Web Architecture Overview

## **The Basics**

We can generally divide website 'stuff' into two parts. The client and the server. In a slightly simplified world the client is your browser and the server is some computer off in the far far away that has data that your client wants. It is more complex than that because there can be clients that request stuff from web servers that are not browsers and sometimes servers can request stuff from other servers. Sort of turning a server into a client temporarily. But the major concept in the same, one computer is asking for stuff and the other computer is serving that stuff.

## **The Client-Server Relationship**

### **The Client**

For the purposes of this class we can think of the client as the web browser. When we say that things happen client side we are referring to things that happen within the browser without the need to contact a server. So we can have pages that are interactive and we can have that all be done client side. An example of a client side interaction can be seen with the [jQuery UI Accordion](http://jqueryui.com/accordion/). When you click on a tab it will expand that tab and contract the rest. All of that logic and interaction is happening on the client with no need to communicate to the server. So we say it is a client side interaction or that it is happening client side.

### **The Server**

The server is a bit harder to pin down. To excessively simplify the server is the thing the client is requesting data from. In general this is probably going to be some computer elsewhere like when you are accessing http://news.google.com. But it can also be a server application you are accessing locally. In that case the address would look more likehttp://localhost:3000. localhost refers to accessing a resource on the same computer. Finally you can use your own computers file system as a server. It can't do anything special like change content on a page, it can only serve static files without changing them. It would look like file:///D:/Documents/hello-world.html. This is just directly accessing a file on your computer. The important thing to know is that other people (like the instructor and graders) can only access your page if your address looks like the first address. If it is being accessed as localhost or with file:// then only you can access it.

When stuff is server side then it has access to all the data on the server. So a weather site might have the most recent weather data on the server or a shopping website might have the most up to date inventory. When a page is generated on the server it can use this to generate the page that will be sent back to the client. But once it gets sent to the client there is nothing more the server can do to change the page unless the client makes another request.

# Web Languages

## **Intro**

There are a ton of different languages used across the web. This module will introduce a handful of them. Broadly speaking there are languages that deal with content and layout, client side interactions and server side interactions. We will talk about the language options for all of these.

## **Document Structure**

HTML stands for Hypertext Markup Language. When people think of the language used to write web pages, this is usually what comes to mind. But we need to be careful with what the purpose of HTML is. It defines the document structure. No more, no less. It is not responsible for what the page looks like. Browsers have conventions about how certain HTML elements should look but the job of HTML is not to decide the look of the page. It is to decide the structure and meaning of the content.

## **Page Look and Feel**

CSS stands for Cascading Style Sheets. This is the language that is used to actually modify the layout and styling of a page. HTML might define what is in a particular section but CSS will determine what that section looks like. CSS can change the spacing between things, the color of things or even make very simple interactivity possible. For example is is possible to change the color of something when you mouse over it using CSS. However there is not much more than that in terms of logic.

## **Client Side Interactivity**

When we talk about client side interactions we are pretty much exclusively talking about JavaScript. JavaScript is a language that has syntax that resembles the C language. It has all of the control structures one would expect to find in a robust programming language. However, the type and class system is going to be different than what you are used to.

We will be doing a lot of work with JavaScript in this class. It has a host of built in features that let it do things like interact with the structure of a page, change the styling of a page and make requests to servers to get data. One can have simple JavaScript that just changes the look here and there or entire fluid dynamics simulations can be written in JavaScript and rendered in the browser.

## **Server Side Languages**

So far it has seemed pretty clear. One task, one language. Server side is where it all goes off the rails. There are a huge number of languages that can be used, often with different parts of bigger servers written in different languages. PHP, C#, PHP, JavaScript, Ruby, Go, Python - These are just a few of the many different languages servers are programmed in. Some, like Python or JavaScript, are seldom used alone but are instead used with a framework for server side programming. In this class we will be using Node.js and JavaScript to program on the server. Django is a very popular framework that is used with Python. However some languages like PHP do not require a framework at all and have many of those features baked into the language itself.

