Learner guide

ICTWEB503 - Create web based programs

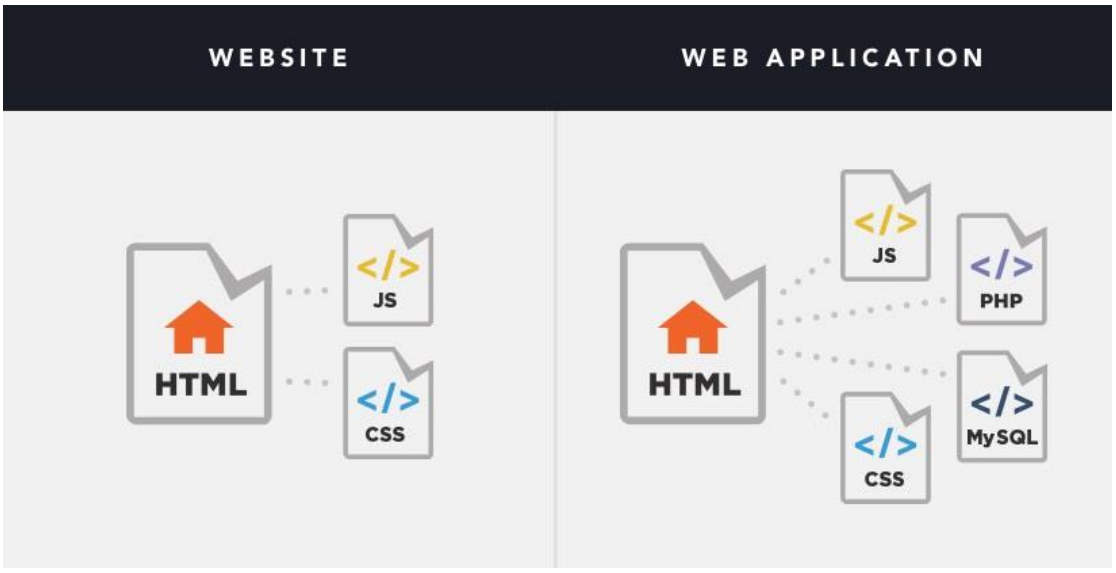
INSTITUTE OF TECHNOLOGY AUSTRALIA

Version Control

Unit code	Document version	Release date	Comments/actions
ICTWEB503	1.0	30.11.2017	First draft
ICTWEB503	2.0	25.7.2018	Second version

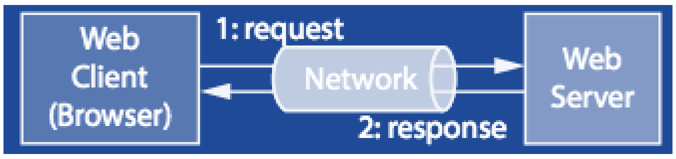
TABLE OF CONTENTS

CHAPTER 1: DETERMINE THE HYPERTEXT TRANSFER PROTOCOL (HTTP) AND ITS IMPLICATIONS WHEN DEVELOPING WEB APPLICATIONS



HTTP Protocol

The HyperText Transfer Protocol (HTTP) is the foundation for data communication on the Web, and it involves request/response interactions:



HTTP is an application layer protocol used to deliver resources in distributed hypermedia information system. In a Web application, the request initiates activities that are implemented over the middleware. and the response typically involves returning resources to the browser.

In other words, HTTP (Hypertext Transfer Protocol) is the set of rules for transferring files (text, graphic images, sound, video, and other multimedia files) on the World Wide Web. As soon as a Web user opens their Web browser, the user is indirectly making use of HTTP. HTTP is an application protocol that runs on top of the TCP/IP suite of protocols (the foundation protocols for the Internet).

HTTP concepts include (as the Hypertext part of the name implies) the idea that files can contain references to other files whose selection will elicit additional transfer requests. Any Web server machine contains, in addition to the Web page files it can serve, an HTTP daemon, a program that is designed to wait for HTTP requests and handle them when they arrive. Your Web browser is an HTTP client, sending requests to server machines. When the browser user enters file requests by either "opening" a Web file (typing in a Uniform Resource Locator or URL) or clicking on a hypertext link, the browser builds an HTTP request and sends it to the Internet Protocol address (IP address) indicated by the URL. The HTTP daemon in the destination server machine receives the request and sends back the requested file or files associated with the request. (A Web page often consists of more than one file.)

The current version of HTTP is HTTP 1.1.

In order to build and debug web applications, it is vital to have a good understand of how HTTP works.

HTTP - Resources

The resources delivered as part of this protocol typically include hypertext, marked up using the HyperText Markup Language (HTML), cascading style sheets (CSS), hypermedia and scripts

- Hypertext – text that can be displayed on a computer, or other display device, possibly styled with CSS, and containing references (i.e., hyperlinks) to other hypertext that the reader is able to immediately access, usually via a mouse click.
- Hypermedia – the logical extension of hypertext to graphics, audio and video.
- Hyperlinks – define a structure over the Web. Indeed, this is the structure that Google uses to determine the relevance of hyperlinks that are returned to you by a search.
- Scripts – code that can be executed on the client side.

HTTP - Background

The HTTP protocol is extremely lightweight and simple – indeed, that's one of the main reasons for its success.

Initially, with HTTP/0.9 (the first documented HTTP protocol), a client could only issue GET requests, asking a server for a resource.

Ex.

GET /welcome.html

will cause the server to return the contents of the requested file (the response was required to be HTML).

The HTTP/1.0 protocol, introduced in 1996, extended HTTP/0.9 to include request headers along with additional request methods.

The HTTP/1.1 extension followed soon thereafter, and included the following improvements:

- Faster response, by allowing multiple transactions to take place over a single persistent connection.
- Faster response and bandwidth savings, by adding cache support.
- Faster response for dynamically-generated content, by supporting chunked encoding, which allows a response to be sent before its total length is known.
- Efficient use of IP addresses, multiple domains can be served from a single IP address.
- Support for proxies.
- Support for content negotiations.

HTTP - Basic

- HTTP has always been a stateless protocol.
- This refers to the fact that the protocol does not require the server to retain information related to previous client requests.
- Thus, each client request is executed independently, without any knowledge of the client requests that preceded it.
- This made if very difficult for web applications to respond intelligently to user input, i.e., to create the interactivity that users expect when they use computer applications.
- Cookies, sessions, URL encoded parameters and a few other technologies have been introduced to address this issue, thereby allowing for the emergence of Web 2.0 and 3.0 applications.

HTTP - Sessions

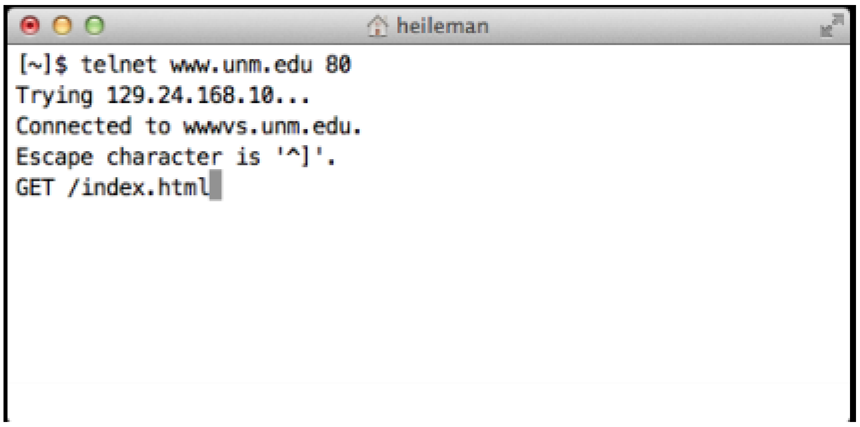
An HTTP session proceeds as follows:

Step 1: An HTTP client (e.g., a browser) establishes a TCP connection to a particular port on a host server (typically this is port 80), and initiates a request. Establishing the TCP connection may first involve using an DNS server in order to obtain an IP Address.

Step 2: An HTTP server listening on that port waits for a client's request message.

Step 3: Upon receiving the request, the server processes it and sends back a status line, such as "HTTP/1.1 200 OK", along with a message of its own (i.e., a response), the body of which might be a requested resource, an error message, or some other information.

In addition to using browser developer tools, you can also directly explore how a web server responds to client requests using telnet:

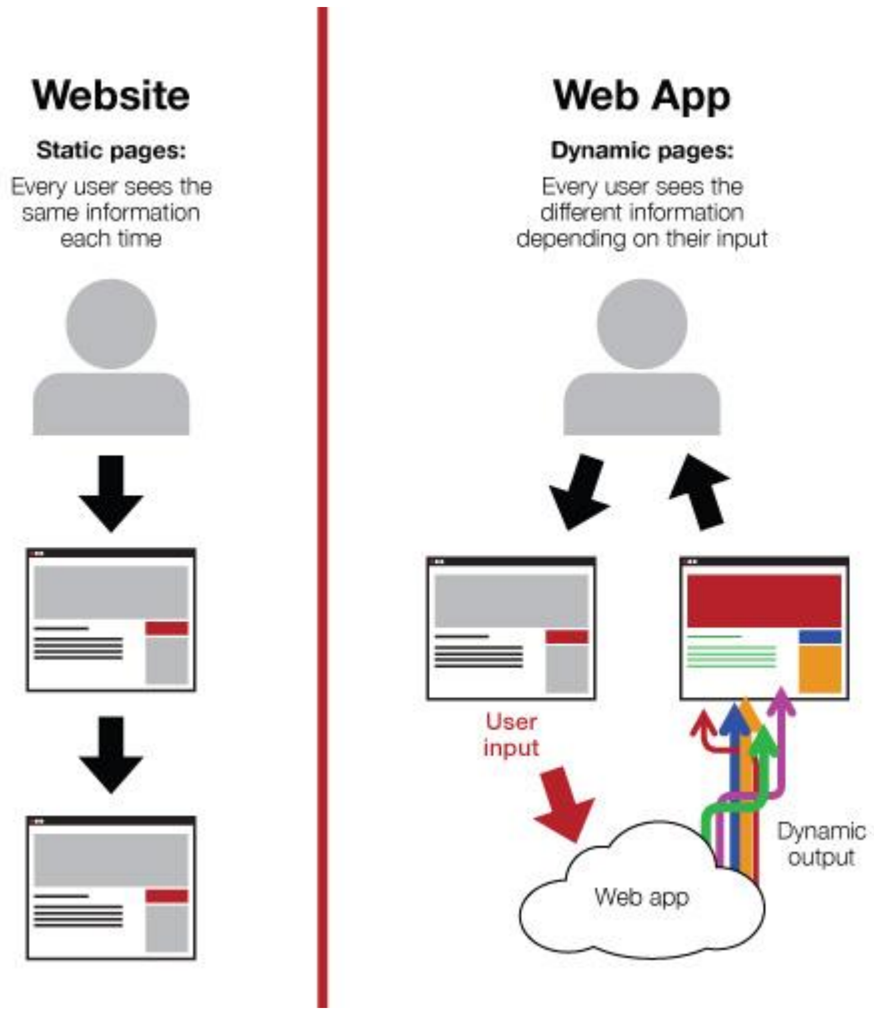


```
[~]$ telnet www.unm.edu 80
Trying 129.24.168.10...
Connected to wwwvs.unm.edu.
Escape character is '^]'.
GET /index.html
```

This tells the server that you (the client) are making an HTTP GET request, asking for the file index.html.

Hit <return>, and the HTML associated with the resource will be provided.

Difference between a website and website app


Website
Static pages:
Every user sees the same information each time
Web App
Dynamic pages:
Every user sees the different information depending on their input
User input
Web app
Dynamic output

A website shows static or dynamic data that is predominantly sent from the server to the user only, whereas a web application serves dynamic data with full two way interaction.

A website shows essentially the same data. Some of it may be dynamic (e.g. CNN or BBC website), but it is generally a one way affair - you are a consumer only.

A web application is two way; you see data that is not only dynamic but often also specific to you. You can work with this data through the web application to publish new content or send meaningful requests back to the server or through the server to third parties (including other users). Examples include;

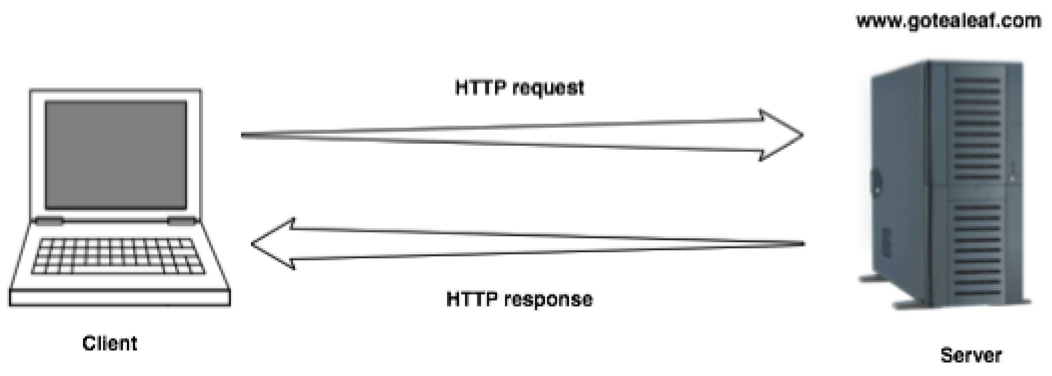
- a stock/share dealing application with real time price data and account, allowing you to deal in real-time.
- a photo editing application.
- a wedding wish list generator, shareable with your guests

- a web game with a persistent world
- A wordpress site's publishing interface (where you write blog posts and generally manage the site) is a web application but the blog itself is not.
- Youtube after you log in (so you can post comments or videos).

