**How do web servers work?**

What happens when you enter in the address field of your browser the [URL](http://www.aprelium.com/data/doc/2/abyssws-linux-doc-html/glossary.html#XURL) http://www.aprelium.com/doc/sample.html?

First, the browser slices the URL in 3 parts:

* http://: This part indicates that the document you want to access can be retrieved from web server, which understands the [HTTP](http://www.aprelium.com/data/doc/2/abyssws-linux-doc-html/glossary.html#XHTTP) protocol. The HTTP protocol is a standardized language of communication between browsers and web servers.
* www.aprelium.com: This is the host name of the computer from which the document can be downloaded.
* /doc/sample.html: This is the virtual path of the document in the www.aprelium.com's web server.

Then, the browser contacts a DNS (Domain Name Server) to know the IP address of the computer which full qualified domain name is www.aprelium.com. The domain name server is usually run by your ISP or by your company.

The browser establishes a connection channel with the web server on the computer which IP address was given by the DNS server and requests the document on the host which name is www.aprelium.com and which virtual path is doc/sample.html. The browser has to specify in the request the host name because many modern web servers (including Abyss Web Server) have the ability to serve more than a one host from a single computer with a single IP address only. This is called *virtual hosting*. In such a case, the IP address of this computer is associated with more than one domain name.

The server decodes the request and maps the virtual path to a real one, which should match an existing file. The server sends the file to the browser with some useful information such as its last modification time and its MIME type. The MIME type helps the browser decide how to display the received document. In our example, it is a HTML file. So the server sets its MIME type to text/html and the browser understands that it must render it as text.

Sometimes you enter a URL without an explicit filename such as http://www.aprelium.com/doc. The browser sends the request to the web server as in the previous example. The server detects that the virtual path maps to a directory and not to a file. It searches then in this directory an index file. Index files are usually named index.html or index.htm. If it finds for example index.html, it acts as if the requested URL was http://www.aprelium.com/doc/index.html. If no index file is found, the web server generates a listing of the directory contents and sends it to the browser or reports an error.

**Setting up a web site**

Using your favorite web pages editor (also known as HTML editor), create a web site and put its files in the directory <Abyss Web Server directory>/htdocs. The index files must be called index.html or index.htm.

Other computers on the network can access your web site if they browse http://<your host name>:<host port>.

<your host name> is the host name of your computer or its IP address. Ask your network's administrator for this information. If you want to connect to the server from the computer it runs on, you can also use 127.0.0.1, ::1, or localhost.

<host port> is the number of the port the server waits for connections on. It is printed on the server's window or terminal. If <host port> is 80 (which is the default), it can be omitted from the URL and the web server can be accessed with onlyhttp://<your host name>.

Put simply, HTTPis the protocol that allows Web browsers and servers to communicate. It forms the basis of what a Web server must do to perform its most basic operations.

When discussing how a Web server works, it is not enough to simply outline a diagram of how low-level network packets go in and out of a Web server.

HTTP started out as a very simple protocol, and even though it has had numerous enhancements, it is still relatively simple. As with other standard Internet protocols, control information is passed as plain text via a TCP connection.

In fact, HTTP connections can actually be made using standard "telnet" commands.

For example:

/home/chughes > telnet www.extropia 80

GET /index.html HTTP/1.0

<- Extra char return needed

Note that port 80 is the default port a Web server "listens" on for connections.

In response to this HTTP GET command, the Web server returns to us the page "index.html" across the telnet session, and then closes the connection to signify the end of the document.

The following is part of the sample response:

<HTML>

<HEAD>

<TITLE>eXtropia Homepage</TITLE>

[...]

</HEAD>

</HTML>

But this simple request/response protocol was quickly outgrown, and it wasn't long before HTTP was refined into a more complex protocol (currently version 1.1). Perhaps the greatest change in HTTP/1.1 is its support for persistent connections.

In HTTP/1.0, a connection must to be made to the Web server for each object the browser wishes to download. Many Web pages are very graphic intensive, which means that in addition to downloading the base HTML page (or frames), the browser must also retrieve a number of images. Many of them may actually be quite small and merely sliced up to provide some hard-coded formatting framework to the rest of the HTML page.

Establishing a connection for each one is wasteful, as several network packets have to be exchanged between the Web browser and Web server before the image data can ever start transmitting. In contrast, opening a single TCP connection that transmits the HTML document and then each image one-by-one is more efficient, as the negotiation of starting new TCP connections is eliminated.