# HTTP and TCP/IP

## **Intro**

HTTP stands for HyperText Transfer Protocol and TCP/IP stands for Transmission Control Protocol/Internet Protocol. Discussing how TCP/IP works is beyond the scope of this class. For now it can be thought of as something like the telephone system. You dial a number to connect to a server, then you connect to a port which is like an extension. If a program is waiting for that port to 'ring' then it will accept the transmission and start a conversation with the client who called.

HTTP is more fundamental to this class. It is more like that language that is spoken on the 'call' between the client and the server. We previously talked about how requests are made to servers and how servers respond to the requests. HTTP is what defines how that process works.

Within HTTP there are lots of different parts of the specification but for basic web interactions we only care about a handful of topics.

## **HTTP Request and Response**

There are two part to a usual browser/server web interaction. The request and the response. The request gets sent from the browser to the server. It is composed of two parts, the headers and the body. Often times the body is blank as there is no additional information that needs to get sent along. But if something like a form is submitted to the server by the client then that information is included in the request body sent by the client.

### **An Example Request**

#### **Request Headers**

Here is a sample request that is submitting a POST request to a page hosted at OSU. This first block of code are the request headers. This is where most of the magic happens when a client is communicating with a server.

POST /about HTTP/1.1

Host: main.oregonstate.edu

Connection: keep-alive

Content-Length: 128

Cache-Control: max-age=0

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,\*/\*;q=0.8

Origin: http://main.oregonstate.edu

Upgrade-Insecure-Requests: 1

User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/44.0.2403.157 Safari/537.36

Content-Type: application/x-www-form-urlencoded

Referer: http://main.oregonstate.edu/about

Accept-Encoding: gzip, deflate

Accept-Language: en-US,en;q=0.8

Cookie: mp\_fe42a3507c097e9a9d1e9f881d833cfb\_mixpanel=%7B%22distinct\_id%22%3A%20%2214af9e5614bf2-08c9f53fa-63161675-1fa400-14af9e5614cb07%22%2C%22%24initial\_referrer%22%3A%20%22%24direct%22%2C%22%24initial\_referring\_domain%22%3A%20%22%24direct%22%7D; \_\_utma=26995683.1248642055.1411404198.1440290989.1440561171.61; \_\_utmz=26995683.1440561171.61.45.utmcsr=infosu.oregonstate.edu|utmccn=(referral)|utmcmd=referral|utmcct=/; has\_js=1; \_ga=GA1.2.1248642055.1411404198

This is a very typical looking request header and we are going to ignore most of the fields in here for now. However, there are a few that we really want to pay attention to.

**The first line**

This specifies the request type (often a GET or a POST), the resource being requested (in this case the /about page) and the version of HTTP being used.

**Host**

This is who the client is sending the request to.

**Accept**

This is the type of file the client is OK getting back from the server. They are listed in order of preference. So here it is saying it wants a text file containing HTML. If it can't have that application specific xhtml and xml is OK and so on. Until it hits \*/\* which means any file will work. There are a TON on content types that can be listed here.

**Cookie**

This is information, stored on the client that is sent to the server with every request. It can be used to keep track of which client is which when several clients are interacting with a server. It is important to know that browsers are sending this with pretty much every HTTP request.

**Referer**

This is the page the client came from. It can be occasionally useful for things like analytics but it can be fooled so it is not good to rely on it for important tasks. Referer is the actual spelling in this context.

#### **Request Body**

In this case I search for the term "foobar" so it can be seen in the data sent in the request body seen below. In addition there was quite a bit of other data sent along in the body. You will often be inspecting the request body to help debug issues when dealing with HTML forms or AJAX requests.

This is a standard way to format forms that browsers and many JavaScript libraries will take care of automatically. Key value pairs will be in the form of key=value and they will be separated by & symbols.

term=foobar&op=Search&group\_path=&form\_build\_id=form-i4fdI34k9NbfdPoXctJDBbagH31-y2MkKhTl0ZS-u4M&form\_id=osu\_search\_top\_hat\_form

It may be somewhat surprising that the body is so much smaller than the headers. We often think of headers as being a summary. And usually this would be correct. But often the request is very simple so there is no need for a substantive body, there are exceptions, like when uploading a file in which case the body is packed with information.