What is a website is generally accepted, but what is and is not a web application is a bit of a grey area; I would argue that Google maps is not a web application (its a dynamic web page with search and filtering), whereas Google mail is a true web application and so is much of social media.

1.1 Identify the hypertext transfer protocol (HTTP) required

It is critical to understand that HTTP protocol is stateless. In other words, the server does not hang on to information between each request/response cycle.



Each request made to a resource is treated as a brand-new entity, and different requests are not aware of each other. This *statelessness* is what makes HTTP and the internet so distributed and difficult to control, but it's also the same ephemeral attribute that makes it difficult for web developers to build **stateful** web applications.

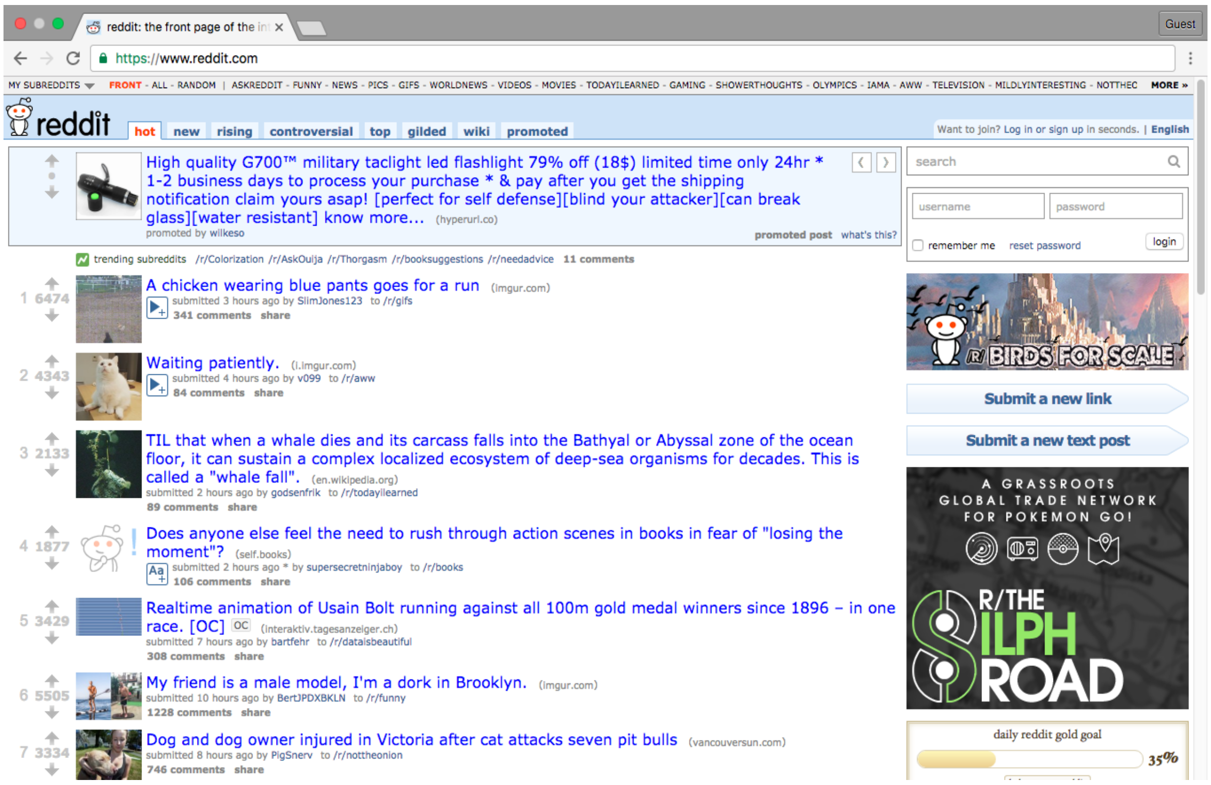
As we look around the internet and use familiar applications, we feel that the application somehow has a certain state. For example, when we log in to Facebook or Twitter, we see our username at the top, signifying our authenticated status. If we click around (which generates new requests to Facebook's servers) we are not suddenly logged out; the server response contains HTML that still shows our username, and the application seems to maintain its state.

In this chapter, we'll focus on how this happens by discussing some of the techniques being employed by web developers to simulate a stateful experience. Along the way, we'll also discuss some techniques used on the client to make displaying dynamic content easy. The approaches we'll discuss are:

- Sessions
- Cookies
- Asynchronous JavaScript calls, or AJAX

A Stateful App

Let's begin by looking at an illustration of a stateful app. When you make a request to https://www.reddit.com, the home page shows up:



reddit: the front page of the in ×

Guest

https://www.reddit.com

MY SUBREDDITS - FRONT - ALL - RANDOM | ASKREDDIT - FUNNY - NEWS - PICS - GIFS - WORLDNEWS - VIDEOS - MOVIES - TODAYILEARNED - GAMING - SHOWERTHOUGHTS - OLYMPICS - IAMA - AWW - TELEVISION - MILDLYINTERESTING - NOTTHEC MORE »

reddit hot new rising controversial top gilded wiki promoted

Want to join? Log in or sign up in seconds. | English

High quality G700™ military taclight led flashlight 79% off (18$) limited time only 24hr * 1-2 business days to process your purchase * & pay after you get the shipping notification claim yours asap! [perfect for self defense][blind your attacker][can break glass][water resistant] know more... (hyperurl.co)
promoted by wilkeso
promoted post what's this?

search

username password
remember me reset password login

trending subreddits /r/Colorization /r/AskOuija /r/Thorgasm /r/booksuggestions /r/needadvice 11 comments

1 6474 A chicken wearing blue pants goes for a run (imgur.com)
submitted 3 hours ago by SlimJones123 to /r/gifs
341 comments share

2 4343 Waiting patiently. (i.imgur.com)
submitted 4 hours ago by v099 to /r/aww
84 comments share

3 2133 TIL that when a whale dies and its carcass falls into the Bathyal or Abyssal zone of the ocean floor, it can sustain a complex localized ecosystem of deep-sea organisms for decades. This is called a "whale fall". (en.wikipedia.org)
submitted 2 hours ago by godsenfrik to /r/todayilearned
89 comments share

4 1877 Does anyone else feel the need to rush through action scenes in books in fear of "losing the moment"? (self.books)
submitted 2 hours ago * by supersecretninjaboy to /r/books
106 comments share

5 3429 Realtime animation of Usain Bolt running against all 100m gold medal winners since 1896 – in one race. [OC] OC (interaktiv.tagesanzeiger.ch)
submitted 7 hours ago by bartfehr to /r/dataisbeautiful
308 comments share

6 5505 My friend is a male model, I'm a dork in Brooklyn. (imgur.com)
submitted 10 hours ago by BertJPDXBKLN to /r/funny
1228 comments share

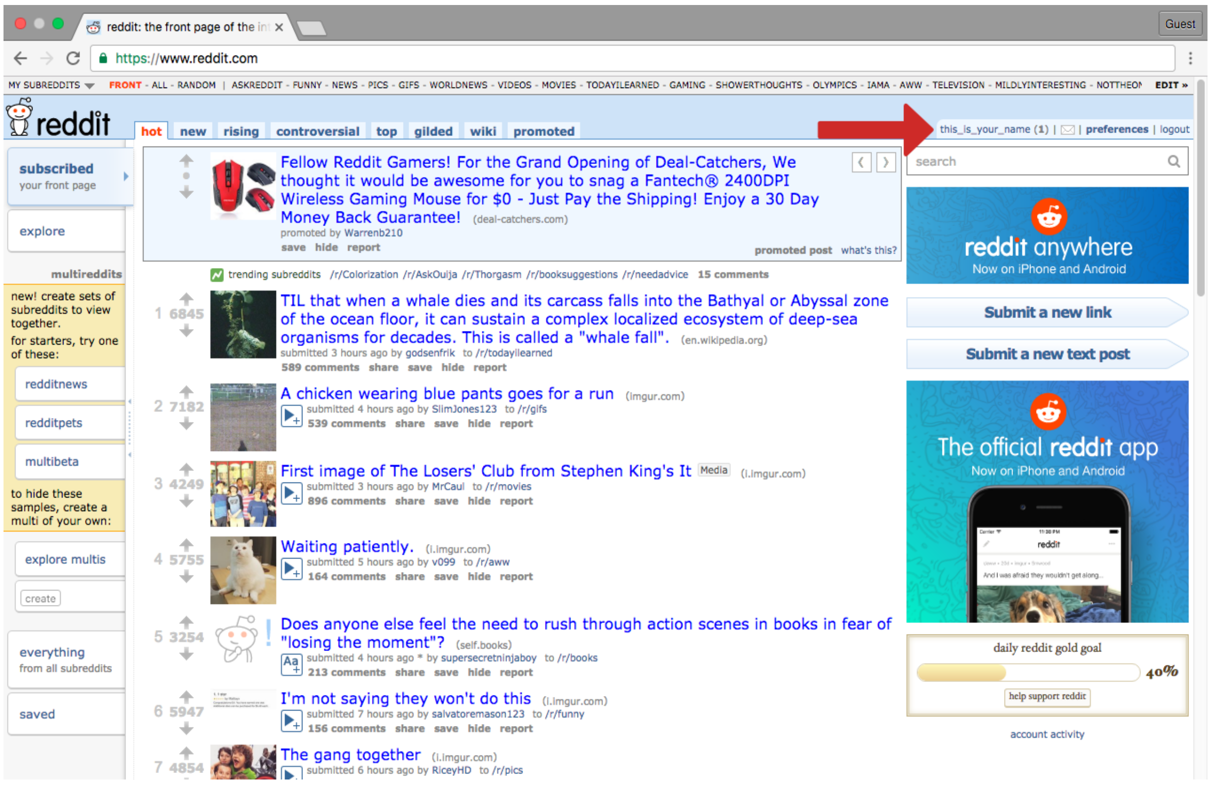
7 3334 Dog and dog owner injured in Victoria after cat attacks seven pit bulls (vancouversun.com)
submitted 8 hours ago by PigSnerv to /r/nottheonion
746 comments share

Submit a new link

Submit a new text post

daily reddit gold goal 35%

Next, login with your username and password:



reddit: the front page of the in ×

Guest

https://www.reddit.com

MY SUBREDDITS - FRONT - ALL - RANDOM | ASKREDDIT - FUNNY - NEWS - PICS - GIFS - WORLDNEWS - VIDEOS - MOVIES - TODAYILEARNED - GAMING - SHOWERTHOUGHTS - OLYMPICS - IAMA - AWW - TELEVISION - MILDLYINTERESTING - NOTTHEON EDIT »

reddit hot new rising controversial top gilded wiki promoted

this_is_your_name (1) | | preferences | logout

subscribed
your front page

explore

multireddits
new! create sets of subreddits to view together.
for starters, try one of these:

redditnews

redditpets

multibeta

to hide these samples, create a multi of your own:

explore multis

create

everything
from all subreddits

saved

Fellow Reddit Gamers! For the Grand Opening of Deal-Catchers, We thought it would be awesome for you to snag a Fantech® 2400DPI Wireless Gaming Mouse for $0 - Just Pay the Shipping! Enjoy a 30 Day Money Back Guarantee! (deal-catchers.com)
promoted by Warrenb210
save hide report
promoted post what's this?

search

trending subreddits /r/Colorization /r/AskOuija /r/Thorgasm /r/booksuggestions /r/needadvice 15 comments

1 6845 TIL that when a whale dies and its carcass falls into the Bathyal or Abyssal zone of the ocean floor, it can sustain a complex localized ecosystem of deep-sea organisms for decades. This is called a "whale fall". (en.wikipedia.org)
submitted 3 hours ago by godsenfrik to /r/todayilearned
589 comments share save hide report

2 7182 A chicken wearing blue pants goes for a run (imgur.com)
submitted 4 hours ago by SlimJones123 to /r/gifs
539 comments share save hide report

3 4249 First image of The Losers' Club from Stephen King's It Media (i.imgur.com)
submitted 3 hours ago by MrCaul to /r/movies
896 comments share save hide report

4 5755 Waiting patiently. (i.imgur.com)
submitted 5 hours ago by v099 to /r/aww
164 comments share save hide report

5 3254 Does anyone else feel the need to rush through action scenes in books in fear of "losing the moment"? (self.books)
submitted 4 hours ago * by supersecretninjaboy to /r/books
213 comments share save hide report

6 5947 I'm not saying they won't do this (i.imgur.com)
submitted 7 hours ago by salvatoremason123 to /r/funny
156 comments share save hide report

7 4854 The gang together (i.imgur.com)
submitted 6 hours ago by RiceyHD to /r/pics

Submit a new link

Submit a new text post

daily reddit gold goal 40%

help support reddit

account activity

At the top, you'll notice your username, signifying that you're now successfully authenticated. If you refresh the page, which issues another request to the reddit server at https://www.reddit.com, you'll see that the page stays the same; you seem to be still logged in. What's happening here? Isn't HTTP supposed to be a stateless protocol? How does the server know to remember your username and dynamically display it on the page even after sending a new request? This behavior is so common now that we don't think twice about it. It's how the items in your shopping cart can stay as you keep adding items to it, sometimes even across multiple days. It's how Gmail identifies you and displays your name and some customized greeting. It's how all modern web applications work.