Of course, it's all very well to define HTTP as the protocol for a browser to communicate with a Web server. However, there is more to a Web server than its function a communications protocol. Ultimately, a Web server serves up content.

When discussing how a Web server works, it is not enough to simply outline a diagram of how low-level network packets go in and out of a Web server.

This content must be identified in a way such that a Web browser can download and display that content in correctly. The primary mechanism for deciding how to display content is the MIME type header.

Multipurpose Internet Mail Extension (MIME)types tell a Web browser what sort of document is being sent. Such type identification is not limited to simple graphics or HTML.

In fact, more than 370 MIME types are distributed with the Apache Web server by default in the mime.types configuration file. And even this list does not represent the entire universe of possible MIME types! MIME types are distinguished using a type/subtype syntax associated with a file extension. Here is a brief snippet from an Apache mime.typesfile.

text/xml xml

video/mpeg mpeg mpg mpe

video/quicktime qt mov

From this, we can see that files containing MPEG video content end with file extensions such as mpeg, mpg, or mpe. So a file with the name "southpark.mpeg" would be served up as being an MPEG video file

**How does DNS work?**

**What is DNS? - DNS Flow Chart**

When you visit a domain such as dyn.com, your computer follows a series of steps to turn the human-readable web address into a machine-readable IP address. This happens every time you use a domain name, whether you are viewing websites, sending email or listening to Internet radio stations like Pandora.

Step 1: Request information

The process begins when you ask your computer to resolve a hostname, such as visiting http://dyn.com. The first place your computer looks is its local DNS cache, which stores information that your computer has recently retrieved.

If your computer doesn’t already know the answer, it needs to perform a DNS query to find out.

Step 2: Ask the recursive DNS servers

If the information is not stored locally, your computer queries (contacts) your ISP’s recursive DNS servers. These specialized computers perform the legwork of a DNS query on your behalf. Recursive servers have their own caches, so the process usually ends here and the information is returned to the user.

Step 3: Ask the root nameservers

If the recursive servers don’t have the answer, they query the root nameservers. A nameserver is a computer that answers questions about domain names, such as IP addresses. The thirteen root nameservers act as a kind of telephone switchboard for DNS. They don’t know the answer, but they can direct our query to someone that knows where to find it.

Step 4: Ask the TLD nameservers

The root nameservers will look at the first part of our request, reading from right to left — www.dyn.com — and direct our query to the Top-Level Domain (TLD) nameservers for .com. Each TLD, such as .com, .org, and .us, have their own set of nameservers, which act like a receptionist for each TLD. These servers don’t have the information we need, but they can refer us directly to the servers that do have the information.

Step 5: Ask the authoritative DNS servers

The TLD nameservers review the next part of our request — www.dyn.com — and direct our query to the nameservers responsible for this specific domain. These authoritative nameservers are responsible for knowing all the information about a specific domain, which are stored in DNS records. There are many types of records, which each contain a different kind of information. In this example, we want to know the IP address for www.dyndns.com, so we ask the authoritative nameserver for the Address Record (A).

Step 6: Retrieve the record

The recursive server retrieves the A record for dyn.com from the authoritative nameservers and stores the record in its local cache. If anyone else requests the host record for dyn.com, the recursive servers will already have the answer and will not need to go through the lookup process again. All records have a time-to-live value, which is like an expiration date. After a while, the recursive server will need to ask for a new copy of the record to make sure the information doesn’t become out-of-date.

Step 7: Receive the answer

Armed with the answer, recursive server returns the A record back to your computer. Your computer stores the record in its cache, reads the IP address from the record, then passes this information to your browser. The browser then opens a connection to the webserver and receives the website.

This entire process, from start to finish, takes only milliseconds to complete.

**How Do You Organize Web Servers for Performance?**

No matter how good your Web server is or how powerful a machine it is running on, there is always going to be a limit to the number of pages it can serve in a given time frame -- particularly if you are relying on a high percentage of dynamic content. Dynamic content typically relies on heavy database usage or processing of other program code, which takes up many server-side resources.

When discussing how a Web server works, it is not enough to simply outline a diagram of how low-level network packets go in and out of a Web server.

