoft and is typically used by businesses to manage large databases. Because of the prevalence of existing Microsoft software in businesses, C# was adopted quickly.
Associated Framework: ASP.NET

Go
Go is a programming language created by Google with performance in mind.

Google does things at an unprecedented level of scale, so instead of making existing languages adapt to their needs, they decided it would be a better idea to make a new one with scalability in mind.
*Associated Frameworks: Gorilla & Revel*

Java
Java is one of the oldest and most widely adopted programming languages. Originally intended to be used to develop standalone desktop applications, a team of developers found a way to use it on web servers in the early 2000s.
Associated Framework: Spring

Node.js (JavaScript)
As the popularity of JavaScript grew to add interactivity to a website's interface, some members of the community found a way to also use it as a server-side language. Node uses the same JavaScript syntax on the web server.
Associated Frameworks: Express & Hapi
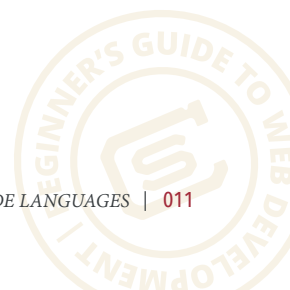
Python

Python is popular in universities to teach students their first programming languages, and it is widely used in production. Since it's popular in academic hubs, its community thrives in writing math and science libraries.
*Associated Framework: Django*

PHP

Unlike other languages that had to be adapted for use on the web, PHP was designed with web development in mind from day one. Many content management systems like Wordpress are written in PHP.
Associated Frameworks: Laravel & Symfony

Ruby

Ruby touts itself as an elegant and productive programming language. Originally popular in Japan in the '90s, Ruby grew in popularity in the rest of the world after its now renowned framework, Ruby on Rails, was added.
Associated Framework: Ruby on Rails
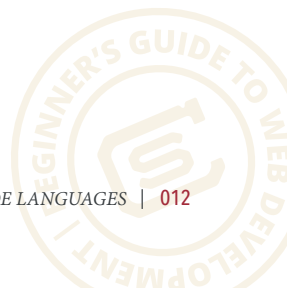
MAKING WEB PAGES MORE INTERACTIVE WITH JAVASCRIPT

In the early days of the web, web pages were largely static — meaning that after a site's HTML documents were downloaded and displayed by the web browser, the content wouldn't change. To display changes made on a web page, the web browser would have to download a completely new version of the HTML documents from the web server — a slow and inefficient process.

Nowadays, web pages are considered more "dynamic," since they're capable of responding to user input in real time. Things like the ability to "favorite" social media posts or update the inventory in a shopping cart in an online shop are examples of dynamic content — they don't require the browser to re-download the page to display changes.

The language capable of providing much of this interactivity and responsiveness is called JavaScript, and it runs directly inside of the web browser. All modern web browsers are capable of interpreting the JavaScript language.

JavaScript code is written in separate files that are downloaded alongside a site's HTML documents when the browser visits a website. Both file types are capable of referencing each other's content.

Creating a Button

To conceptualize some of the things JavaScript is capable of, let's take a look at a button that changes color every time it's clicked. Let's assume we created this purple button using HTML. Using JavaScript code, we're able to "watch" for when a user clicks our button. After we see a user click, we can use JavaScript to change the button's color to green.

Without JavaScript responding to user input in real time, our button wouldn't have the ability to change colors when clicked. Before JavaScript, we would have had to re-download an entirely new HTML document that contained a green button instead of a purple button in order to display the change to the user.

Making the Real World More Interactive

Building upon our button example, let's take a look at JavaScript concepts at work in the "real world" in an online shopping site like Amazon.com. Amazon embraces JavaScript-driven interactivity to make their shopping experience easier.


Click to open expanded view

The most obvious use of JavaScript on Amazon.com is on their product pages. Above, we see a product page for pens. Surrounding the primary pen image, a series of smaller images allows shoppers to scroll through different product angles, colors, and styles of the same pen. As you place your cursor over these smaller images, the primary image changes. You can also hover over the main product image to zoom in on the product.

In this instance, JavaScript code is watching where our cursor moves and is responding dynamically without ever having to re-download the page with a new image.

JavaScript has allowed sites like Amazon to provide an enhanced customer experience to shoppers and also reduces the amount of workload their web servers have to do delivering web pages. These features have a direct impact on Amazon's sales.

Form Validation: Another Type of Interactivity

So far, we've been exploring how JavaScript can impact visual elements. JavaScript can also play an important role behind the scenes by performing functions that aren't even seen by the user. An example of that is form validation, or verifying the information a user enters into a form before it can be submitted.
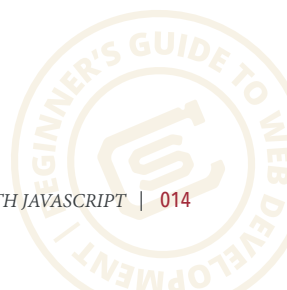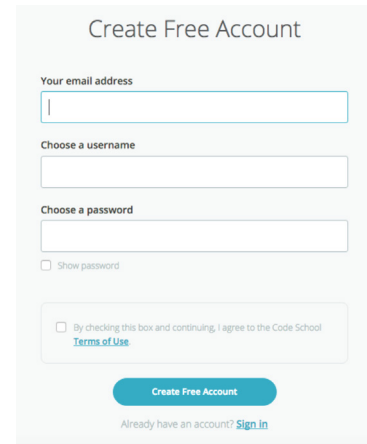
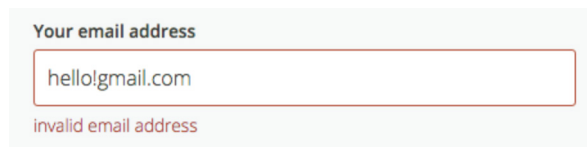This form has three unique validation requirements:

First, all three fields are mandatory and cannot be empty when the form is submitted. Each new user who registers for an account must provide an email address, a username, and a password.