Sessions

It's obvious the stateless HTTP protocol is somehow being augmented to maintain a sense of statefulness. With some help from the client (i.e., the browser), HTTP can be made to act as if it were maintaining a stateful connection with the server, even though it's not. One way to accomplish this is by having the server send some form of a unique token to the client. Whenever a client makes a request to that server, the client appends this token as part of the request, allowing the server to identify clients. In web development, we call this unique token that gets passed back and forth the **session identifier**.

This mechanism of passing a session id back and forth between the client and server creates a sense of persistent connection between requests. Web developers leverage this faux statefulness to build sophisticated applications. Each request, however, is technically stateless and unaware of the previous or the next one.

This sort of faux statefulness has several consequences. First, every request must be inspected to see if it contains a session identifier. Second, if this request does, in fact, contain a session id, the server must check to ensure that this session id is still valid. The server needs to maintain some rules with regards to how to handle session expiration and also decide how to store its session data. Third, the server needs to retrieve the session data based on the session id. And finally, the server needs to recreate the application state (e.g., the HTML for a web request) from the session data and send it back to the client as the response.

This means that the server has to work very hard to simulate a stateful experience, and every request still gets its own response, even if most of that response is identical to the previous response. For example, if you're logged into Facebook, the server has to generate the initial page you see, and the response is a pretty complex and expensive HTML that your browser displays. The Facebook server has to add up all the likes and comments for every photo and status, and present it in a timeline for you. It's a very expensive page to generate. Now if you click on the "like" link for a photo, Facebook has to regenerate that entire page. It has to increment the like count for that photo you liked, and then send that HTML back as a response, even though the vast majority of the page stayed the same.

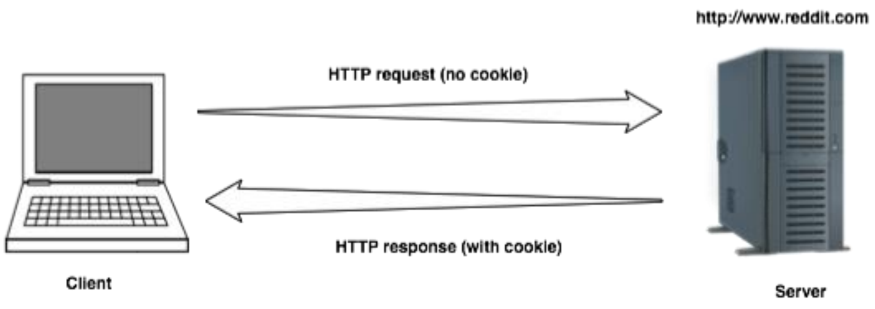
nately, Facebook uses Ajax instead of refreshing your browser. But if Facebook didn't use Ajax each
h would take a really long time.

There are many advanced techniques that servers employ to optimize sessions and, as you can imagine, there are also many security concerns. Most of the advanced session optimization and security concerns are out of scope of this book, but we'll talk about one common way to store session information: in a browser cookie

Cookies

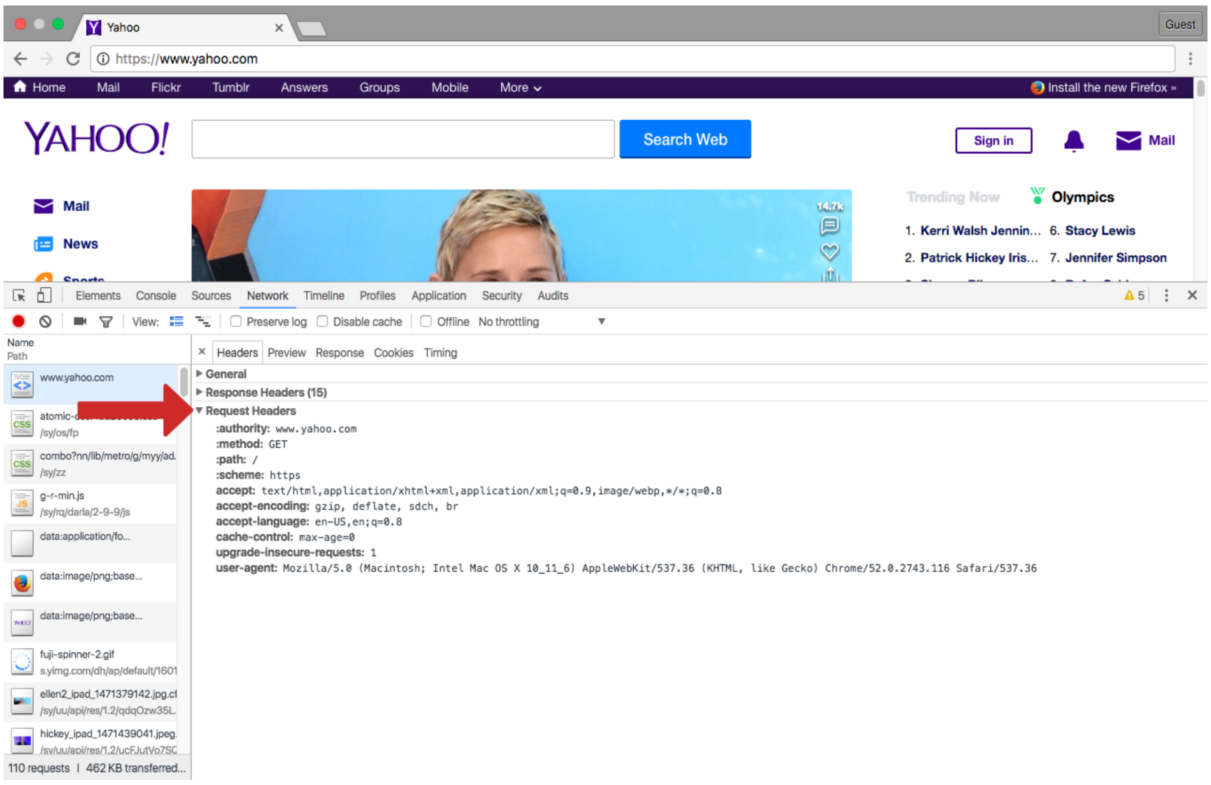
A cookie is a piece of data that's sent from the server and stored in the client during a request/response cycle. **Cookies**or **HTTP cookies**, are small files stored in the browser and contain the session information.

By default, most browsers have cookies enabled. When you access any website for the first time, the server sends session information and sets it in your browser cookie on your local computer. Note that the actual session data is stored on the server. The client side cookie is compared with the server-side session data on each request to identify the current session. This way, when you visit the same website again, your session will be recognized because of the stored cookie with its associated information.



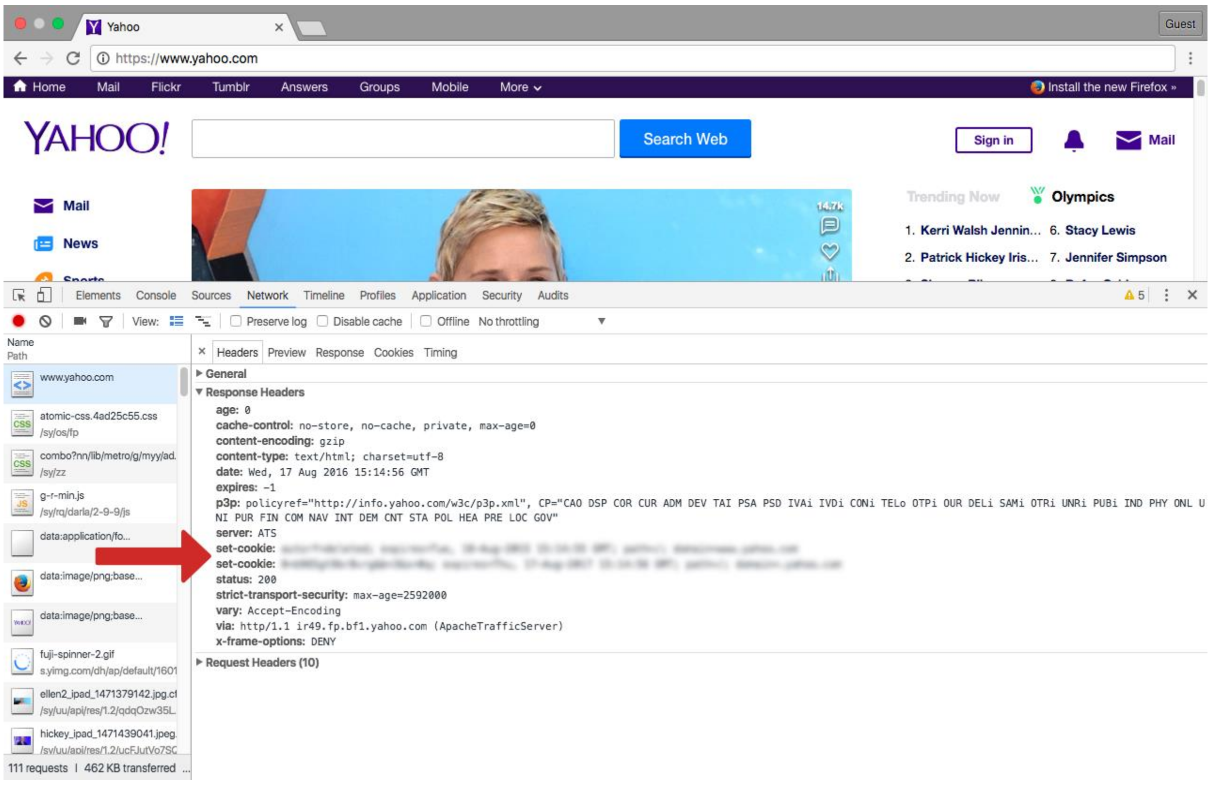
Let's see a real example of how cookies are initiated with the help of the browser inspector. We'll make a request to the address to https://www.yahoo.com. Note that if you're following along, you may have to use a different website if you already have an existing cookie from Yahoo.

With the inspector's Network tab open, navigate to that address and inspect the request headers:


Yahoo
https://www.yahoo.com
Home Mail Flickr Tumblr Answers Groups Mobile More
Search Web
Sign in
Mail
News
Trending Now
Olympics
1. Kerri Walsh Jennin...
2. Patrick Hickey Iris...
6. Stacy Lewis
7. Jennifer Simpson
Elements Console Sources Network Timeline Profiles Application Security Audits
Preserve log
Disable cache
Offline
No throttling
Headers Preview Response Cookies Timing
www.yahoo.com
110 requests | 462 KB transferred...