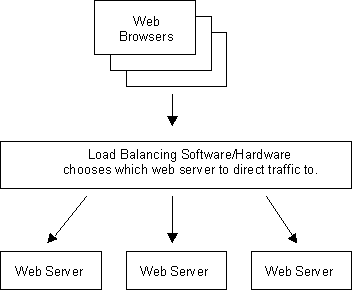
Another problem comes up when running a Web site that has grown popular beyond its immediate means of serving content and ways to spread this load out -- usually across multiple machines, and sometimes across multiple sites.

There are a number of ways to achieve load balancing. Perhaps the simplest way is to split the content across multiple hosts. For example, you could place all static HTML files on one host, all images on another, and have the third run all CGI scripts. Of course, this is a very crude form of load balancing and, depending on the content of the site, may have very little effect.

For example, if a single CGI script is causing a bottleneck for a Web site, moving it to a separate server helps only the HTML, images, and otherCGI scripts continue operating. Unfortunately, the heavily-loaded CGI script will still be a bottleneck for users who are utilizing that particular resource. Thus, load balancing on a Web server requires a bit more sophistication to figure out the right mix of where to migrate workload.

In other words, a number of factors must be worked out before deciding on a correct solution for load balancing. In particular, examining the access patterns for the site is crucial to the performance tuning and load balancing process.

The list below outlines some possible mechanisms used to spread the load among Web servers. We have included a brief description of these mechanisms to provide a sense of how the Web server ties into external infrastructure and the clients that may direct traffic to them. The figure below illustrates the concept of a load-balanced Web server farm.



DNS balancing (round-robin type)

Hardware load balancing

Software load balancing

Reverse proxying

Content spreading across hosts

Content spreading across outsourced providers

Load Balancing - DNS

DNS balancing is one of the easiest ways to create a Web site that can handle more hits. It basically involves having multiple copies of the site on separate physical servers. However, each server must be identical.

Then, the DNS server for the hostname of the site such as www.extropia.com is set up to return multiple IP addresses for the site. The DNS server can do this by either just returning more than one IP address for the hostname or returning a different IP address for each DNS request it receives.

Either way, what happens is a very basic distribution across the Web servers, although as far as the Web browsers are concerned there is only one Web site. This balancing is very basic, however, as it is difficult to determine to which IP address each client will resolve the site name. Also, since DNS query answers are essentially cached by the clients and other DNS servers, that single client will continue accessing the same Web server for the duration of the user's visit.

It is possible then that through a luck of the draw, heavy Web site users may get one IP address, and less-frequent Web site users tend to get another IP address. Thus, even with this load-balancing technique in effect, it is possible that the Web server belonging to the first IP address will be highly loaded, and the other one will be lightly loaded, rather than having the load spread evenly between the two.

Unfortunately the problems with this sort of "poor-man's load balancing" does not stop there. DNS Caches may not stay alive forever. So it is possible that a client, while using a Web site, may end up receiving a different IP address for the Web site. This can cause problems with dynamic sites, particularity ones that need to store data about the client.

As it is possible for a single client to hit more than one of the Web servers, this data needs to be shared across all of them. Depending on how complex the data is, this may be a nontrivial programming task to get the data shared in real-time amongst all the Web servers equally.

Load Balancing -- Software/Hardware

Software and hardware load balancing is similar to the DNS method just discussed, but rather than having the client attempting to access multiple IP addresses, only one is published. A machine is then set up to intercept HTTP requests to this one IP address and distribute them among the multiple servers hosting the Web site. Usually this distribution occurs at the level of TCP/IP routing which transparently maps a single source/destination IP address to a particular Web server in the Web farm.

This can be done with both hardware or software, with hardware solutions generally being more robust, but of course more expensive. The balancing this achieves is usually much better than the DNS method, as the load balancer can distribute the requests more evenly across the Web servers.

Also, these types of load balancers typically also occasionally detect when a Web server in the pool has gone down, and they can dynamically redirect the request to an identical Web server. With DNS load balancing, the client is stuck with a cached IP address of a downed Web server and cannot be redirected to a new one until the Web browser can request another IP address from the DNS server.

Load Balancing -- Reverse Proxying

Another quick-win method of reducing load on a Web site is to use a reverse proxy, which intercepts requests from clients and then proxies those requests on to the Web server, caching the response itself as it sends it back to the client.