Second, the user's email address must be real. If the email isn't real, it would be impossible to contact them.

Finally, users need to create a secure password with at least six characters.

Using JavaScript, we are able to verify that all these requirements have been met before allowing the form to be submitted. If information is entered incorrectly, developers must include visual cues to show a user why their form is not yet complete. For example, if a user accidentally leaves out the "@" sign from their email address, we can provide an "Invalid email address" error message.

JavaScript Must Be Really Hard to Learn, Right?

Actually, not really. While there are many advanced concepts behind JavaScript, it does not take much to write your first few lines of JavaScript code and get something working.

Our team here at Code School built the JavaScript.com website with this in mind. There you will be able to write your first lines of JavaScript code, as well as find additional resources on how you can go about gaining deeper knowledge of the language.

JavaScript Outside of the Browser

So far, we've described JavaScript as a programming language that works within the web browser. While it was originally developed exclusively for this purpose, some members of the JavaScript community found a way to repurpose the language to also be used on web servers in a project called Node.js. The ability to use JavaScript both within the web browser and on a web server reduces the learning curve of building a complete website.

We'll learn more about Node.js and other server languages in the next chapter.
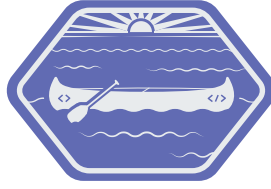
DEPLOYING YOUR FIRST WEBSITE

Once you've put in all the hard work of creating a website, you need to get it on the web so people can navigate to it and access its content. This process is called *deployment*. Deployment is a fancy word for "getting your website on the web," and there are a few different parts of that process we'll discuss below:

1. Finding a domain name
2. Finding a hosting service
3. Uploading files with SFTP
4. Deploying server-side applications

Finding a Domain Name

A domain name is the address you type into a web browser to visit a website. A few domain names you might be familiar with are *facebook.com*, *google.com*, *wikipedia.org*, and even *codeschool.com*, which is the one you're visiting right now.
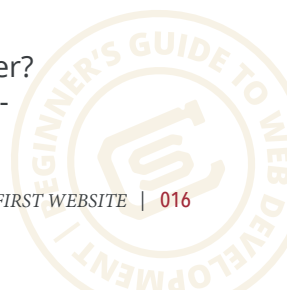
Finding a domain name with `.com` at the end that doesn't already exist can be quite challenging these days because a lot of them are already taken. Fortunately, there are many alternatives thanks to new domain types (also known as top-level domains, or TLDs), such as `.coffee`, `.technology` or even `.florist`. That's right — anybody can own a domain like `jon.technology` these days.

How to Purchase a Domain Name

Owning a domain name costs money, and there are a number of services out there that allow you to find and purchase them, such as GoDaddy, Namecheap, and Google Domains. Pricing for domain names ranges from about $10 to $80 per year depending on the TLD you choose. Domain names ending in `.com` or `.net` are usually cheaper than those ending in `.io` or `.coffee,` for example.

With these services, you can search for available domain names — often with tools that allow you to search for multiple domain names or TLDs simultaneously. Once you've found a domain name you'd like, you'll be asked to provide some legally required information about yourself as the owner of the domain name, make payment, and the domain name will be all yours.

Now that you've purchased a domain name, you need to associate it with a web server so that every time someone types your domain name into their browser, the website files located on your server will load. But where do you get that server? You *could* use a home computer and serve files from your home internet connec-

tion, but you'd then be required to set up and maintain the computer, as well as pay for all of the bandwidth yourself. Alternatively, you could pay to rent a web server and bandwidth from a hosting service, which is often more practical and cost-effective. When you rent a server from a **hosting service**, they'll give you a way to access it, upload files, and even install custom **scripts** to keep your site up and running.

Hosting services often start at just a few USD per month depending on the amount of storage and bandwidth you expect to use.

Setting Up Nameservers

All of your website's files are stored and served from the web server that your hosting service provides. But how does a web browser know to look for your web server when someone types your domain name into their web browser? That's the job of a **nameserver**.

A nameserver can be thought of as a phone book for the internet. Nameservers are maintained by the companies who sell domain names. They maintain a list of all the currently registered domain names around the world and the IP addresses associated with the web servers hosting their files.

Nameservers are also able to communicate with each other so that an updated list of all currently registered domain names is always available. After you purchase a domain name, you'll be asked to configure that domain name's nameserver information. Registering your domain name with a nameserver ensures that anytime someone types your domain name into their browser, the correct files from your server will display.

It is often possible to purchase a domain name and hosting services from the same company. In this case, your domain name's nameserver information may be automatically configured for you.

Once your nameserver is configured, you should be able to visit your own website by typing your domain name into your browser.