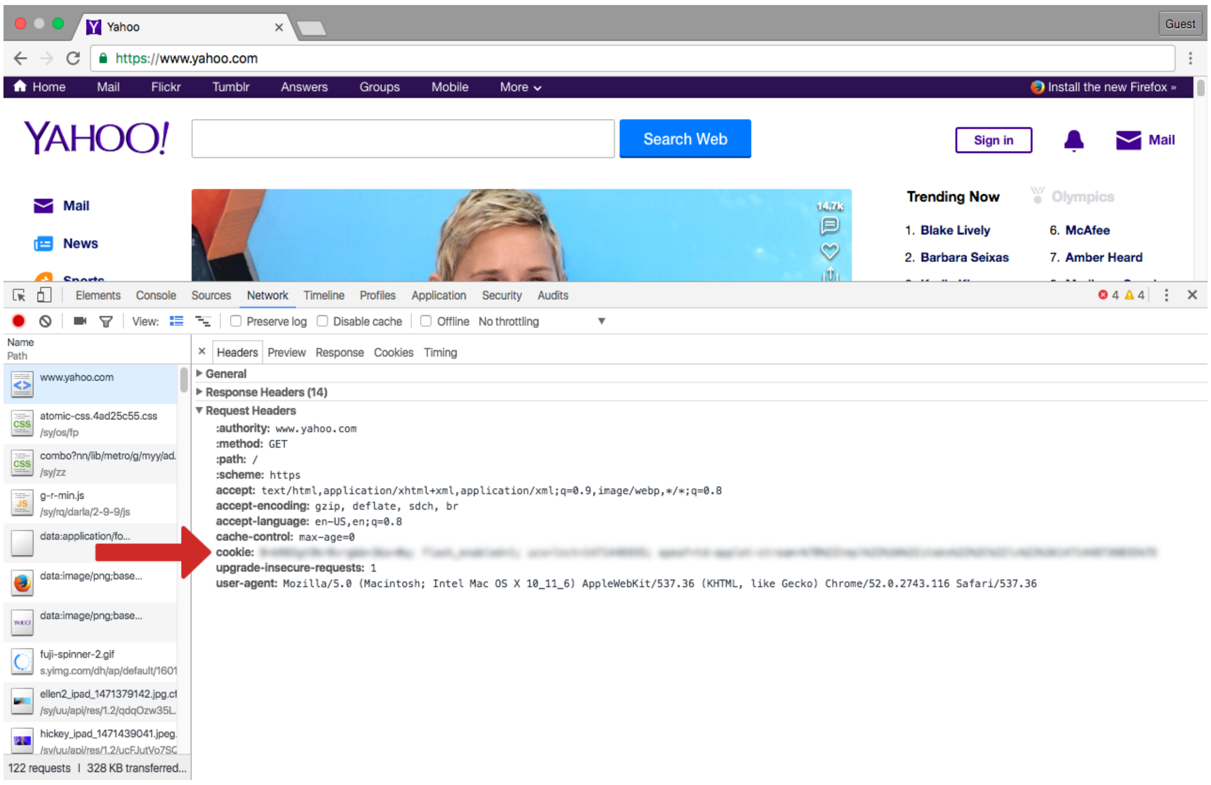
```
▶ General
▶ Response Headers (15)
▼ Request Headers
   :authority: www.yahoo.com
   :method: GET
   :path: /
   :scheme: https
   accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
   accept-encoding: gzip, deflate, sdch, br
   accept-language: en-US,en;q=0.8
   cache-control: max-age=0
   upgrade-insecure-requests: 1
   user-agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/52.0.2743.116 Safari/537.36
```

Note it has no reference to cookies. Next, look at the response headers:



```
General
Response Headers
  age: 0
  cache-control: no-store, no-cache, private, max-age=0
  content-encoding: gzip
  content-type: text/html; charset=utf-8
  date: Wed, 17 Aug 2016 15:14:56 GMT
  expires: -1
  p3p: policyref="http://info.yahoo.com/w3c/p3p.xml", CP="CAO DSP COR CUR ADM DEV TAI PSA PSD IVAi IVDi CONi TELo OTPi OUR DELi SAMi OTRi UNRi PUBi IND PHY ONL U
  NI PUR FIN COM NAV INT DEM CNT STA POL HEA PRE LOC GOV"
  server: ATS
  set-cookie: [redacted]
  set-cookie: [redacted]
  status: 200
  strict-transport-security: max-age=2592000
  vary: Accept-Encoding
  via: http/1.1 ir49.fp.bf1.yahoo.com (ApacheTrafficServer)
  x-frame-options: DENY
Request Headers (10)
```

You'll notice it has **set-cookie** headers that add cookie data to the response. This cookie data got set on the first visit to the website. Finally, make a request to the same address and then look at the request headers:

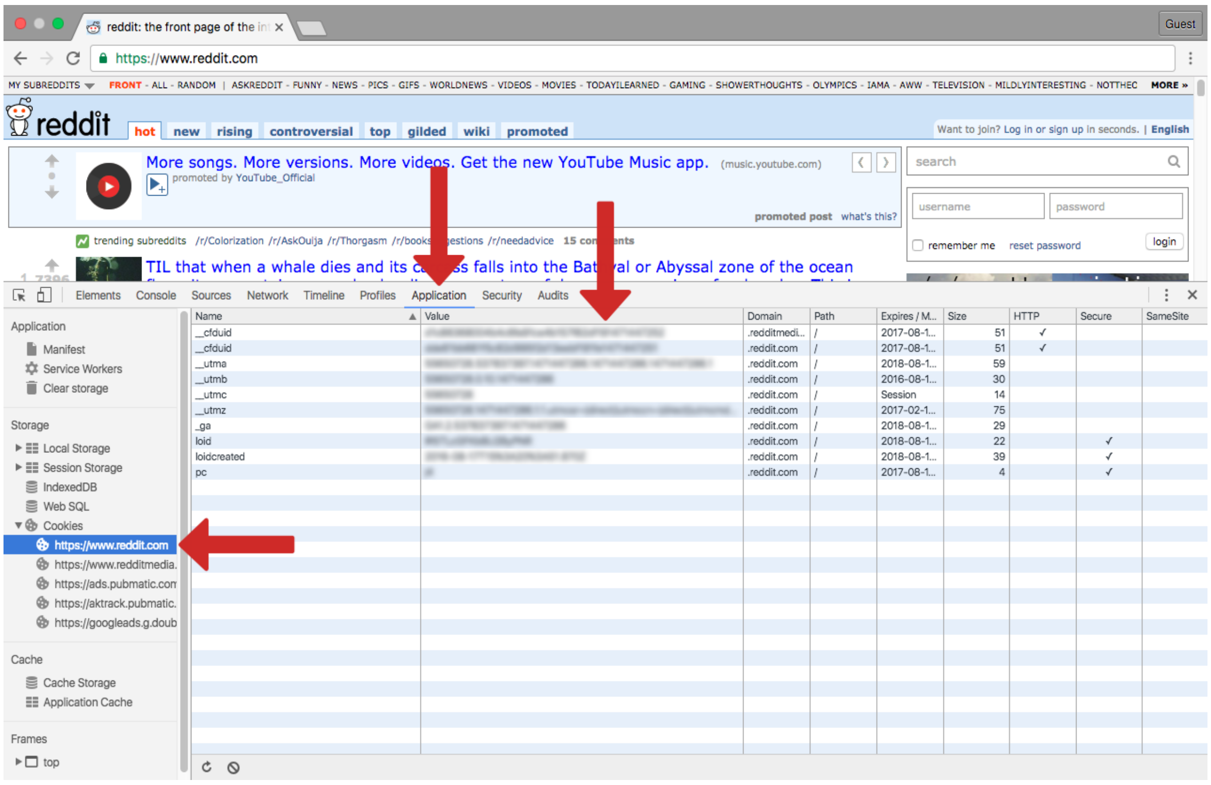


```
General
Response Headers (14)
Request Headers
  :authority: www.yahoo.com
  :method: GET
  :path: /
  :scheme: https
  accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
  accept-encoding: gzip, deflate, sdch, br
  accept-language: en-US,en;q=0.8
  cache-control: max-age=0
  cookie: [redacted]
  upgrade-insecure-requests: 1
  user-agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/52.0.2743.116 Safari/537.36
```

You'll see the **cookie** header set (note that this is the *request* header, which implies it's being sent by your client to the server). It contains the cookie data sent previously by the **set-cookie** response header. This piece of data will be sent to the server each time you make a request and uniquely identifies you -- or more precisely, it identifies your client, which is your browser. The browser on your computer stores these cookies. Now, if you were to close your browser and shut down your computer, the cookie information would still be persisted.

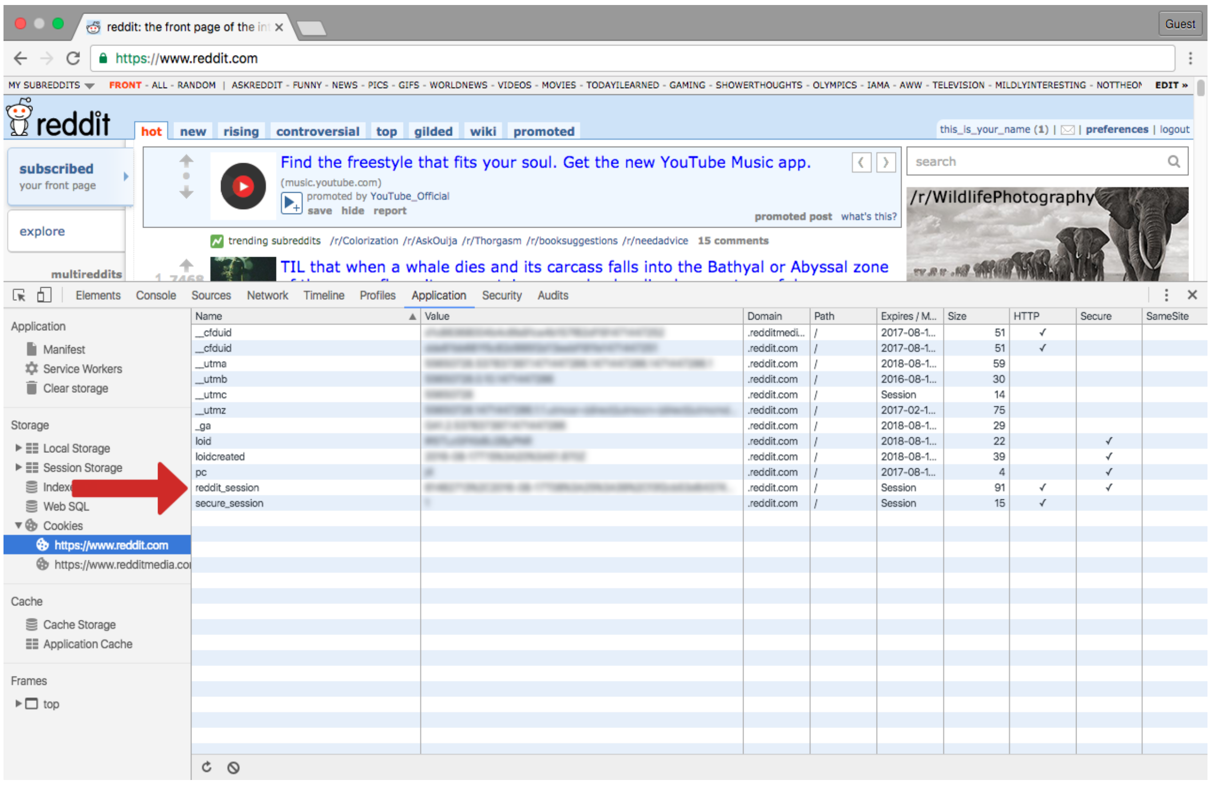
Let's now revisit the original example of how Reddit, or any web application, keeps track of the fact that we maintain our authenticated status even though we issue request after request. Remember, each request is unrelated to each other and are not aware of each other - how does the app "remember" we're authenticated? If you're following along, perform the following steps with the inspector open:

1. Click the Application tab and navigate to https://www.reddit.com
2. Expand the cookies section and click on www.reddit.com where you'll see the cookies that came with our initial request under the value column:



Name	Value	Domain	Path	Expires / M...	Size	HTTP	Secure	SameSite
__cfduid	[redacted]	.redditmedi...	/	2017-08-1...	51	✓		
__cfduid	[redacted]	.reddit.com	/	2017-08-1...	51	✓		
__utma	[redacted]	.reddit.com	/	2018-08-1...	59			
__utmb	[redacted]	.reddit.com	/	2016-08-1...	30			
__utmc	[redacted]	.reddit.com	/	Session	14			
__utmz	[redacted]	.reddit.com	/	2017-02-1...	75			
_ga	[redacted]	.reddit.com	/	2018-08-1...	29			
loid	[redacted]	.reddit.com	/	2018-08-1...	22		✓	
loidcreated	[redacted]	.reddit.com	/	2018-08-1...	39		✓	
pc	[redacted]	.reddit.com	/	2017-08-1...	4		✓	

3. After logging in, you should notice a unique session in the second to last row. That session id is saved into a cookie in your browser, and is attached along with every future request that you make to reddit.com:

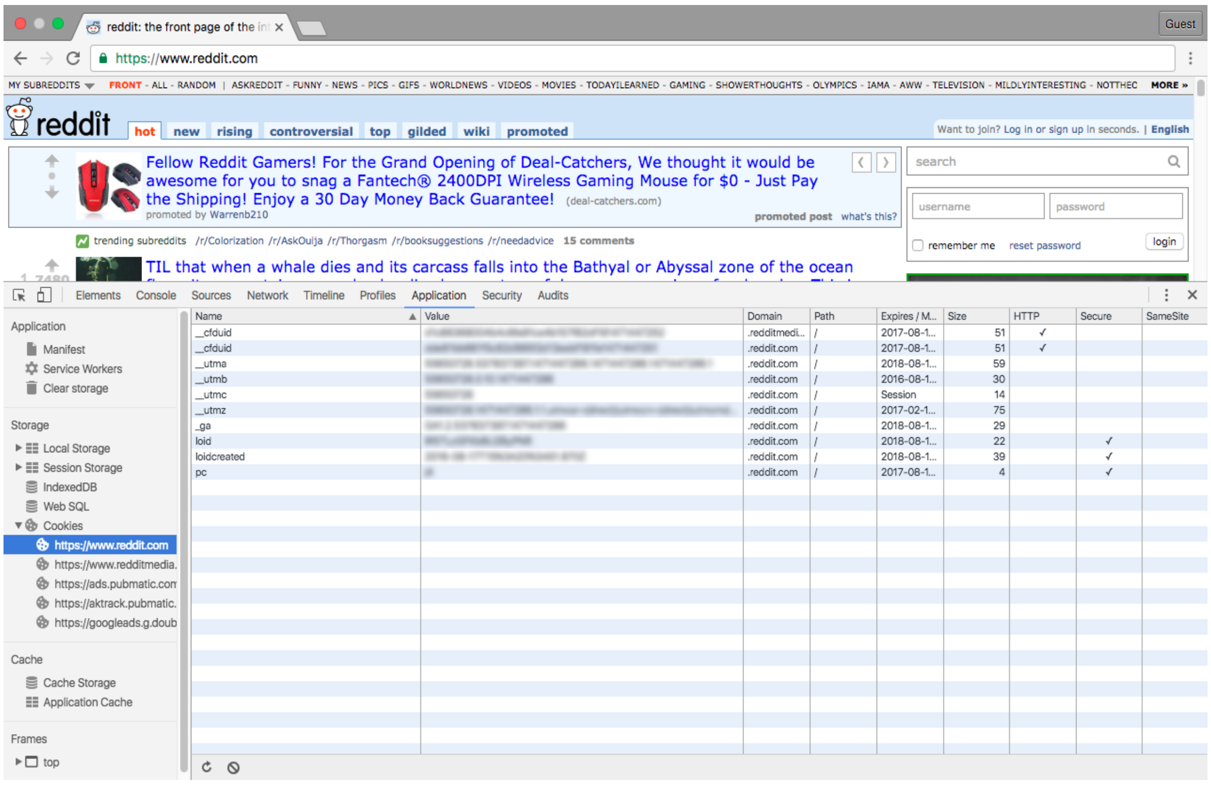


Name	Value	Domain	Path	Expires / M...	Size	HTTP	Secure	SameSite
__cfduid	[redacted]	.redditmedi...	/	2017-08-1...	51	✓		
__cfduid	[redacted]	.reddit.com	/	2017-08-1...	51	✓		
__utma	[redacted]	.reddit.com	/	2018-08-1...	59			
__utmb	[redacted]	.reddit.com	/	2016-08-1...	30			
__utmc	[redacted]	.reddit.com	/	Session	14			
__utmz	[redacted]	.reddit.com	/	2017-02-1...	75			
_ga	[redacted]	.reddit.com	/	2018-08-1...	29			
loid	[redacted]	.reddit.com	/	2018-08-1...	22		✓	
loidcreated	[redacted]	.reddit.com	/	2018-08-1...	39		✓	
pc	[redacted]	.reddit.com	/	2017-08-1...	4		✓	
reddit_session	[redacted]	.reddit.com	/	Session	91	✓	✓	
secure_session	[redacted]	.reddit.com	/	Session	15	✓		

With the session id now being sent with every request, the server can now uniquely identify this client. When the server receives a request with a session id, the server will look for the associated data based on that id, and in that associated session data is where the server "remembers" the state for that client, or more precisely, for that session id.

> Where is the session data stored?
>
> The simple answer is: on the server somewhere. Sometimes, it's just stored in memory, while other times, it could be stored in some persistent storage, like a database or key/value store. Where the session data is actually is not too important right now. The most important thing is to understand that the session id is stored on the client, and it is used as a "key" to the session data stored server side. That's how web applications work around the statelessness of HTTP.

It is important to be aware of the fact that the id sent with a session is unique and expires in a relatively short time. In this context, it means you'll be required to login again after the session expires. If we log out, the session id information is gone:



Name	Value	Domain	Path	Expires / M...	Size	HTTP	Secure	SameSite
__cfduid	[illegible]	.redditmedi...	/	2017-08-1...	51	✓		
__cfduid	[illegible]	.reddit.com	/	2017-08-1...	51	✓		
__utma	[illegible]	.reddit.com	/	2018-08-1...	59			
__utmb	[illegible]	.reddit.com	/	2016-08-1...	30			
__utmc	[illegible]	.reddit.com	/	Session	14			
__utmz	[illegible]	.reddit.com	/	2017-02-1...	75			
_ga	[illegible]	.reddit.com	/	2018-08-1...	29			
loid	[illegible]	.reddit.com	/	2018-08-1...	22		✓	
loidcreated	[illegible]	.reddit.com	/	2018-08-1...	39		✓	
pc	[illegible]	.reddit.com	/	2017-08-1...	4		✓	

This also implies that if we manually remove the session id (in the inspector, you can right click on cookies and delete them), then we have essentially logged out.

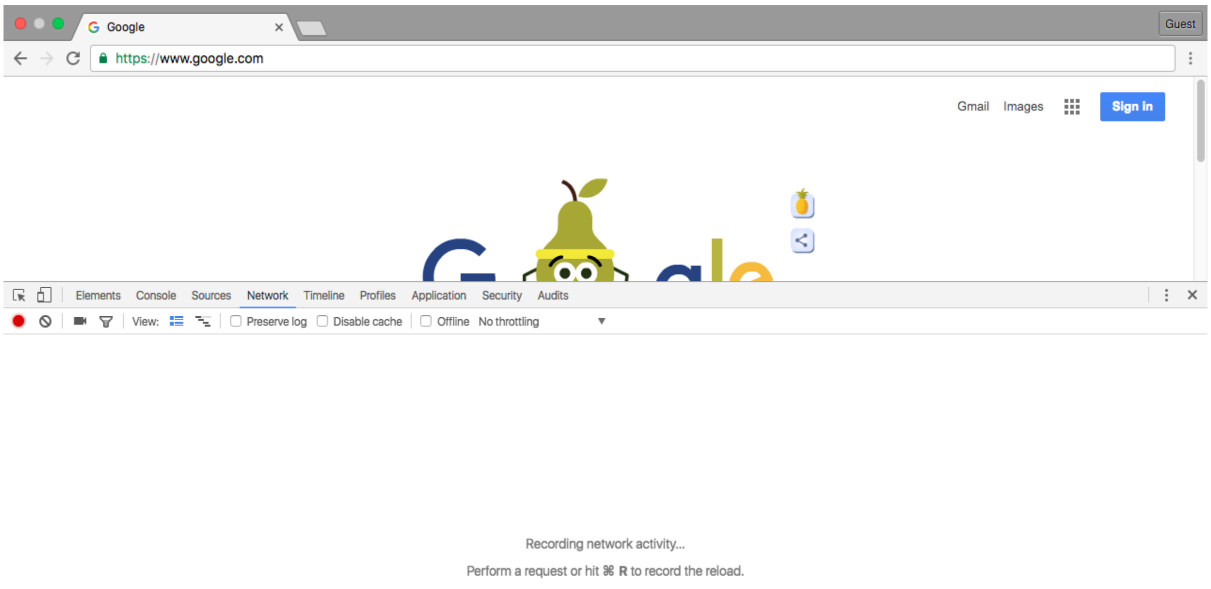
To recap, we've seen that the session data is generated and stored on the server-side and the session id is sent to the client in the form of a cookie. We've also looked at how web applications take advantage of this to mimic a stateful experience on the web.

AJAX

Last, we'll briefly mention AJAX and what it means in the HTTP request/response cycle. AJAX is short for Asynchronous JavaScript and XML. Its main feature is that it allows browsers to issue requests and process responses *without a full page refresh*. For example, if you're logged into Facebook, the server has to generate the initial page you see, and the response is a pretty complex and expensive HTML page that your browser displays. The Facebook server has to add up all the likes and comments for every photo and status, and present it in a timeline for you. As we described earlier, it's a very expensive page to re-generate for every request (remember, every action you take -- clicking a link, submitting a form -- issues a new request).

When AJAX is used, all requests sent from the client are performed *asynchronously*, which just means that the page doesn't refresh. Let's see an example by performing some search on google:

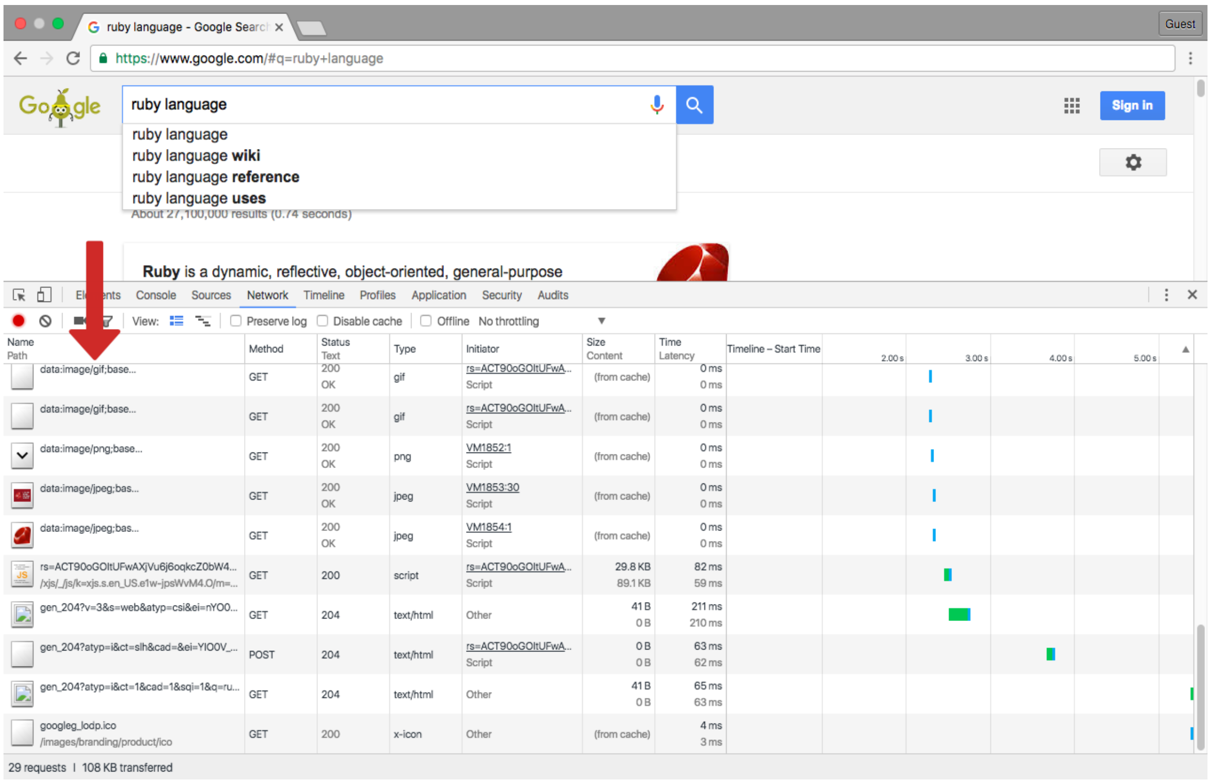
- Make a request to https://www.google.com and open the browser's inspector network tab. You'll notice the network tab is empty:



Recording network activity...

Perform a request or hit ⌘ R to record the reload.

- As soon as you start your search, you'll see the network tab gets flooded with requests.



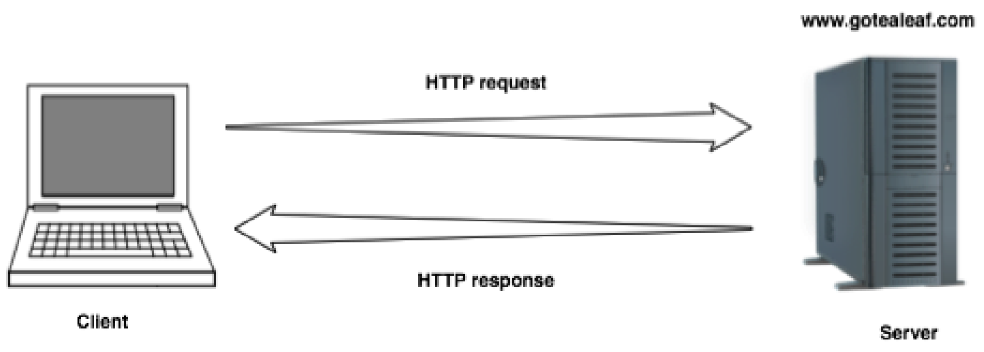
Name Path	Method	Status Text	Type	Initiator	Size Content	Time Latency
data:image/gif;base...	GET	200 OK	gif	rs=ACT90oGOltUFwA... Script	(from cache)	0 ms 0 ms
data:image/gif;base...	GET	200 OK	gif	rs=ACT90oGOltUFwA... Script	(from cache)	0 ms 0 ms
data:image/png;base...	GET	200 OK	png	VM1852:1 Script	(from cache)	0 ms 0 ms
data:image/jpeg;bas...	GET	200 OK	jpeg	VM1853:30 Script	(from cache)	0 ms 0 ms
data:image/jpeg;bas...	GET	200 OK	jpeg	VM1854:1 Script	(from cache)	0 ms 0 ms
rs=ACT90oGOltUFwAXjVu6j6oqkcZ0bW4... /xjs/_/js/k=xjs.s.en_US.e1w-jpsWvM4.O/m=...	GET	200	script	rs=ACT90oGOltUFwA... Script	29.8 KB 89.1 KB	82 ms 59 ms
gen_204?v=3&s=web&atyp=csi&ei=nYO0...	GET	204	text/html	Other	41 B 0 B	211 ms 210 ms
gen_204?atyp=i&ct=slh&cad=&ei=YIO0V_...	POST	204	text/html	rs=ACT90oGOltUFwA... Script	0 B 0 B	63 ms 62 ms
gen_204?atyp=i&ct=1&cad=1&sqi=1&q=ru...	GET	204	text/html	Other	41 B 0 B	65 ms 63 ms
googleg_lodp.ico /images/branding/product/ico	GET	200	x-icon	Other	(from cache)	4 ms 3 ms

29 requests | 108 KB transferred

No doubt you've performed searches many times and notice the page doesn't refresh. The Network tab however gives us some new insight into what's happening: every letter you type is issuing a new request, which means that an AJAX request is triggered with every key-press. The responses from these requests are being processed by some callback. You can think of a callback as a piece of logic you pass on to some function to be executed after a certain event has happened. In this case, the callback is triggered when the response is returned. You can probably guess that the callback that's processing these asynchronous requests and responses is updating the HTML with new search results.