This is useful because it means that for static content the proxy doesn't have to always contact the Web server, but can often serve the request from its own local cache. This in turn reduces the load on the Web server. This is especially the case when the Web server also serves dynamic content, since the Web server hardware can be less tuned to static content (when it is cached by a front-end proxy) and more tuned to serving dynamic content. It is also sometimes the case that although the Web server is serving dynamically created pages, these pages are cachable for a few seconds or maybe a few minutes. By using a reverse proxy, the serving of these pages speeds up dramatically.

Reverse proxying in this manner can also be used alongside the simple load balancing method mentioned earlier, where static and dynamic content are split across separate servers. Obviously the proxy would be used on only the static content Web server.

Load Balancing -- Distributing Content

Earlier, we talked briefly about distributing content among several Web servers and hard-coding their links. For example, if there is a bottleneck for bandwidth, the images on a Web site could be distributed across a couple servers (e.g., the documents for www.extropia.com could stay at www.extropia.com while the images would be referenced as image1.extropia.com and image2.extropia.com if they were split among two separate servers).

However, we also mentioned that this sort of "load balancing" is not really very dynamic and does not respond well to changing usage patterns that the previously discussed load balancing techniques deal with.

One recent entry into the market of load balancing sites with heavy content to download (such as images) are service providers that specialize in hosting images, sound, multimedia, and other large files for distribution. These service providers have purchased disk space at ISPs worldwide and rent space on all of those ISPs.

Then, these service providers use load balancing techniques that distribute the work of sending around the data files all over the world. Typically, these are more sophisticated load balancers that use a combination of DNS load balancing and software- hardware-based load balancing. They can tell where a user is geographically so that the images are served closest to the user. Thus, a user in Germany would get images stored on a German ISP, and a user in Hong Kong would get images served from a Hong Kong ISP.

How Does a Web Server Differ From an Application Server

We've already talked about what a Web server can do. But what about an application server? The distinction used to be quite clear. A Web server only served up HTML and images for viewing on a browser. And while an application could exist on the Web server, that application was typically restricted to just generating HTML and image data.

When discussing how a Web server works, it is not enough to simply outline a diagram of how low-level network packets go in and out of a Web server.

Likewise, in the beginning, the definition of an application serverwas fairly concrete. An application server merely contained raw business/application logic of an application and did not contain database or user interface code.

In many cases, the application server served as the middle-tier of three-tier programming. The figure below contains an illustration of the three-tier programming model.

In other words, an application server sits in the middle of other programs and serves to process data for those other programs. Usually, in the case of three-tier programming, the two layers that are separated by the application server is the User Interface layer and the Database/Data Storage layer.

Note that the concept of an application server should not be confused with Web applications sitting on a Web server itself. Web applications generally also contain application logic, but since they primarily serve to generate HTML for a Web browser, they are also user interface related and generally do not ever reside on a pure application server.

Data Marshalling

Data marshalling is a term used to refer to the way applications talk to each other. Similar to how Web servers wrap human-readable content in HTML to make it palatable to the eye, application servers wrap application-readable content inside other tags to allow the data to be interpreted by the receiving application. These "tags" formed the standards around which application servers were formed. For example, CORBA servers use a protocol called IIOP (Internet Inter-Orb Protocol) to transfer data between application objects. More recently, XML extensiblee Markup Language) has made data marshalling as easy as making up your own tags that are similar in syntax to HTML except that they describe data rather than how content should be displayed in a browser. More information about XML can be found in Selena Sol's XML tutorial.

In the past, most such application servers talked the language of data marshalling protocols such as IIOP for CORBA, Java's object serialization for RMI, and DCOM for remotely activating Microsoft ActiveX objects. However, the rise of XML (extensible markup language)as an internet friendly data marshalling language has blurred the boundaries.

Web Servers are turning into application servers that serve XML data alongside HTML data. Likewise, application servers are being marketed as being able to add value by having the capability of acting as a simple Web server while still delivering on it's core application server functionality.

Nearly all Java Enterprise Bean servers market the capability to simultaneously serve Java Servlets and Java Server Pages -- traditionally the realm of Web servers. Likewise, new data marshalling languages such as the Microsoft endorsed SOAP (Simple Object Access Protocol)XML standard have implementations that run in conjunction with existing Web servers.

So what should you use now that the distinction has become blurred? Web Server or Application Server?

The reality is that one size does not fit all. Typically, application servers are tuned for processing data using objects or code that represents application logic. Likewise, Web servers tend to be tuned to sending out data.