Getting Your Files on the Server

Now that your domain name and web server are configured, it's time to get your website's files onto the server. While you may have an entire website built on your local computer, it won't be visible to the rest of the world until it's on your server! For small websites, like those that contain only basic HTML and JavaScript, you may be able to upload files directly to the server. For more complicated websites, like those that are dependent on specific programming languages or frameworks to operate correctly, you'll need a web server capable of running custom **scripts** that ensure the latest versions of each programming language or framework necessary to display your website are installed on the web server.

In most cases, "deployment" simply means getting your website files onto the server. For more complicated websites, deployment also means creating and running the scripts necessary to keep your web server up to date with the correct version of each programming language and framework.

Writing these types of scripts from scratch requires advanced knowledge of server administration, which some individuals dedicate their entire career to learning. While it may sound difficult, most skilled developers eventually learn some aspects of server administration throughout their career.

Managed and Cloud-based Servers

Thankfully, deploying your first website likely won't require learning all of the intricacies of managing a web server. Many hosting companies have removed the burden of learning all there is to know about server administration to make deploying websites easier. Companies like Heroku, DigitalOcean, and Amazon Web Services make it easy to rent a web server to host your website with the tools and necessary scripts already installed.

While these companies aren't identical in the services they offer, the end goal is similar: renting a server to deploy your website or application to the web.

The process of deploying your application will vary depending on the service you end up using, as well as the language you used to develop your application, but in many cases, server providers will give you sample deployment scripts that you can configure to meet your site's needs.

SEMANTIC HTML

HTML is a markup language, which means it's made up of two things: *content* and *markup* to describe that content. In HTML, markup exists in the form of different *tags*, and you can use those tags to describe the individual pieces of content throughout a website. When a site's markup is "semantic," it means the tags appropriately describe the content.

In an element like `<p>Hello world!</p>`, the `<p>` and `</p>` parts make up the tag, while the Hello world! part is the content inside. The reason we're using `<p>` here is because the `<p>` tag stands for *paragraph* in HTML. A paragraph can be a short sentence or something longer, but in either case, the `<p>` tag is a *semantic* choice because it describes what comes inside.

Websites can have many different types of content, so HTML has many different types of tags. There are tags for links, forms, images, videos, headers, footers, and a whole lot more. Think of any kind of content, and there's likely a semantic way to write it in HTML.

Why Is Semantic HTML Important?

Before a browser can show anything on screen, it has to read a website's HTML and then figure out what to do with it. Every HTML tag has its own default styles and behavior, so if the wrong tag gets used on some content, that element won't work in the browser as expected. Semantic markup is important because it keeps websites working in a consistent and usable way.

Styles

If we look at what the browser shows for non-semantic and semantic markup, there are several differences between the two.
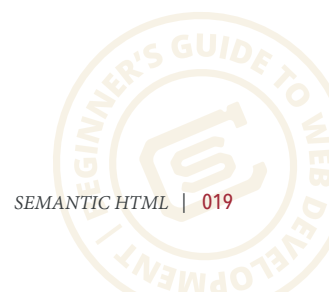
Chocolate Chip Cookies
Quick and easy, ooey, gooey chocolate chip cookies made from scratch at home. Yum!
Ingredients
3 cups flour
1 cup butter
2 cups chocolate chips

Chocolate Chip Cookies

Quick and easy, ooey, gooey chocolate chip cookies made from scratch at home. **Yum!**

Ingredients

- 3 cups flour
- 1 cup butter
- 2 cups chocolate chips

The first example uses the generic tags `<div>` and `<span>` to mark up content, while the second example uses semantic markup like `<p>` for paragraphs, `<h1>` and `<h2>` for headings, and `<ol>` and `<li>` for lists. Improving semantics throughout the markup makes the resulting page look a lot more like a functional website.

Functionality

Apart from how things *look*, semantic markup can also improve how things *function*. Look at what happens (or doesn't happen) when we try to interact with these two web pages.

In the first example, we're using CSS to style non-semantic markup exactly the same as semantic markup. But even though the two look the same, they don't function the same. Generic tags like `<div>` and `<span>` don't have any default functionality, so nothing happens when we try to focus or click them. A semantic tag like `<button>` does have default functionality, so it can be focused and clicked without the need for any extra code.

Conclusion

It's important to be semantic when writing HTML because it improves how meaningful the markup is. Semantic markup makes things easier to write and maintain, but it can also save a lot of time in development by enabling default styles and functionality. Even better: Websites written with semantic markup are more usable and can offer a better experience than websites written in HTML that's not semantic.

CSS PREPROCESSORS AND FRAMEWORKS

CSS preprocessors and frameworks are both tools to help developers save time building websites. A CSS *preprocessor* is its own language built on top of CSS that adds scripting functionality. A CSS *framework* is a prewritten style sheet that includes its own layout and theme for user interface elements.
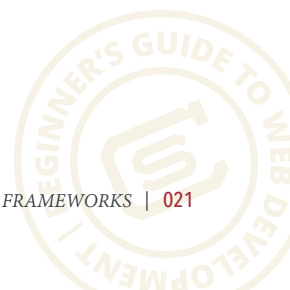
CSS Preprocessors

CSS is designed to be a simple language. It's intuitive to learn CSS and write it on small websites, but on larger, more complex websites, writing CSS can become very tedious. That's where preprocessors come in. Developers wanted more options when writing their style sheets, so they created CSS preprocessors to add programming functionality.