### **An Example Response**

Here is the response that the server gives back to the above request.

HTTP/1.1 302 Found

Server: Apache/2.2.15 (CentOS)

X-Powered-By: PHP/5.4.45

X-Drupal-Cache: MISS

Cache-Control: no-cache, must-revalidate, post-check=0, pre-check=0

Content-Language: en

Location: http://main.oregonstate.edu/search/osu/foobar/0/3/

Content-Type: text/html; charset=utf-8

Content-Length: 18

Accept-Ranges: bytes

Date: Tue, 08 Sep 2015 23:58:05 GMT

X-Varnish: 305863683

Age: 0

Via: 1.1 varnish

Connection: close

X-Varnish-Cache: MISS

Lets take a look at the important pieces of this response.

**The Fist Line**

The first line of the response has probably the most important piece of information. The HTTP status code. In this case 302 (which Drupal misuses so signify that a redirect happened). You can read about the various codes in the [Wikipedia entry](https://en.wikipedia.org/wiki/List_of_HTTP_status_codes) on HTTP status codes. Generally speaking values strictly less than 400 are OK and values of 400 or greater are a problem.

**Location**

This is where you actually end up after the server does its thing. In this case the address that shows up in the address bar is not the same one the link pointed to when we did the search. The server generated a page and sent us there after we ran the search. This is basically the sign post that lets the browser know what to display in the address bar.

**Content-Type**

This lets the browser know what kind of data was sent back. This is how the browser knows if it should try to parse HTML, just display plain text or render an image.

**Content-Length**

This is how much data was sent back in the response body. At times this can be tremendously helpful with debugging. Sometimes weird bugs will crop up and it is because hidden characters that the browser can't display get sent back. You can use this to compare how much data you expected to get sent back with how much was actually sent back.

Next lets look at the response body. As it turns out, the first thing we got back was the 302 response which was basically an instruction for the browser to go look elsewhere for the information it needed. So the response body was empty. But, we can look at the response body of the place we ended up at.

<!DOCTYPE HTML>

<!--[if lte IE 7]> <html lang="en" class="ie ie7 classic"> <![endif]-->

<!--[if IE 8]> <html lang="en" class="ie ie8 classic"> <![endif]-->

<!--[if gt IE 8]><!--><html class="classic" lang="en"><!--<![endif]-->

<head profile="http://www.w3.org/1999/xhtml/vocab">

...

</div> <!-- /page-wrapper -->

</body>

</html>

In this case the response body was the HTML content of the page we asked for. It is heavily abbreviated above so that it could fit easily on this page. There is noting terribly unique about it. It is the same thing you would see if you opened the file in a text editor. The response body is usually just the content of a file. Nothing more, nothing less.

## **The Life-cycle of a Web Page**

You have probably requests tens of thousands of web pages but may not have considered what is actually involved in getting that page to your computer and displaying it so lets break it down.

### **User Clicks a Link**

This one is pretty straight forward. You click on a button. This fires off an even that the browser responds to.

### **Browser Sends a Request**

Your browser gets together a fair amount of information and sends it off to the server in a request. This request includes a header that includes the request method, the identifier of the thing being requested, what format the client wants the data back in, information about the browser and page state and a bunch of other stuff. It can also include a body with yet more information like stuff to upload or more detailed information about the request.

### **Server Responds to the Request**

After getting routed from server to server, eventually the request ends up at its destination. At this point the server looks for the requested resource, probably on its hard drives somewhere. It might return the thing directly or it might modify it then return it or it might build the entire page on-the-fly and return it based on the state of the server or the contents of the request.

### **Client Receives the Response**

The client gets the response and parses it. It may render it directly or it might change how it renders it based on logic included in the file it got back. Additionally the file it got back might tell the browser it needs yet more files (for example, images to display in the page) and this will cause the browser to fire off more requests.