Web sitesic rule of thumb is that if you think of your Web site as, well, a Web site, then you should probably be using a Web server. Even if the data you are serving is dynamic, the fact that your site is Web centric means that all the configuration parameters of your Web server are best served tuning how you display information to other people's browsers.

For example, although it is easy to find an application server that can serve Web pages, you will be hard-pressed to provide one that supports Server-Side Includes (SSI) which is a feature nearly every Web server supports out of the box. Of course, you can still add application server components to a Web server if part of the data will be related to applications.

However, If you find that you are using application server components as the primary reason for your Web site, or that the Web server itself is being dragged down by all the resources that the application components may be using, then you should consider moving the application components to their own application server.

One thing we should mention is that breaking out application components of a Web site into a separate application server can help you in several ways. In some cases, breaking out the application components can increase both performance and stability of an application. However, you should be wary of doing this too early because the addition of yet another server in your environment can also add quite a bit of complexity to managing your infrastructure.

As we just mentioned, breaking out application components in this way also usually aids in performance. For example, Web servers are usually highly tuned to serve data from disk such as HTML pages and images very efficiently. Such a server is tuned to speed up IO operations. On the other hand, application objects are usually operating on pure logic alone -- they take data in from a stream, process it, and send new data back out again. This is a CPU rather than IO intensive activity. Thus, the application server is best served when it is tuned for CPU usage.

In addition, breaking out components usually adds to the stability of an application. Application servers are tested by their respective vendors to work in the context of executing application logic and so they are thoroughly debugged in that context. Likewise, Web servers are heavily tested within the context of serving documents either dynamically or statically. Mixing the two into one server can cause unexpected bugs that neither vendor has tested for.

Note that I use the word vendor loosely here since Apache has no "vendor" but rather is a community of open source developers who have arguably tested Apache beyond the scope that many other Web server vendors test their own products. However, even an Open Source community may have limits to the mix of environments that their products have been tested in.

Of course, these benefits should be tempered by what we mentioned earlier. If your Web site is mostly serving document data, it probably doesn't make much sense to rush out and purchase an application server. Indeed, if your environment is simple, you could be adding unnecessary complexity to the overall solution, thus maikng it harder to maintain and administrate -- unless you have devoted human resources to such an endeavor. As a rule of thumb, the more servers there, the more that must be maintained.

### The days of old - the browser object model

When I started with JavaScript everything was about the browser object model (BOM). The [DOM recommendation](http://www.w3.org/DOM/) was not quite finished yet and the lack of browser support stopped it from being a standard (to me the implementation and general market support makes something a standard, not that a group defines it as one).

When you were talking JavaScript, you talked about document.write() to produce content, forking code to be supported by browsers with document.all anddocument.layers, scripting of frames and interaction and opening of different browser windows with the window.open() method.

In order to access the content of the current document (or the one in a popup window, a frame or, in the case of Netscape, in a layer you had document.images,document.links and last but not least document.forms with its elements collection. You used the name attributes to access different elements likeforms.myform.surname.value and other abominations.

### First steps outside the HTML world

That said, JavaScript was already reaching further. One of the first things I did with it outside the browser was 1999. I helped other developers in my company (etoys back then, RIP) to work faster and not having to do the same work over and over again by customizing the company-approved editor: Allaire Homesite 4.0. Homesite had anApplication object that allowed you to access parts of the application itself, the document you are currently working on and all the others that are open. This allowed you to write quite powerful Macros in an easy manner using JavaScript. You also had access to the file system via the internal user dialogs.

In the browser world you also had bespoke implementations of JavaScript - first and foremost Microsoft's JScript to script Internet Explorer and access its chrome and other parts that weren't available to other browsers. If you renamed your HTML document from example.html to example.hta you even had a "HTML Application" which worked outside the browser's security model and gave you access to the file system to read out folders and file names.

### Enter the DOM and subsequently DOM Scripting

Across the browsers, the DOM got support starting with patchy implementations in IE5 and Opera and finally getting full-on support and developer tutorials in Netscape 6 based on Gecko as the rendering engine and Spidermonkey as the scripting engine. This lead to Phoenix, then to Firebird and finally to Firefox as we know it now.

In order to flag up a change in the JavaScript development world, "DOM Scripting" got coined as a term for writing "modern" JavaScript. Instead of relying on custom browser implementations our scripts started using the W3C standard DOM to access the document and gave us full access to anything that is inside HTML tags. You can read out the hierarchy of the document, traverse the element tree and access, alter or even create elements as and when you need them.