The most commonly used CSS preprocessors are Sass, Less, and Stylus, and some of their common features include:

→ Addition, subtraction, multiplication, and division. Writing out math problems makes code easier to maintain in the future and helps avoid scary "magic numbers" that get pasted in without explanation.

→ Variables. An example would be setting a color variable in one place and then referencing it multiple times throughout a style sheet.

→ Functions. For example, the built-in percentage() function that turns any number into a percentage value in CSS.

→ If/else conditionals. Conditionals are useful for toggling CSS throughout a style sheet or even switching between an entire light or dark theme.

→ Loops. Loops help avoid repetitive code when writing things like grids or creating individual classes for a bunch of different colors.

These features are present across almost every programming language, so it's no surprise that developers would want them in CSS, too.

But how do preprocessors add their own features to CSS? Well, they work by *compiling* special style sheets into plain CSS. That means a developer can use variables, functions, loops, and more throughout their style sheets, and the preprocessor will compile everything into CSS that browsers can understand.

CSS preprocessors are tremendously useful, but they only make it easier to write CSS — they don't come with their own styles. That's why CSS *frameworks* exist.

CSS Frameworks

CSS frameworks contain prewritten styles for common design elements across the web. Websites today often need a lot of the same things: buttons, tables, forms, and grids, for example. CSS frameworks offer a quick and well-documented way to style all of these and more.

Frameworks like Bootstrap and Foundation are popular open source projects with hundreds of contributors. Their code is well tested by developers all over the world. If something breaks, it's sure to get reported and fixed by the community. Their popularity also means there are a lot of plugins and themes available for more specific features.

Let's take a closer look at some of what Bootstrap has to offer, like tables. Tables can be one of the most tedious things to style in CSS. The browser default tables look pretty rough, and it takes difficult CSS to get them looking nice. Luckily, Bootstrap has its own styles for tables that look good and are quick and easy to work with.

#	First Name	Last Name	Username
1	Mark	Otto	@mdo
2	Jacob	Thornton	@fat
3	Larry	the Bird	@twitter

Almost every website needs some kind of grid system. Twelve-column grids are the most popular because they allow layouts to divide into halves, thirds, fourths, and so on. Setting up a 12-column grid that's responsive across different screen sizes can be a lot of work, but using a CSS framework like Bootstrap means developers can start their project with a grid system already in place — which means they can focus on the more interesting and demanding aspects of development.

.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1	.col-md-1
.col-md-8								.col-md-4			
.col-md-4				.col-md-4				.col-md-4			
.col-md-6						.col-md-6					
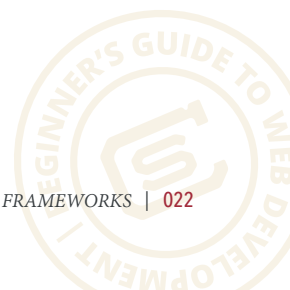
A lot of CSS frameworks are built with preprocessors, too. It's not a one-or-the-other decision — developers can use one, both, or neither.

Conclusion

CSS preprocessors and frameworks can both save developers a lot of time building websites. They offer features and convenience that aren't possible when writing CSS from scratch, and their growing popularity proves just how useful they are.

BUILDING RESPONSIVE WEBSITES

Web designers and developers used to just make different versions of websites for different devices, but now that sites are viewable on desktop, laptop, smartphone, and tablet screens, there needs to be a way to build a single website that can display appropriately on each device. Enter responsive websites.

Here's what that looks like on alistapart.com:

See how everything in the desktop browser moves around to fit on a smartphone? That's the main idea behind responsive web design, and there are a lot of advantages that come with it.

Consistency

A single, responsive website offers people a *consistent experience* across all their devices. Navigation and features may look different on different-sized screens, but they're easy to recognize and offer the same functionality. That means if someone usually uses a site on their laptop, they won't have to hunt things down if they need to do something on their phone.

The URLs are the same, too, so any shared links won't take people to the "desktop version" on mobile or the "mobile version" on desktop.
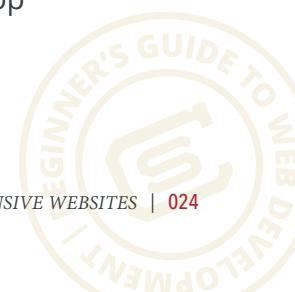
Build Once, Not Twice

Responsive web design doesn't just make things easy to use and understand for users — it's also better for web designers and developers. Creating a responsive website means coordinating design and development all at once, rather than separately for different desktop and mobile sites. It's much simpler to design and develop one website than two.

Conclusion

Responsive web design has become wildly popular across today's websites. The next time you're browsing the web, try stretching your browser back and forth on different sites to see if they're responsive. Finding a site that isn't responsive is a lot harder than you'd think.

SINGLE-PAGE APPLICATIONS

The term "single-page application" (or SPA) is usually used to describe applications that were built for the web. These applications are accessed via a web browser like other websites, but offer more dynamic interactions resembling native mobile and desktop apps.

The most notable difference between a regular website and an SPA is the reduced amount of page refreshes. SPAs have a heavier usage of AJAX — a way to communicate with back-end servers without doing a full page refresh — to get data loaded into our application. As a result, the process of rendering pages happens mostly on the client-side.