### **Client Renders the Page**

Finally as the requests finish it will start rendering the page. Depending on the speed of the connection and size of the requests page it may render it instantly or you may see things render one at a time. In any case, once it all finishes the server can do nothing else to influence the page. However, the browser now can make changes, like causing a drop down menu to drop down when you hover over it or change the color of a link when you hover over it. At some point you, the user, will do something that will cause this whole process to kick off again and it will clear the current page and go through the process again to load a new page

# Intro to Git and Version Control

## **Intro**

So far, when you have worked on software you probably have been editing a file, saving it, compiling it and moving on when it worked. If you are lucky you might have had no real problems doing this. But at some point, you are going decide that some decision you made 5 saves back was a very wrong one. Or you are going to want to go though and edit all your code to make it cleaner, but you know this runs the risk of breaking something and don't want to take that risk. Version control can help solve a lot of these problems.

## **Basics of Git**

There is a lot of depth one can go into when talking about version control. Git is just one tool to implement version control. Other popular options are SVN and Mercurial. However, Git is what we will be using for this class and we will only be scratching the surface.

There are a few terms that will be used throughout the class when talking about assignment deliverables or in videos showing off what Git does. Here are the most important terms and their definitions:

### **Master**

Master is the main branch of your repository. It should represent your software in its most advanced, complete state. So the most recent commit in Master should always represent the most complete version of your software that you deem to be correct and functional.

### **Branch**

Often times you need to do work that will either change or temporarily break your software while you make changes. A branch is a separate version of your software project. It often branches off of Main (but can branch off of other branches) and when it does, it basically takes a snapshot of Main when is branches off. Any changes you make will only exist in the branch until you merge them back into main.

### **Commit**

A commit is sort of like a snapshot of your workspace. Changes are not saved to the repository until they are added and committed. GitHub desktop automates the adding, but be aware if you are using a different tool, adding and committing are two different steps. Every time you make a commit you will be adding a snapshot in time you can branch off of or roll back to if needed. Committing is probably the most common operation in Git.

### **Merge**

Merging is the process of taking a branch and applying those changes to the thing it branched off of. Often times this is really straight forward. Git is really smart when it comes to merging. It can tell which lines were changed and where they belong in the file even if the Master has been changed after you made a branch. The only time things get messy is if changes were made to the same line of code in both the branch you are merging and the branch you are merging into. In this case it will ask you which version is right.

### **Clone**

Cloning makes a copy of a repository. When the copy is made generally all the branches are cloned as well. You will clone your repository on GitHub to your local computer to do work or you may end up cloning repositories I make to get access to things like automated test suites. The repo you cloned from is referred to as origin.

### **Push**

Pushing is sort of the opposite of cloning. After you clone you often want to get your changes back to the place you cloned from. To do this you push to that repository. Unlike cloning, when you push, you only push a single branch to a single branch. Often you are pushing your current branch to origin's master branch.

### **Pull-Request**

GitHub is a tool that is separate from Git. It is sort of like a wrapper that adds additional features. Its main purpose is to help people work on and share repositories from remote locations. At an office, there is often one repo on the network that everyone can clone and push work to easily. It gets more difficult when everything is not on the same network. So a feature that GitHub created is the pull-request. If I make a change that I think improves software, I can make a pull request. This basically tells the owner of the repository that I want them to pull my changes into their repository. I usually don't use this feature as I am often the only one contributing to my projects, but it is the work-flow that GitHub desktop uses so it is worth mentioning here. Once the repo owner authorizes the pull request it works like a merge.

[◀](http://eloquentjavascript.net/11_language.html) [◆](http://eloquentjavascript.net/index.html) [▶](http://eloquentjavascript.net/13_dom.html)

# Chapter 12

# JavaScript and the Browser

The browser is a really hostile programming environment.

Douglas Crockford, The JavaScript Programming Language (video lecture)

The next part of this book will talk about web browsers. Without web browsers, there would be no JavaScript. And even if there were, no one would ever have paid any attention to it.