We won't get into what the callback looks like or how to issue an AJAX request, but the main thing to remember is that AJAX requests are just like normal requests: they are sent to the server with all the normal components of an HTTP request, and the server handles them like any other request. The only difference is that instead of the browser refreshing and processing the response, the response is processed by a callback function, which is usually some client-side JavaScript code.

1.2 Recognise the limitations HTTP has when developing web applications



There are a number of limitations using HTTP when developing web applications, such as:

Every asset requires its own request, in HTTP, so if you have 5 pictures, 2 CSS files, and 1 JavaScript file included in your webpage then 10 HTTP requests must be made:

- 1 for the webpage itself
- 1 to check for a favicon
- 5 for the pictures/images
- 2 for the CSS files
- 1 for the JavaScript file

How does HTTP work? How is HTTPS different from HTTP?

There are other limitations between using HTTP and HTTPS:

You click to check out at an online merchant. Suddenly your browser address bar says HTTPS instead of HTTP. What's going on? Is your credit card information safe?

Good news. Your information is safe. The website you are working with has made sure that no one can steal your information.

Instead of HyperText Transfer Protocol (HTTP), this website uses HyperText Transfer Protocol Secure (HTTPS). Using HTTPS, the computers agree on a "code" between them, and then they scramble the messages using that "code" so that no one in between can read them. This keeps your information safe from hackers. They use the "code" on a Secure Sockets Layer (SSL), sometimes called Transport Layer Security (TLS) to send the information back and forth.

Other limitations of HTTP/1.0 when developing web applications

The other limitation is that HTTP/1.0 only defined the GET, HEAD and POST methods and for ordinary browsing this is enough but creates substantial issues when developing web applications. However, one may want to use HTTP to edit and maintain files directly on the server, instead of having to go through an FTP server as is common today. HTTP/1.1 adds a number of new methods for this.

These are:

PUT

PUT uploads a new resource (file) to the server under the given URL. Exactly what happens on the server is not defined by the HTTP/1.1 spec, but authoring programs like Netscape Composer use PUT to upload a file to the web server and store it there. PUT requests should not be cached.

DELETE

Well, it's obvious, isn't it? The DELETE method requests the server to delete the resource identified by the URL. On UNIX systems this can fail if the server is not allowed to delete the file by the file system.

META HTTP-EQUIV

Not all web servers are made in such a way that it is easy to set, say the Expires header field or some other header field you might want to set for one particular file. There is a way around that problem. In HTML there is an element called META that can be used to set HTTP header fields. It was really supposed to be parsed by the web server and inserted into the response headers, but very few servers actually do this, so browsers have started implementing it instead. It's not supported in all browsers, but it can still be of some help.

The usage is basically this: put it in the HEAD-element of the HTML file and make it look like this:

<META HTTP-EQUIV="header field name" CONTENT="field value">

The Host header field

Many web hotels let a single physical machine serve what to the user looks like several different servers. For instance, http://www.foo.com/, http://www.bar.com/ and http://www.baz.com/ could all very well be served from the same web server. Still, the user would see different pages by going to each of the different

URLs. To enable this, an extension to the HTTP protocol was needed to let the web server know which of these different web servers the user wanted to access.

The solution is the Host header field. When the user requests http://www.bar.com/ the web server will receive a header set to "Host: www.bar.com" and thus know which top page to deliver. It will also usually create separate logs for the different virtual servers.

HTTP/1.1 and HTTP/2

HTTP/1.1 ,update to HTTP 1, was standardized back in 1999. HTTP/1.1 was helping people for running their business online but there were some limitations in HTTP/1.1 which were then updated in v2 .

- Latency in loading a web page
- Head of line blocking problem
- Multiplexing multiple requests over a single TCP connection
- Different resource request for each resource file
- HTTP headers are long which causes delay in web page loading

The web back then was very different, with slower web pages and slower internet connections. But over the years, the way that we use the web has changed significantly and websites are now much more complex than they used to be.

Major update to HTTP launched in 2015 as HTTP/2 which resolved issues faced in HTTP/1.1. HTTP/2 was derived from the earlier experimental SPDY protocol, developed by Google. It made web applications faster, simpler, and more robust.

How to switch to HTTP/2

HTTP/2 is backwards-compatible with HTTP/1.1. However you need to do following things to switch to HTTP/2 :

- **Adopt HTTPS and get a TLS certificate**
- **Using image Sprites instead of multiple files**
- **Adding Images Inline**
- **Concatenating JavaScript and CSS files**
- **Domain Sharding**

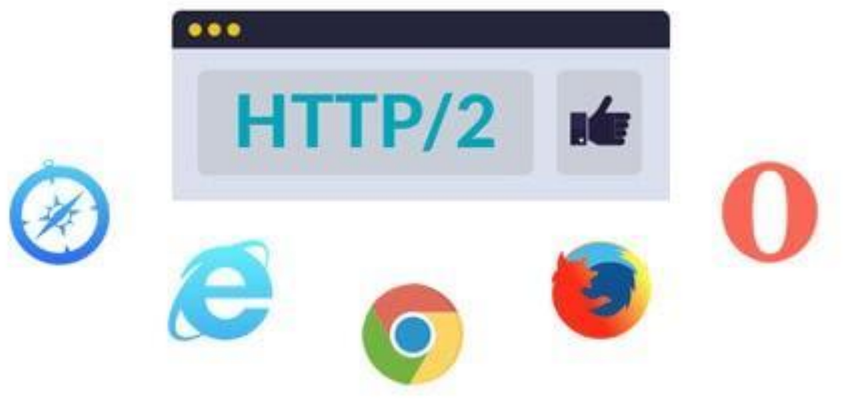
How HTTP/2 resolved HTTP/1.1's limitations

HTTP/2 focuses on specific goals and issues of concern in HTTP/1.1 :

- To create a algorithm which allows client and server to elect to use HTTP 1.1 or 2.0
- To maintain high compatibility for methods,status code,URI's and header fields with HTTP/1.1
- To decrease latency to improve web page load speed
- Support common existing use cases of HTTP, such as desktop web browsers, mobile web browsers, web APIs,web servers at various scales, proxy servers, reverse proxy servers, firewalls, and content delivery networks.

Which Browsers Support HTTP/2?

Most of the browsers now support HTTP/2. The list of browsers includes latest versions of Chrome, Firefox, Safari, Internet Explorer, Opera and others. You can use Akamai's tool to check if your browser supports HTTP/2.



How HTTP/2 help improve the web page load speed

Header Compression

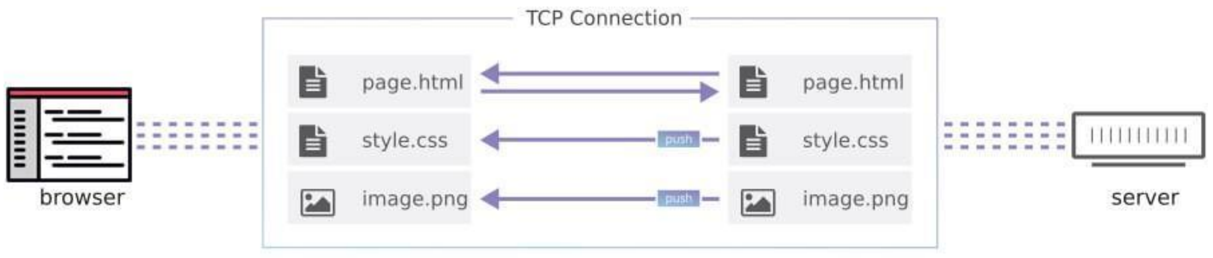
The header fields are transmitted after the request or response line. HTTP/1.1 requests and responses are not compressed and transmitted in plain text which causes a lot of unnecessary bytes being downloaded.

But in HTTP/2, the header fields are compressed and transmitted in binary codes making it more compact and efficient for the user computer to load.

Header compression decreases the overhead of client request which lowers bandwidth and decreases page load time.

Server Push

In HTTP/1.1, when your browser connects to a web server requesting a web page, the HTML file is sent first and then your browser request for other resource files like CSS, javascript etc. But in HTTP2 Server push, In addition to the response to the original request, a server can send multiple responses to a single client request without the client having to request each one explicitly.

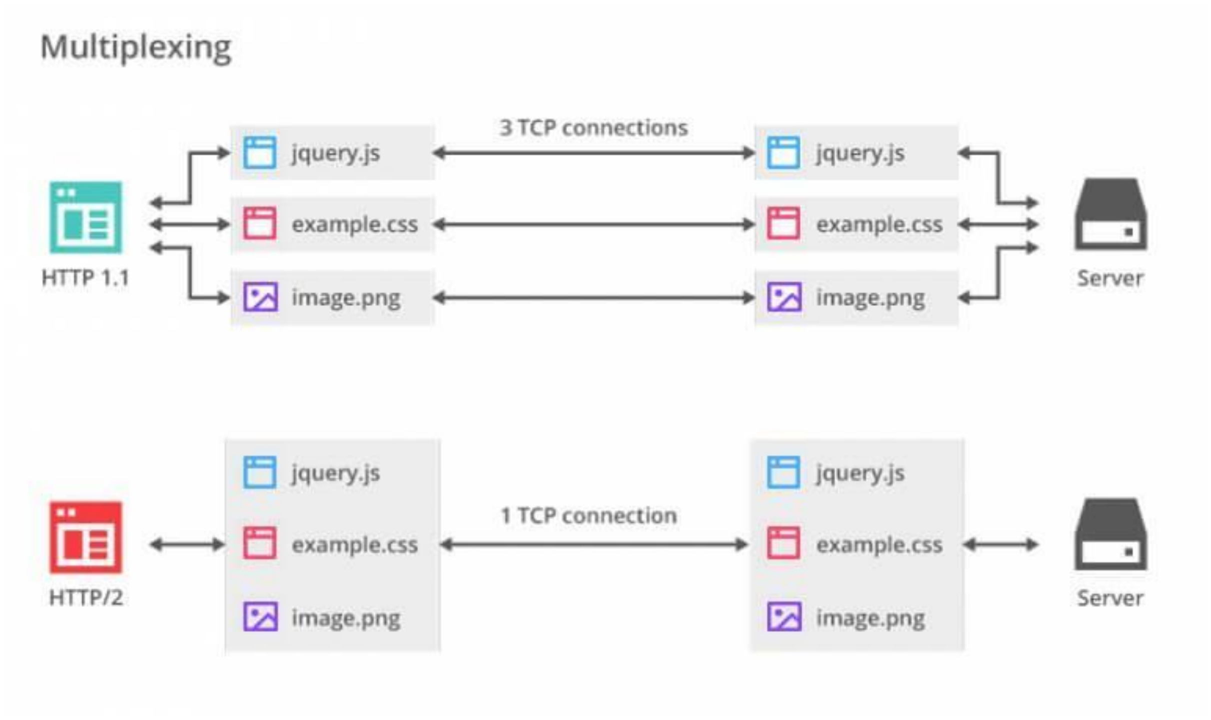


Single TCP Connection, Single HTTP Request

If you have ever added an inline CSS, javascript or any other source file through data URI then you already experienced server push. By manually adding the source file into the URI, we are pushing that source file to the client, without waiting for the client to request it. We can achieve the same results with HTTP/2 with additional performance benefits.

Multiplexing multiple requests over a single TCP connection

One of the limitations of HTTP/1.1 is that it is only able to request files one by one on a single connection — this means requesting the file, waiting for a response, downloading the file and then requesting the next one. This creates the problem of head-of-line blocking and inefficient use of the underlying TCP connection.