Single-page App Cons

While building SPAs is trendy and considered a modern development practice, it's important to be aware of its cons, including:

→ The browser does most of the heavy lifting, which means performance can be a problem — especially on less capable mobile devices.
→ Careful thought must be put into search engine optimization (SEO) so your content can be discoverable by search engines and social media websites that provide a link preview.

Mitigating Cons With Server-side Rendering

Most modern JavaScript frameworks are working on ways to handle server-side rendering of SPAs — meaning the user would get a fully populated page when the SPA is loaded for the first time, instead of, for example, seeing a loading indicator.

Server-side rendering can alleviate some of the burden browsers have to go through when rendering pages, and will also help with the problem of SEO and content discoverability.

Popular JavaScript Frameworks for Building SPAs

The more interactivity that happens on the client-side, the more JavaScript code is needed to make those interactive pieces function well. And the more code is written, the more important it is to have a clean and well-architected codebase. And this is exactly the problem JavaScript frameworks help solve — each with its own approach.

There are a lot of open source JavaScript frameworks that help with building SPAs, such as:

→ Angular
→ React
→ Ember
→ Aurelia
→ Vue.js
→ Cycle.js
→ Backbone

The list could go on and on — but let's dive a bit more into the first two here: Angular and React.

Angular

Angular is a front-end framework built to ease the burden of writing complex apps while keeping everything testable and organized. The first version of Angular was created back in 2009, and it was way ahead of its time. When Angular was first written, it solved a lot of problems that have now been fixed at the JavaScript language level with the release of ES2015. While Angular 1 had to create its own solution for modules, for instance, ES2015 now provides a solution to JavaScript modules right at the language level.

With inadequacies like this in mind, the Angular team at Google rewrote the framework from scratch. The new and improved version of Angular (known as Angular 2) is very promising. Angular's overall proposition has not changed: It provides a holistic solution to writing apps. Built right into the framework is a way to do application routing, communicating with web servers, and more. You won't need any extra packages to get a basic but fully functioning web app up and running.

Angular also provides a whole ecosystem (Angular CLI) for actually building apps that includes a tool to scaffold applications, as well as a solution for building mobile web apps with performance in mind as a first-class citizen (Angular Mobile).

React

React is not considered a framework, per se — rather, it touts itself as a view library. But don't let that fool you, as React was built to solve user interface problems at a very large scale.

React was pretty disruptive when it was first announced. The idea of "Rethinking Best Practices" was attached to it with good reason — React's proposition was very different than the trends the rest of the JavaScript community was moving toward at the time. While other framework authors were focusing on applying the MVC pattern to writing apps and having a clear separation of code associated to the view and other parts of the application, React proposed coupling those together

and simplifying the code by means of composition and other functional paradigms through components.

In the time since React was released, other popular frameworks, such as Angular and Ember, have also moved toward a more component-based architecture, as well as applied similar ideas React brought to light, such as the virtual DOM, which is how React applies changes to the view.

React's overall approach to building apps is different than Angular and Ember because it is just a view library — it does not provide a way to do client-side routing or even a way to load data from a back-end server bundled with the library. This means that to get a simple yet fully functioning app (that needs these missing features) to work, you will need to look into other libraries that are not maintained by the same Facebook core team that maintains React.

Don't be intimidated by the idea of relying on outside libraries, though. There are many popular community-backed libraries that have even been adopted by teams inside of Facebook, including react-router, which provides a way to client-side routing; axios, which has a system to make AJAX calls that easily integrate into React apps; and redux, which is a state container that helps developers handle the flow of data in React apps.

Finding the Right Framework for You

There are ongoing flame wars online about which is the best framework for building apps. Indeed, each has its own strengths and weaknesses, which we will not get into in this chapter. They all, however, have one thing in common: JavaScript.

And that's what we recommend: Learn JavaScript and its modern ecosystem well. Adopt Node.js, even for your client-side development, as it will bring a lot of good tooling to help bundle your code, run linters to avoid common syntax mistakes, run unit tests, and more. Once you've adopted Node.js, be sure to bring all your dependencies in through NPM, including your CSS dependencies, such as Font Awesome, Twitter Bootstrap, etc. Adopt ES2015 (also known as ES6) and all its latest features through Babel, which transpiles ES6 code back to ES5 JavaScript code that browsers can understand, since they haven't been fully upgraded to support the latest features of JavaScript yet. Also, if you are a fan of strict typed languages, then be sure to look into Flow and/or TypeScript.

Once you've gotten a hang of tooling and the JavaScript ecosystem, and have identified which parts you like the most, pick a framework that fits your choice. Ultimately, the each framework's goal is the same: building an awesome app. So it will be up to you to figure out if you want, for example, a more object-oriented or a more functional approach when building apps.

You'll also have to decide how much control you want over your application's architecture. If you want to make most of the architecture decisions yourself, then

you might be more interested in bringing different libraries together, which is more compatible with how React works. But if you prefer to have most decisions made for you, and you're okay with giving up a bit of flexibility, then maybe Angular or Ember will be the best choice, as they both provide more opinionated ways of writing your app right out of the box.

Ultimately, there is no wrong or right answer when choosing your JavaScript framework and tooling of choice, so it is up to you to figure out what works best in your situation, depending on your level of experience, how much flexibility your job provides, how much code is already written, and other factors.