Web technology has, from the start, been decentralized, not just technically but also in the way it has evolved. Various browser vendors have added new functionality in ad hoc and sometimes poorly thought out ways, which then sometimes ended up being adopted by others and finally set down as a standard.

This is both a blessing and a curse. On the one hand, it is empowering to not have a central party control a system but have it be improved by various parties working in loose collaboration (or, occasionally, open hostility). On the other hand, the haphazard way in which the Web was developed means that the resulting system is not exactly a shining example of internal consistency. In fact, some parts of it are downright messy and confusing.

## Networks and the Internet

Computer networks have been around since the 1950s. If you put cables between two or more computers and allow them to send data back and forth through these cables, you can do all kinds of wonderful things.

If connecting two machines in the same building allows us to do wonderful things, connecting machines all over the planet should be even better. The technology to start implementing this vision was developed in the 1980s, and the resulting network is called the Internet. It has lived up to its promise.

A computer can use this network to spew bits at another computer. For any effective communication to arise out of this bit-spewing, the computers at both ends must know what the bits are supposed to represent. The meaning of any given sequence of bits depends entirely on the kind of thing that it is trying to express and on the encoding mechanism used.

A network protocol describes a style of communication over a network. There are protocols for sending email, for fetching email, for sharing files, or even for controlling computers that happen to be infected by malicious software.

For example, a simple chat protocol might consist of one computer sending the bits that represent the text “CHAT?” to another machine and the other responding with “OK!” to confirm that it understands the protocol. They can then proceed to send each other strings of text, read the text sent by the other from the network, and display whatever they receive on their screens.

Most protocols are built on top of other protocols. Our example chat protocol treats the network as a streamlike device into which you can put bits and have them arrive at the correct destination in the correct order. Ensuring those things is already a rather difficult technical problem.

The Transmission Control Protocol (TCP) is a protocol that solves this problem. All Internet-connected devices “speak” it, and most communication on the Internet is built on top of it.

A TCP connection works as follows: one computer must be waiting, orlistening, for other computers to start talking to it. To be able to listen for different kinds of communication at the same time on a single machine, each listener has a number (called a port) associated with it. Most protocols specify which port should be used by default. For example, when we want to send an email using the SMTP protocol, the machine through which we send it is expected to be listening on port 25.

Another computer can then establish a connection by connecting to the target machine using the correct port number. If the target machine can be reached and is listening on that port, the connection is successfully created. The listening computer is called the server, and the connecting computer is called the client.

Such a connection acts as a two-way pipe through which bits can flow—the machines on both ends can put data into it. Once the bits are successfully transmitted, they can be read out again by the machine on the other side. This is a convenient model. You could say that TCP provides an abstraction of the network.

## The Web

The World Wide Web (not to be confused with the Internet as a whole) is a set of protocols and formats that allow us to visit web pages in a browser. The “Web” part in the name refers to the fact that such pages can easily link to each other, thus connecting into a huge mesh that users can move through.

To add content to the Web, all you need to do is connect a machine to the Internet, and have it listen on port 80, using the Hypertext Transfer Protocol(HTTP). This protocol allows other computers to request documents over the network.

Each document on the Web is named by a Uniform Resource Locator (URL), which looks something like this:

http://eloquentjavascript.net/12\_browser.html

| | | |

protocol server path

The first part tells us that this URL uses the HTTP protocol (as opposed to, for example, encrypted HTTP, which would be https://). Then comes the part that identifies which server we are requesting the document from. Last is a path string that identifies the specific document (or resource) we are interested in.

Each machine connected to the Internet gets a unique IP address, which looks something like 37.187.37.82. You can use these directly as the server part of a URL. But lists of more or less random numbers are hard to remember and awkward to type, so you can instead register a domain name to point toward a specific machine or set of machines. I registered eloquentjavascript.net to point at the IP address of a machine I control and can thus use that domain name to serve web pages.