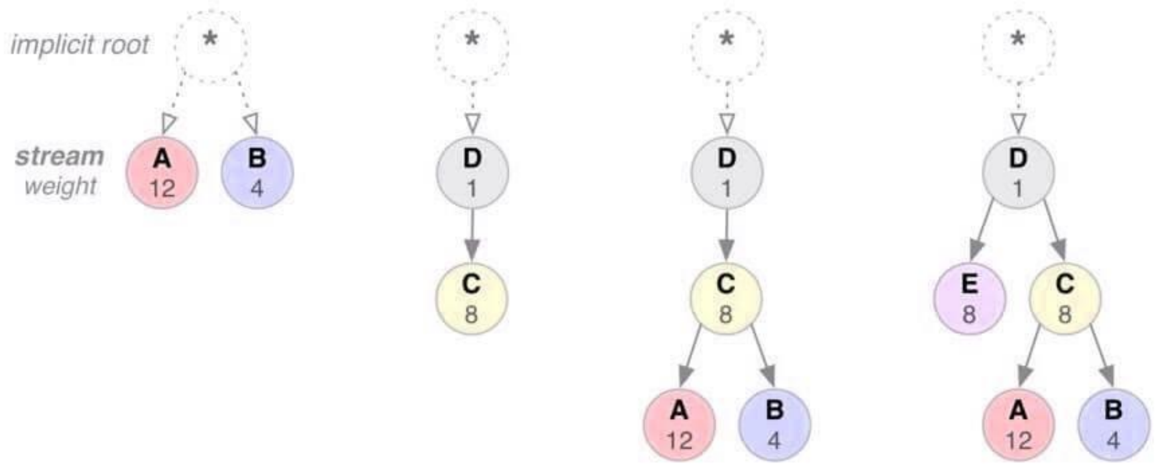
The new binary framing layer in HTTP/2 removes these limitations, and enables full request and response multiplexing, by allowing the client and server to break down an HTTP message into independent frames, interleave them, and then reassemble them on the other end.

Stream Prioritization

Once an HTTP message can be split into many individual frames, HTTP/2 allows for multiplexing of frames from multiple streams. The way in which frames are interleaved and delivered by client and server becomes a critical performance concern. To execute this, HTTP/2 standard associate weight and dependency to each stream.

Each stream may be assigned an integer weight between 1 and 256.

Each stream may be given an explicit dependency on another stream.



HTTP/2 stream dependencies and weights

The combination of stream dependencies and weights allows the client to construct and communicate a "prioritization tree" that expresses how it would prefer to receive responses. In turn, the server can use this information to prioritize stream processing by controlling the allocation of CPU, memory, and other resources, and once the response data is available, allocation of bandwidth to ensure optimal delivery of high-priority responses to the client.

Advantages of HTTP/2

The advantages of switching to HTTP/2 are immediate. Most users are using the browser that can take advantage of the protocol, every day figure is rising up.

More Shielded: HTTP/2 always have encryption on by default, which results in high security across sites.

Mobile friendly: All the mobile sites are allowed with the high amount of requests to prevent data bytes from the headers from being downloaded is the result of header compression feature.

Faster Page Load Times: After removing many impediments of the protocol,HTTP/2 promises to be faster than the current standard.

Compatibility with HTTP/1.1: HTTP/2 is compatible with HTTP/1.1 . To ensure that the switch to the new protocol is as smooth as possible, methods, status codes, header fields, and URIs have remained same as in HTTP/1.1.

Less dependency on alternatives: Multiplexing feature solves the problem of high time-consuming methods to reducing the number of requests from the server — like domain sharding, image sprites or in-lining Javascript and CSS — won't be as indispensable.

Disadvantages of HTTP/2

There are also some limitations businesses should take into account before moving to the new protocol.

Take care of your audience: Although most of the users are using browsers that can take advantage of the protocol there are also a number of people continue to use older browsers which don't. In addition to this, not all web servers support HTTP/2. So current page load speed optimization process will remain necessary.

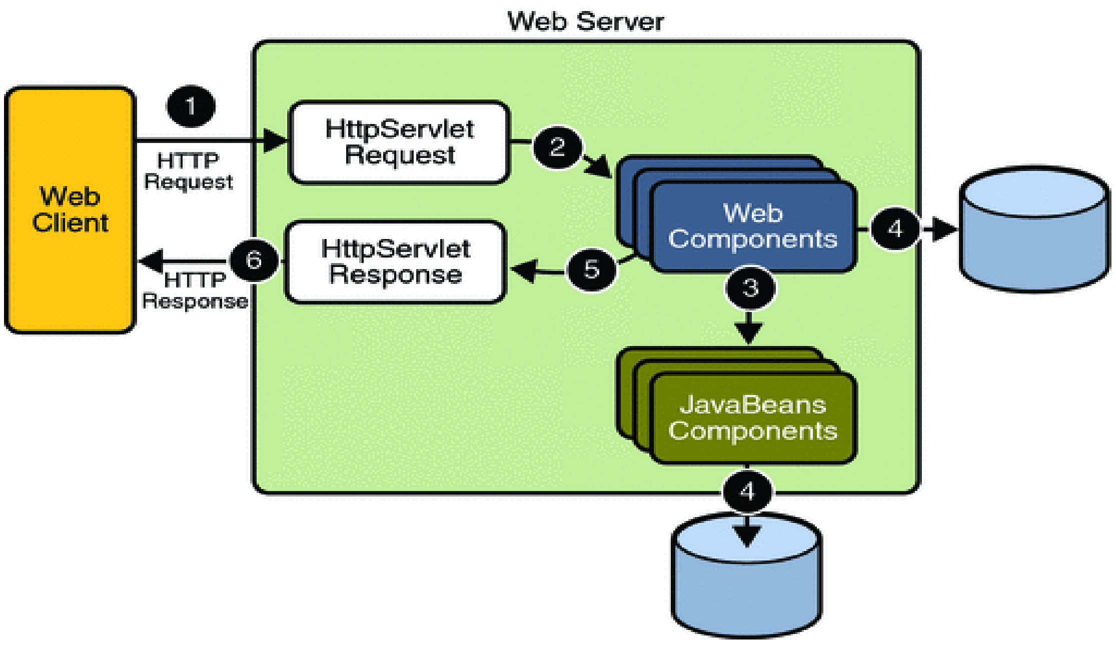
HTTPS Requirements: Moving from HTTP/1.1 will be more difficult for a site not using HTTPS since HTTP/2 has the prerequisite of the site already being on HTTPS. So switching a website to HTTPS will require all HTTP URLs to be converted to the HTTPS. As moving to HTTPS is a positive factor for search engines, therefore, it would be great to upgrade your site.

Page load performance could be minimal: Switching from HTTP/1.1 to HTTP/2 could take a lot of time and resources so it's important to be sure that the impact on site speed worth it. If a website is already optimized for page load speed, the performance improvements might be smaller than expected.

HTTP/2 Unknown facts: HTTP is still new so a lot of research will have to be conducted in order to determine best practices and possible drawbacks.

A dynamic website is one where some of the response content is generated dynamically only when needed. On a dynamic website HTML pages are normally created by inserting data from a database into placeholders in HTML templates (this is a much more efficient way of storing large amounts of content than using static websites).

1.3 Identify the advantages of HTTP in developing web applications



The advantage of using HTTP or HTTPS in your application is that you leverage and automatically can take advantage of a massive, worldwide scale infrastructure already there:

- Clients. If you use HTTP(S), then you can use existing browsers and high quality HTTP clients to debug, test, and use your application.
- Caching infrastructure. HTTP heavily relies on and takes advantage of a caching infrastructure deployed worldwide: local cache in browsers, org caches in HTTP proxies, ISP

caches, reverse proxies, etc. If you use HTTP(S) you can automatically takes advantage of this infrastructure.

- Compression. HTTP(S) support data compression using different algorithms, if you use HTTP(S) you don't need to implement your own solution.
- Security. If you use HTTPS you can take advantage of all the security features provided by HTTPS.
- Content negotiation. HTTP supports content negotiation, that is, clients and servers can negotiate the desired format for a given resource. Servers can provide the same resource (think of URL or URI) in different formats like images, JSON, XML or text; and clients can request a particular format.
- Firewalls traversal. In most firewalls, traffic is closed except for HTTP(S), if you use HTTP(S) your service will be accessible from most locations.
- Tools. There is a huge amount of tools developed around HTTP(S) that are at your disposal if you use these protocols.

This is not an exhaustive list, the main point is that if you use HTTP(S), then you "inherit" all features and can take advantage of all the infrastructure developed and deployed to support these protocols.

Advantages of HTTP/2

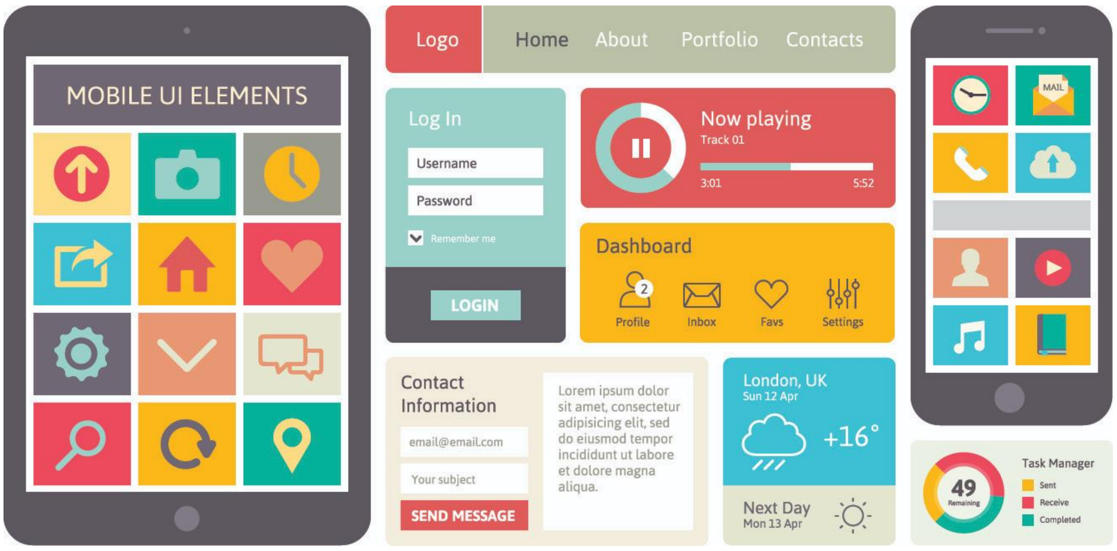
The main difference between the HTTP/1.1 and HTTP/2 is that the way in which data packages are framed and the transportation between the nodes. The high-level syntax of both the network protocol are supported for both. The client receives the data pushed by the server even when the browser request is not initiated. If there more than one requests, it is then multiplexed and pipelined over one TCP connection. This will adversely affect the speed of the web page delivery. For the usage of the HTTP/2 network protocol the server and the client side must have an understanding regarding it's standard. The latest updated browsers support HTTP/2 and thus the browser will load the web pages over HTTP/2 only if the server supports it.

CHAPTER 2: IMPLEMENT SESSION MANAGEMENT

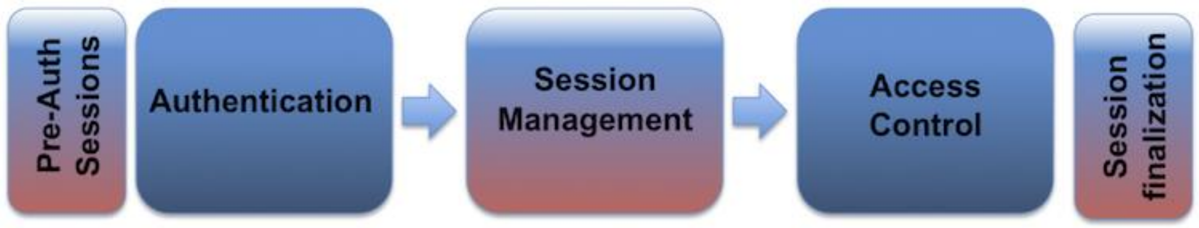
2.1 Create the code to handle session management

2.2 Create the code that retains the user's interaction with the website

2.3 Review and debug the code



Session management is the rule set that governs interactions between a web-based application and users. Browsers and websites use HTTP to communicate, and a web session is a series of HTTP requests and response transactions created by the same user. Since HTTP is a stateless protocol, where each request and response pair is independent of other web interactions, each command runs independently without knowing previous commands. In order to introduce the concept of a session, it is necessary to implement session management capabilities that link both the authentication and access control (or authorization) modules commonly available in web applications.



There are two types of session management – cookie-based and URL rewriting. These can be used independently or together. A web administrator uses session management to track the frequency of visits to a website and movement within the site.

Web application session management is another critical area that developers often overlook. HTTP does not have the capability to handle user authentication and keep track of user requests, which means Web applications must handle these tasks. Attackers can hijack active sessions unless user credentials and session identifiers are protected by properly implemented encryption.

Such session management vulnerabilities can be discovered by performing both code reviews and penetration tests, and particular focus should be paid to how session identifiers are handled and the methods used for changing users' credentials. To combat Web application session management issues, account management functions and transactions should require re-authentication, with two-factor authentication being a good option to enable for high-value transactions.