LEARNING GIT AND VERSION CONTROL

If we could only pick one tool for any modern web developer to learn, it would be **Git**. Git is a **version control system (VCS)** that helps you manage changes to all the files in your project as you write and change code. Git also has the added benefit of being a **distributed VCS**, which means teams with many developers can use Git to avoid issues while working on the same part of an app at the same time.

Before we dig deeper into what Git is actually doing, let's talk about how making changes to files works without a version control system.

Say that you and your friend love watching movies. One day you're talking about all the movies each of you has seen when you realize there isn't much overlap, so you decide to make a list of all the movies you want to show your friend. You start by opening up a file and adding some movies, and then you save the file when you're done. Easy, right?

The next day you tell your friend how excited you were to make the list and how you want to see their list too, but it doesn't make sense to have two separate lists, so you email your file to them. One day passes, and then another, and you really wish they'd get that list back to you because you thought of some more great movies to add. You can't take it anymore, so you open up your original copy of the list and add your movies to the end. "I'll just figure it out later," you think.

Days pass, and you've forgotten all about the list when an email appears in your inbox with the subject line, "Here's my list, friend!" You open it up, but to your surprise they haven't just added their movies to the end, they've just randomly tossed them in between everything in your original list! To make matters worse, some of the movies you added after you sent the original were also added by your friend, so now there are duplicate movie names and there's no way to even tell which one of you added which movie.

A distributed version control system like Git could have saved you both a lot of trouble. Git stores files in a **repository**, which looks just like a directory with files in it but has a few special powers. First, when you save a file that's in a Git repository, you'll see a message that the repository has **unstaged** changes. You can use the command `git add` to **stage** a file, and then `git commit` to **save** it into the repository.

The reason for this two-step process is that you can stage multiple files at once and then save them with a single commit. You can set a message for each commit, too, like, "added Francis Ford Coppola movies," or "removed duplicates," and you can view a list of all the commits at once by typing `git log`. If you're smart with your commit messages, this log becomes a very readable history of how your files have changed.

But what about your friend making changes at the same time as you? How does Git help with that? That's where the **distributed** part comes in. By using a service like GitHub, BitBucket, or GitLab, you can also store a complete copy of a Git repository on the web, and anyone working on the project in that Git repository can access the latest version of all the files in that project. The process of saving a change with a commit is still the same — make a change, stage it, then commit it — but now there's one more step: **pushing** that change up to GitHub (or another hosted Git service). That way your friend can first check if you've pushed any changes and **pull** them down to their local computer so they have the freshest copy before editing.

Git is an extremely powerful tool whether you're working alone or on a very large team, and most developers writing code today need to learn and use Git on a daily basis.

OPEN SOURCE CONTRIBUTIONS

Open source software generally means code that is open to the public, and anyone is welcome to propose changes or enhancements to it. (Keep in mind, though, that just because open source software is open to the public does not mean it is not subject to proper licensing and copyright laws.)

Most of the world's open source code today lives under a service called GitHub, which not only hosts code but also allows people to manage and moderate the code repositories. Having an active GitHub account with relevant open source contributions in your field of work has become increasingly more attractive to prospective employers and is slowly becoming a standard measurement for the quality of an engineer.

Why Make Code Open Source?

This question is occasionally brought up by those who are new to coding or not used to the direction the industry is moving toward. It's a fair question — why in the world would companies or even individual developers share their code with the rest of the world? The answer can be reduced to one word: **collaboration**.

The quality of a project can only increase when more people who care about it start putting their heads together to improve it. When more people are involved in a library's development, more use cases are brought to light, and the code becomes more robust over time as it starts getting used by lots of people around the world.

Ultimately, the pros of open sourcing code far outweigh the cons, but even with that in mind, open source is still not for everyone. The policies of certain corporations, and even the laws governing some public institutions, still forbid code from being open sourced, and that's okay.

5 Easy Steps for Contributing

If you're ready to start getting noticed by certain communities you care about, and even prospective employers, then it's time to start contributing to open source! If you're just getting started writing software and still aren't sure of your skill level, contributing to open source projects is a great way to learn how to collaborate with other developers and will improve your code as it gets critiqued by potentially tens if not hundreds of other developers. This might sound a little intimidating, but don't let that stop you — most communities are very understanding and welcoming to beginners, as long as you abide by the rules.

Now let's go over the unwritten law of the land in the open source world. Pretend you were using an open source library and found a bug in it. Usually the flow of contributing to open source code works like so:

1. Read the CONTRIBUTING file
Some open source projects have a file at the root directory called `CONTRIBUTING.md`. This file usually contains instructions for preliminary steps you need to take before becoming a contributor.

In this file, they might ask you for a certain convention on how to file bugs, etc. Be sure to read this file and follow the instructions. Here are a couple of examples of the `CONTRIBUTING` file in the jQuery and Angular repositories.

2. File a bug in the project's repository
At this point, this process varies depending on what was requested by the project's authors in the `CONTRIBUTING` file. But in general, the next step to contributing to an open source project is to file a bug in the project's repository.

This is where GitHub really shines, as it makes it really easy for anyone to communicate with the authors of the library in question. There, you would "open an issue," and notify the authors of the bug you found and what steps they must take to reproduce the bug.