If you type the previous URL into your browser’s address bar, it will try to retrieve and display the document at that URL. First, your browser has to find out what address eloquentjavascript.net refers to. Then, using the HTTP protocol, it makes a connection to the server at that address and asks for the resource /12\_browser.html.

We will take a closer look at the HTTP protocol in [Chapter 17](http://eloquentjavascript.net/17_http.html#http).

## HTML

HTML, which stands for Hypertext Markup Language, is the document format used for web pages. An HTML document contains text, as well as tagsthat give structure to the text, describing things such as links, paragraphs, and headings.

A simple HTML document looks like this:

edit & run code by clicking it

<!doctype html>

<html>

<head>

<title>My home page</title>

</head>

<body>

<h1>My home page</h1>

<p>Hello, I am Marijn and this is my home page.</p>

<p>I also wrote a book! Read it

<a href="http://eloquentjavascript.net">here</a>.</p>

</body>

</html>

The tags, wrapped in angle brackets (< and >), provide information about the structure of the document. The other text is just plain text.

The document starts with <!doctype html>, which tells the browser to interpret it as modern HTML, as opposed to various dialects that were in use in the past.

HTML documents have a head and a body. The head contains informationabout the document, and the body contains the document itself. In this case, we first declared that the title of this document is “My home page” and then gave a document containing a heading (<h1>, meaning “heading 1”—<h2> to<h6> produce more minor headings) and two paragraphs (<p>).

Tags come in several forms. An element, such as the body, a paragraph, or a link, is started by an opening tag like <p> and ended by a closing tag like</p>. Some opening tags, such as the one for the link (<a>), contain extra information in the form of name="value" pairs. These are called attributes. In this case, the destination of the link is indicated withhref="http://eloquentjavascript.net", where href stands for “hypertext reference”.

Some kinds of tags do not enclose anything and thus do not need to be closed. An example of this would be <img src="http://example.com/image.jpg">, which will display the image found at the given source URL.

To be able to include angle brackets in the text of a document, even though they have a special meaning in HTML, yet another form of special notation has to be introduced. A plain opening angle bracket is written as &lt; (“less than”), and a closing bracket is written as &gt; (“greater than”). In HTML, an ampersand (&) character followed by a word and a semicolon (;) is called anentity, and will be replaced by the character it encodes.

This is analogous to the way backslashes are used in JavaScript strings. Since this mechanism gives ampersand characters a special meaning, too, those need to be escaped as &amp;. Inside an attribute, which is wrapped in double quotes,&quot; can be used to insert an actual quote character.

HTML is parsed in a remarkably error-tolerant way. When tags that should be there are missing, the browser reconstructs them. The way in which this is done has been standardized, and you can rely on all modern browsers to do it in the same way.

The following document will be treated just like the one shown previously:

<!doctype html>

<title>My home page</title>

<h1>My home page</h1>

<p>Hello, I am Marijn and this is my home page.

<p>I also wrote a book! Read it

<a href=http://eloquentjavascript.net>here</a>.

The <html>, <head>, and <body> tags are gone completely. The browser knows that <title> belongs in a head, and that <h1> in a body. Furthermore, I am no longer explicitly closing the paragraphs since opening a new paragraph or ending the document will close them implicitly. The quotes around the link target are also gone.

This book will usually omit the <html>, <head>, and <body> tags from examples to keep them short and free of clutter. But I will close tags and include quotes around attributes.

I will also usually omit the doctype. This is not to be taken as an encouragement to omit doctype declarations. Browsers will often do ridiculous things when you forget them. You should consider doctypes implicitly present in examples, even when they are not actually shown in the text.

## HTML and JavaScript

In the context of this book, the most important HTML tag is <script>. This tag allows us to include a piece of JavaScript in a document.

<h1>Testing alert</h1>

<script>alert("hello!");</script>

Such a script will run as soon as its <script> tag is encountered as the browser reads the HTML. The page shown earlier will pop up an alert dialog when opened.