This power made a lot of the cumbersome browser hacks and workarounds obsolete and allowed us to write cleaner scripts. And just in time as malware exploits and annoying advertising practices and the software counter-strike (PopUp blockers) made it less and less of a good plan to rely on multiple windows or frames. DOM scripting allowed us to create much slicker interfaces in a clean manner.

### Ajax - breaking the rules

Then things got really wild when Ajax got its name. The use of XMLhttpRequest to create asynchronous calls was nothing new, but giving it a name and a methodology to follow worked out swimmingly.

Technically this was right up the alley of DOM scripting as the returned data in Ajax is XML, and you need the DOM or XSLT to convert this into data that can be displayed by a browser inside a HTML document. However, as we sent more and more complex information via Ajax, developers got bored of the cumbersome conversion and started to use responseText with the non-standard innerHTML to output data instead. This was so powerful and easy that innerHTML actually became part of the W3C recommendations.

This is where we are now, we use JavaScript a lot for all kind of Ajax applications that simulate rich client interfaces with CSS, JavaScript and HTML. And of course this leads to problems:

\* We break the normal use of browsers and have to fix things like browser history and bookmarking with hacks  
\* We run into more and more security problems, as every JavaScript on the page is running with the same authority and we do send user data over an easy to intercept channel.

### Fixing the back button

The first issue is quite known and we have to resort to hacks like fragment identifier updating and hidden iframes to seed the browser history (libraries make that easy for you, for example [the Yahoo User Interface Browser History Manager](https://developer.yahoo.com/yui/history/)).

If you look at it from a very pragmatic point of view though than it is a problem of the browser and shouldn't need our input. Ajax applications are a de-facto standard way of working and browsers should provide you with a technology hook to create them and keep a consistent application state.

This is happening right now, as both IE8 and the HTML 5 working drafts talk about a[hashchange event that would allow you to seed the browser history](http://blogs.pathf.com/agileajax/2008/03/ie8-html5-and-a.html), thus making bookmarking and the back button work without resorting to an iframe.

### Trying to add higher security

The security issues are a much bigger problem, especially when we think realistically and consider that we will build more and more modules for application frameworks and networks (like Facebook) than page/site based products in the future. There is not much you can do to sandbox JavaScript as even embedding in an iframe allows the script to modify the top.location property and redirect the current page to other sites without the user's consent.

One approach is to write a pre-processor or converter that checks JavaScript for sanity and filters out potentially dangerous code. One of those is [Caja](http://code.google.com/p/google-caja/).

The other option is to only allow a sub-set of JavaScript in third party code we implement in our systems. This is the approach of [AdSafe](http://www.adsafe.org/) and is much less overhead but also could mean limiting third party solutions.

### Broadening the horizon of JavaScript development

If we take off our blinkers and look around the whole developer world we find that there is a group of developers that have been writing JavaScript for a long time, came up with good best practices and very impressive code without ever touching a browser: Flash developers.

As ActionScript is executed within the Flash environment none of the browser quirks and problems appear (except for the back button and browser history and the solution is the same). This lead to quite an impressive group of developers that have a very pragmatic view of JavaScript. With Adobe now offering AIR as an application framework that allows you to write HTML/CSS/JavaScript applications in an installable manner we should start mixing the world of Flash development where everything just clicks (ok, not really, but much more than outside it) and the vast knowledge of how browsers fail and what security issues to avoid when writing JavaScript. What this will give us is truly portable applications and code and idea re-use the like we never imagined possible.

Other products that try to help us patch the browser are for example [Google gears](http://gears.google.com/), which allow you for example to have a local data storage and keep the interface of web applications very responsive even when there is a lot of calculation going on. The way Gears does that is by allowing multiple JavaScript threads that only do the heavy computation lifting whilst the main browser thread can go on keeping the interface in check.

### What will the future bring?

JavaScript is here to stay, it is easy to learn and provides instant satisfaction when trying it out. Browsers as we know them now will change drastically and maybe make way for more clever applications and frameworks that allow us to use the same technologies and knowledge without the pain and uncertainty. The skillset of JavaScript is a great one to have right now and we can only make it better by keeping an open mind to new ideas rather than dwelling on knowledge gained painfully in the past.