The more information you write about the issue, the better. So if you found what part of the code is causing the bug in the project's codebase, be sure to let them know where the problem is and that you would be happy to fix it for them by opening a pull request. Most authors will welcome a pull request, but it is generally considered good courtesy to wait for the author's feedback in the issue before opening the pull request.

3. Forking the code
While you are waiting for the author's feedback on the issue you opened, there is nothing wrong with forking the project and starting to work on a fix for the problem on your version of the code.

Forking means you are making a copy of the project to your GitHub account so you can freely make changes before submitting it back to the original codebase. The process of submitting it back to the original codebase is called a pull request, which basically means you are asking the author's permission to merge the changes you made into their original codebase.

Once you are done making the changes on your version of the repo, be sure to look for any special instructions the author might have added in the `CONTRIBUT-ING.md` for commit messages. If there are special instructions, then follow their conventions. If not, then a couple general guidelines for good commit messages are to be descriptive about the changes you made, and make the least amount of commits possible. Some libraries will even require you to squash all your changes into one single commit before opening a pull request.

Also, if the library has unit tests, be sure to run them and ensure everything is running properly before submitting your pull request. If you added any new features that might require new tests, be sure to write those tests and commit them into the project as well.

4. Making a pull request
Once you're done and proud of your code, and the author has given you permission to open the pull request, push your code to your remote GitHub repository and open the pull request. Much like opening an issue, be sure to follow any guidelines in `#ID-OF-THE-ISSUE` before opening the pull request. A good guideline when opening pull requests is to mention the ID of the issue that is getting resolved with this pull request. This is as easy as adding the #ID-OF-THE-ISSUE to your description. To find out what the ID of the issue is, simply find the issue in the list of issues, click on it, and look at the end of the URL. The ID will be the number at the end of the URL.

Once your pull request is open, it will be up to the library author to review your code and decide whether or not they are going to accept the changes you made. Usually if they see a problem with your code, they will let you know so you can address it. Don't worry if that happens — it's normal. Once you have addressed any problems they might have found and pushed all the latest changes, the author will be able to accept your pull request, causing your code to merge to the original codebase.

5. TA-DA!
Congrats! Once your code is accepted, you have officially become an open source contributor. But don't stop there — keep looking for more ways to help improve that library (and others) by proactively looking at their list of GitHub issues and offering to help fix some of the problems you see you. (Hint: Not everyone who opens an issue is looking to fix it themselves.)

What Projects to Start Contributing To

It's generally a good idea to start contributing to smaller projects. So, for example, instead of contributing to the jQuery source code, which gets a lot of attention and has been pretty stable for a few years, it might be a better idea to start with one of the community-backed plugins that were built on top of jQuery.

Those authors tend to get less attention and usually need a lot of help fixing bugs or writing new features for their codebases. Those smaller libraries also tend to not have as much of a process when contributing, so it's good to know some of the general unwritten rules laid out above when seeking to help them out.

You may be surprised to find some authors simply don't want any help, which is fine. In those cases, it's better to move on and look for other libraries that are genuinely looking for more help. Again, a great way to get to know an author is by simply opening an issue on their repo and getting a conversation started.

Sometimes you will also find great projects that have been abandoned. In those cases, feel free to message the authors and let them know you would like to either take over the project or simply help them manage issues and pull requests by other developers. Both of these solutions will help the open source project tremendously, as well as increase your experience as a developer by collaborating with other developers.

Contributing With Documentation

If contributing to a project's codebase still feels a little too intimidating, you might want to look into other ways of getting started — documentation, for instance. After all, how good is an open source library if nobody knows how to use it? Many libraries evolve quickly and their documentation often gets overlooked and quickly becomes outdated.

If you're looking to start contributing to open source projects and don't know how to start, consider helping a project with their documentation. Documentation is an essential part of libraries and is usually a low-hanging fruit for prospective contributors that want a foot in the door — not to mention, it helps the project immensely.

CHOOSING AN IDE
OR TEXT EDITOR

When it comes down to it, websites are just a bunch of code that displays text, images, and interactive controls in a web browser, and all that code needs to be written somewhere — but where?
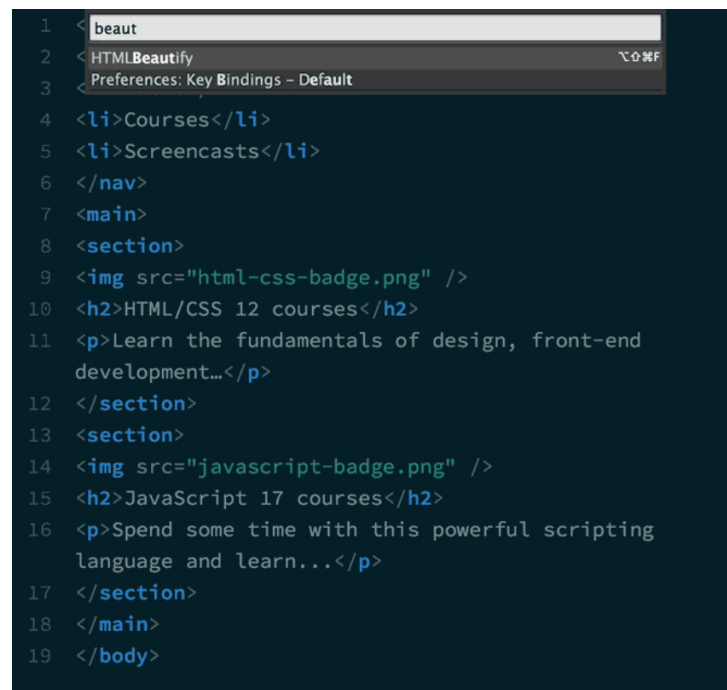
Since it's made up of letters, numbers, and symbols, the code for websites can really be written in any application that allows you to type text and save it as a file. On computers running Windows, that means you could build your entire site with Notepad. On macOS, you could use the built-in TextEdit program. And on Linux, you could use editors like nano or vim. That said, there are several other options that are built specifically with code writing in mind and offer useful features that the simple editors mentioned above do not.