Including large programs directly in HTML documents is often impractical. The <script> tag can be given an src attribute in order to fetch a script file (a text file containing a JavaScript program) from a URL.

<h1>Testing alert</h1>

<script src="code/hello.js"></script>

The code/hello.js file included here contains the same simple program,alert("hello!"). When an HTML page references other URLs as part of itself, for example an image file or a script—web browsers will retrieve them immediately and include them in the page.

A script tag must always be closed with </script>, even if it refers to a script file and doesn’t contain any code. If you forget this, the rest of the page will be interpreted as part of the script.

Some attributes can also contain a JavaScript program. The <button> tag shown next (which shows up as a button) has an onclick attribute, whose content will be run whenever the button is clicked.

<button onclick="alert('Boom!');">DO NOT PRESS</button>

Note that I had to use single quotes for the string in the onclick attribute because double quotes are already used to quote the whole attribute. I could also have used &quot;, but that’d make the program harder to read.

## In the sandbox

Running programs downloaded from the Internet is potentially dangerous. You do not know much about the people behind most sites you visit, and they do not necessarily mean well. Running programs by people who do not mean well is how you get your computer infected by viruses, your data stolen, and your accounts hacked.

Yet the attraction of the Web is that you can surf it without necessarily trusting all the pages you visit. This is why browsers severely limit the things a JavaScript program may do: it can’t look at the files on your computer or modify anything not related to the web page it was embedded in.

Isolating a programming environment in this way is called sandboxing, the idea being that the program is harmlessly playing in a sandbox. But you should imagine this particular kind of sandbox as having a cage of thick steel bars over it, which makes it somewhat different from your typical playground sandbox.

The hard part of sandboxing is allowing the programs enough room to be useful yet at the same time restricting them from doing anything dangerous. Lots of useful functionality, such as communicating with other servers or reading the content of the copy-paste clipboard, can also be used to do problematic, privacy-invading things.

Every now and then, someone comes up with a new way to circumvent the limitations of a browser and do something harmful, ranging from leaking minor private information to taking over the whole machine that the browser runs on. The browser developers respond by fixing the hole, and all is well again—that is, until the next problem is discovered, and hopefully publicized, rather than secretly exploited by some government or mafia.

## Compatibility and the browser wars

In the early stages of the Web, a browser called Mosaic dominated the market. After a few years, the balance had shifted to Netscape, which was then, in turn, largely supplanted by Microsoft’s Internet Explorer. At any point where a single browser was dominant, that browser’s vendor would feel entitled to unilaterally invent new features for the Web. Since most users used the same browser, websites would simply start using those features—never mind the other browsers.

This was the dark age of compatibility, often called the browser wars. Web developers were left with not one unified Web but two or three incompatible platforms. To make things worse, the browsers in use around 2003 were all full of bugs, and of course the bugs were different for each browser. Life was hard for people writing web pages.

Mozilla Firefox, a not-for-profit offshoot of Netscape, challenged Internet Explorer’s hegemony in the late 2000s. Because Microsoft was not particularly interested in staying competitive at the time, Firefox took quite a chunk of market share away from it. Around the same time, Google introduced its Chrome browser, and Apple’s Safari browser gained popularity, leading to a situation where there were four major players, rather than one.

The new players had a more serious attitude toward standards and better engineering practices, leading to less incompatibility and fewer bugs. Microsoft, seeing its market share crumble, came around and adopted these attitudes. If you are starting to learn web development today, consider yourself lucky. The latest versions of the major browsers behave quite uniformly and have relatively few bugs.

That is not to say that the situation is perfect just yet. Some of the people using the Web are, for reasons of inertia or corporate policy, stuck with very old browsers. Until those browsers die out entirely, writing websites that work for them will require a lot of arcane knowledge about their shortcomings and quirks. This book is not about those quirks. Rather, it aims to present the modern, sane style of web programming.

[◀](http://eloquentjavascript.net/11_language.html) [◆](http://eloquentjavascript.net/index.html) [▶](http://eloquentjavascript.net/13_dom.html)