Text Editors

Three popular text-based code editors are Sublime Text, Atom, and Visual Studio Code. Each of these offer features beyond just simple text editing, like standardizing the colors that are used to display different parts of code so you can easily understand code at a glance, powerful find and replace that lets you find all instances of a variable in a certain scope and just replace those, and multiple editing panels in a single window so you can view several related files at once (like HTML, CSS, and JavaScript files).

These editors also have a community-driven plugin system so they can be extended to perform all sorts of useful tasks. For example, Sublime Text 3 has a plugin called **HTMLBeautify**. Once you've installed it, you can run it from a key command or the menu bar and it will look at any HTML in an open editor window and format it based on a set of predefined rules, like standardizing tag indentation based on how they are nested or removing any empty lines between tags.

You can even configure different plugins to run every time you save, which can be great for running scripts that test your code and validate that you didn't introduce any syntax errors early — that way you can fix them before they get deployed on a server.

Code-specific text editors like the ones mentioned above provide a much needed service to programmers and make their lives better by saving time through enhanced productivity, fewer uncaught bugs, and customizability.

Integrated Development Environments

One level above code specific-text editors is a larger tool called an integrated development environment (or IDE). Examples of IDEs are Visual Studio for .NET applications, Eclipse for Java development, and Xcode for Swift and Objective-C development. IDEs usually combine code editors, debuggers, and built-in tools for compiling and running applications.

Tooling

Some IDEs also include platform-specific tools. For example, in Apple's Xcode IDE there's a tool called Interface Builder where you can lay out all the different screens in your app and build up your UI visually before connecting it up with code to dynamically change the data that's displayed while people are using the app.
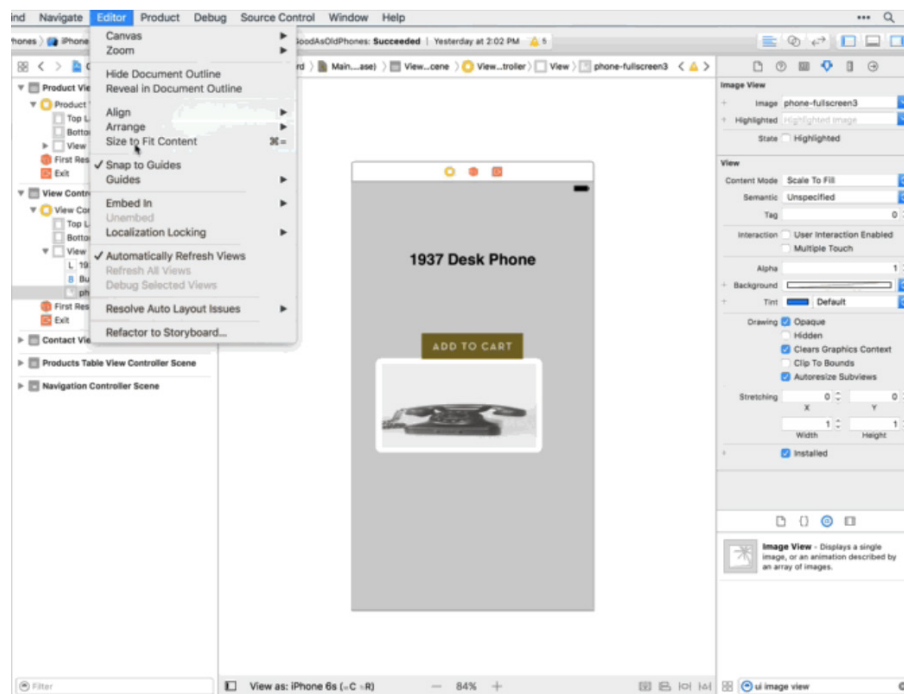
Code Suggestion

Another common feature between different IDEs is intelligent code suggestions. This is an essential feature when working with large frameworks like .NET and UIKit (iOS), but once you've experienced it, you'll likely want it for developing in any language or framework.

Debugging

Debuggers are programs that help you find bugs in your code while it's running so you can remove them before you release an application or website into the world. With the debugger, you can set **breakpoints** on certain lines of your code, and then whenever that code is executed, the IDE will pause the running application and let you inspect the value of all the variables at that moment in time — in other words, the **state** of your application. With any luck, you'll see something that looks off when you look at those values, and you'll be able to look back at your code and make a correction.

The best answer to the question, "What program should I use to write my code?" requires a lot of research and reflection, but the good news is that when you're getting started, the answer is as simple as, "Whatever you want." Then as your skills grow, you can take advantage of all that code-specific text editors and IDEs have to offer.