Session is a conversational state between client and server and it can consists of multiple request and response between client and server. Since HTTP and Web Server both are stateless, the only way to maintain a session is when some unique information about the session (session id) is passed between server and client in every request and response.

There are several ways through which we can provide unique identifier in request and response.

1. User Authentication – This is the very common way where we user can provide authentication credentials from the login page and then we can pass the authentication information between server and client to maintain the session. This is not very effective method because it won't work if the same user is logged in from different browsers.
2. HTML Hidden Field – We can create a unique hidden field in the HTML and when user starts navigating, we can set its value unique to the user and keep track of the session. This method can’t be used with links because it needs the form to be submitted every time request is made from client to server with the hidden field. Also it’s not secure because we can get the hidden field value from the HTML source and use it to hack the session.
3. URL Rewriting – We can append a session identifier parameter with every request and response to keep track of the session. This is very tedious because we need to keep track of this parameter in every response and make sure it’s not clashing with other parameters.
4. Cookies – Cookies are small piece of information that is sent by web server in response header and gets stored in the browser cookies. When client make further request, it adds the cookie to the request header and we can utilize it to keep track of the session. We can maintain a session with cookies but if the client disables the cookies, then it won’t work.
5. Session Management API – Session Management API is built on top of above methods for session tracking. Some of the major disadvantages of all the above methods are:

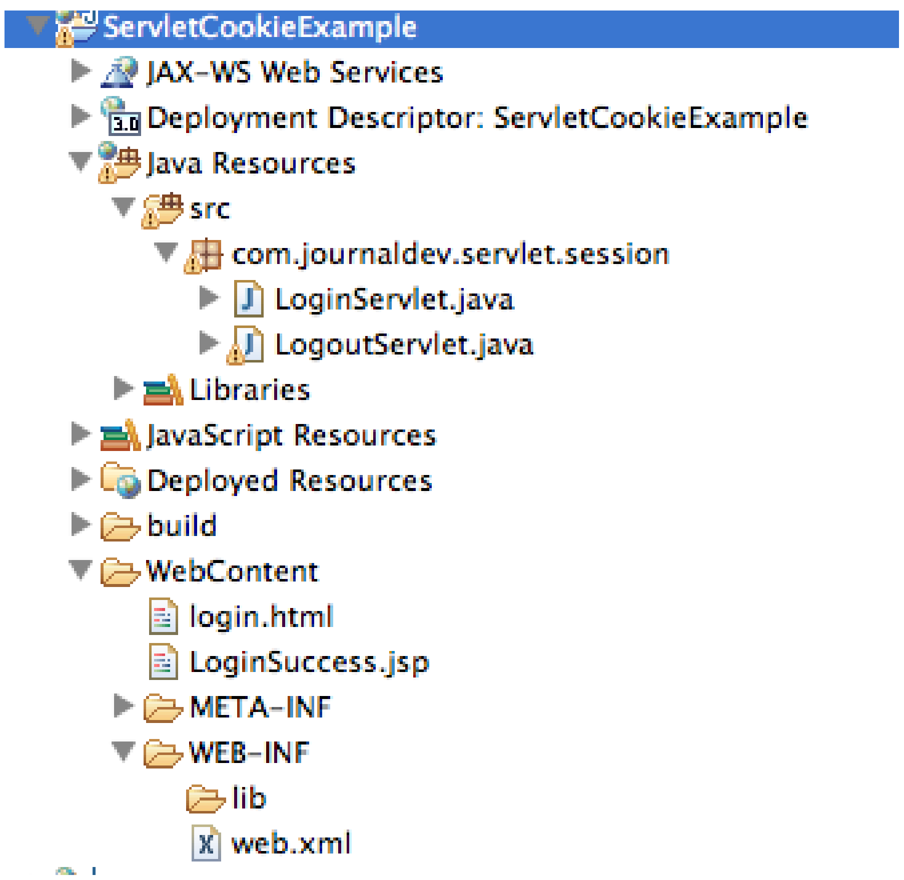
    - Most of the time we don’t want to only track the session, we have to store some data into the session that we can use in future requests. This will require a lot of effort if we try to implement this.
    - All the above methods are not complete in themselves, all of them won’t work in a particular scenario. So we need a solution that can utilize these methods of session tracking to provide session management in all cases.

That’s why we need Session Management API and J2EE Servlet technology comes with session management API that we can use.

Session Management in Java – Cookies

Cookies are used a lot in web applications to personalize response based on your choice or to keep track of session. Before moving forward to the Servlet Session Management API, I would like to show how can we keep track of session with cookies through a small web application.

We will create a dynamic web application ServletCookieExample with project structure like below image.



```
ServletCookieExample
  JAX-WS Web Services
  Deployment Descriptor: ServletCookieExample
  Java Resources
    src
      com.journaldev.servlet.session
        LoginServlet.java
        LogoutServlet.java
    Libraries
  JavaScript Resources
  Deployed Resources
  build
  WebContent
    login.html
    LoginSuccess.jsp
    META-INF
    WEB-INF
      lib
      web.xml
```

Deployment descriptor web.xml of the web application is:

Code:

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID" version="3.0">

  <display-name>ServletCookieExample</display-name>

  <welcome-file-list>

    <welcome-file>login.html</welcome-file>

  </welcome-file-list>

</web-app>
```

Welcome page of our application is login.html where we will get authentication details from user.

Code:

```
<!DOCTYPE html>

<html>

<head>

<meta charset="US-ASCII">

<title>Login Page</title>

</head>

<body>


<form action="LoginServlet" method="post">


Username: <input type="text" name="user">

<br>

Password: <input type="password" name="pwd">

<br>

<input type="submit" value="Login">

</form>

</body>

</html>
```

Here is the LoginServlet that takes care of the login request.

Code:

```
package com.journaldev.servlet.session;


import java.io.IOException;
import java.io.PrintWriter;


import javax.servlet.RequestDispatcher;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.Cookie;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;


/**

 * Servlet implementation class LoginServlet

 */

@WebServlet("/LoginServlet")

public class LoginServlet extends HttpServlet {

	private static final long serialVersionUID = 1L;

	private final String userID = "Pankaj";

	private final String password = "journaldev";


	protected void doPost(HttpServletRequest request,

				HttpServletResponse response) throws ServletException, IOException {


		// get request parameters for userID and password

		String user = request.getParameter("user");
		String pwd = request.getParameter("pwd");


		if(userID.equals(user) && password.equals(pwd)){
			Cookie loginCookie = new Cookie("user",user);
			//setting cookie to expiry in 30 mins
			loginCookie.setMaxAge(30*60);
			response.addCookie(loginCookie);
			response.sendRedirect("LoginSuccess.jsp");
		}else{
			RequestDispatcher rd = getServletContext().getRequestDispatcher("/login.html");
			PrintWriter out= response.getWriter();
			out.println("<font color=red>Either user name or password is wrong.</font>");
			rd.include(request, response);
		}


	}


}
```

Notice the cookie that we are setting to the response and then forwarding it to LoginSuccess.jsp, this cookie will be used there to track the session. Also notice that cookie timeout is set to 30 minutes. Ideally there should be a complex logic to set the cookie value for session tracking so that it won't collide with any other request.

Code:

```
<%@ page language="java" contentType="text/html; charset=US-ASCII"
```

```
   pageEncoding="US-ASCII"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=US-ASCII">

<title>Login Success Page</title>

</head>

<body>

<%

String userName = null;

Cookie[] cookies = request.getCookies();

if(cookies !=null){

for(Cookie cookie : cookies){

        if(cookie.getName().equals("user")) userName = cookie.getValue();

}

}

if(userName == null) response.sendRedirect("login.html");

%>

<h3>Hi <%=userName %>, Login successful.</h3>

<br>

<form action="LogoutServlet" method="post">

<input type="submit" value="Logout" >

</form>

</body>
```

```
</html>
```

Notice that if we try to access the JSP directly, it will forward us to the login page. When we will click on Logout button, we should make sure that cookie is removed from client browser.

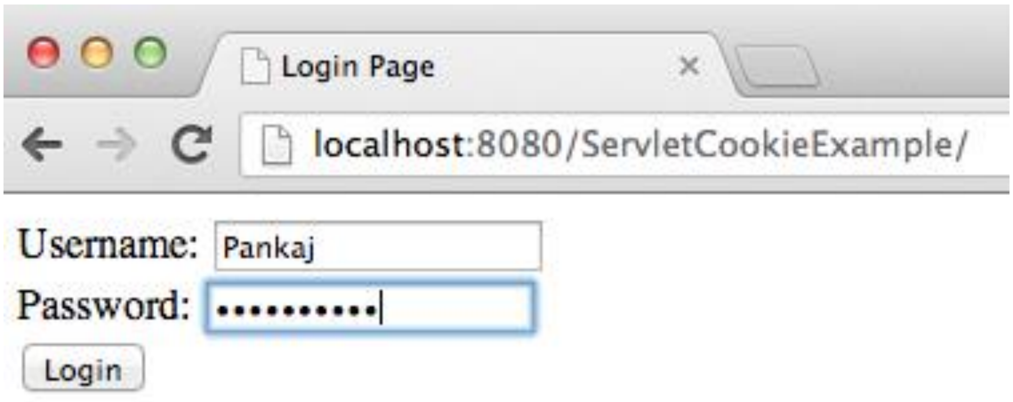
Code:

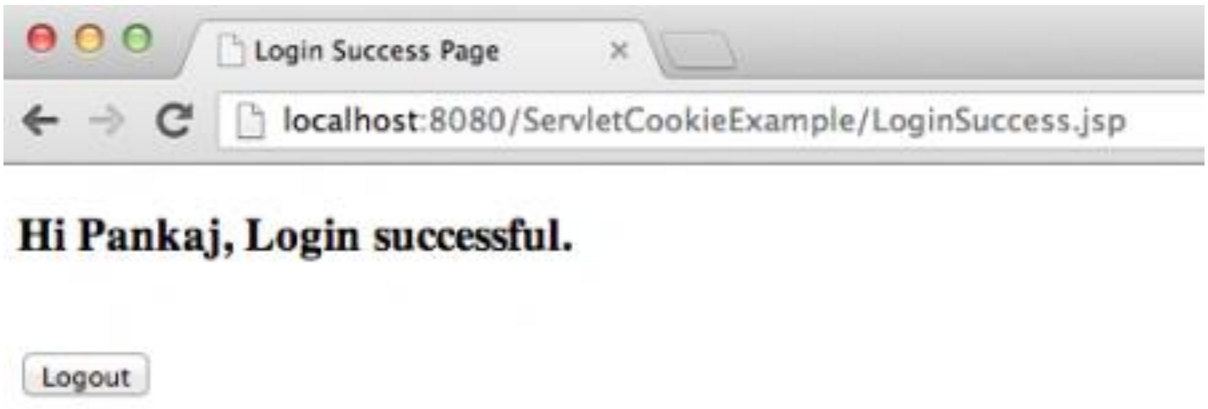
```
package com.journaldev.servlet.session;



import java.io.IOException;



import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.Cookie;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import javax.servlet.http.HttpSession;


/**

 * Servlet implementation class LogoutServlet

 */

@WebServlet("/LogoutServlet")

public class LogoutServlet extends HttpServlet {

        private static final long serialVersionUID = 1L;
```

```
  protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {

        response.setContentType("text/html");

        Cookie loginCookie = null;

        Cookie[] cookies = request.getCookies();

        if(cookies != null){

        for(Cookie cookie : cookies){

                if(cookie.getName().equals("user")){

                        loginCookie = cookie;

                        break;

                }

        }

        }

        if(loginCookie != null){

                loginCookie.setMaxAge(0);

        response.addCookie(loginCookie);

        }

        response.sendRedirect("login.html");

  }


}
```

There is no method to remove the cookie but we can set the maximum age to 0 so that it will be deleted from client browser immediately.

When we run above application, we get response like below images.




Login Success Page
localhost:8080/ServletCookieExample/LoginSuccess.jsp
Hi Pankaj, Login successful.
Logout

Session in Java Servlet – HttpSession

Servlet API provides Session management through HttpSession interface. We can get session from HttpServletRequest object using following methods. HttpSession allows us to set objects as attributes that can be retrieved in future requests.

HttpSession getSession() – This method always returns a HttpSession object. It returns the session object attached with the request, if the request has no session attached, then it creates a new session and return it.

HttpSession getSession(boolean flag) – This method returns HttpSession object if request has session else it returns null.

Some of the important methods of HttpSession are:

String getId() – Returns a string containing the unique identifier assigned to this session.

Object getAttribute(String name) – Returns the object bound with the specified name in this session, or null if no object is bound under the name. Some other methods to work with Session attributes are getAttributeNames(), removeAttribute(String name) and setAttribute(String name, Object value).

long getCreationTime() – Returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT. We can get last accessed time with getLastAccessedTime() method.

setMaxInactiveInterval(int interval) – Specifies the time, in seconds, between client requests before the servlet container will invalidate this session. We can get session timeout value from getMaxInactiveInterval() method.

ServletContext getServletContext() – Returns ServletContext object for the application.

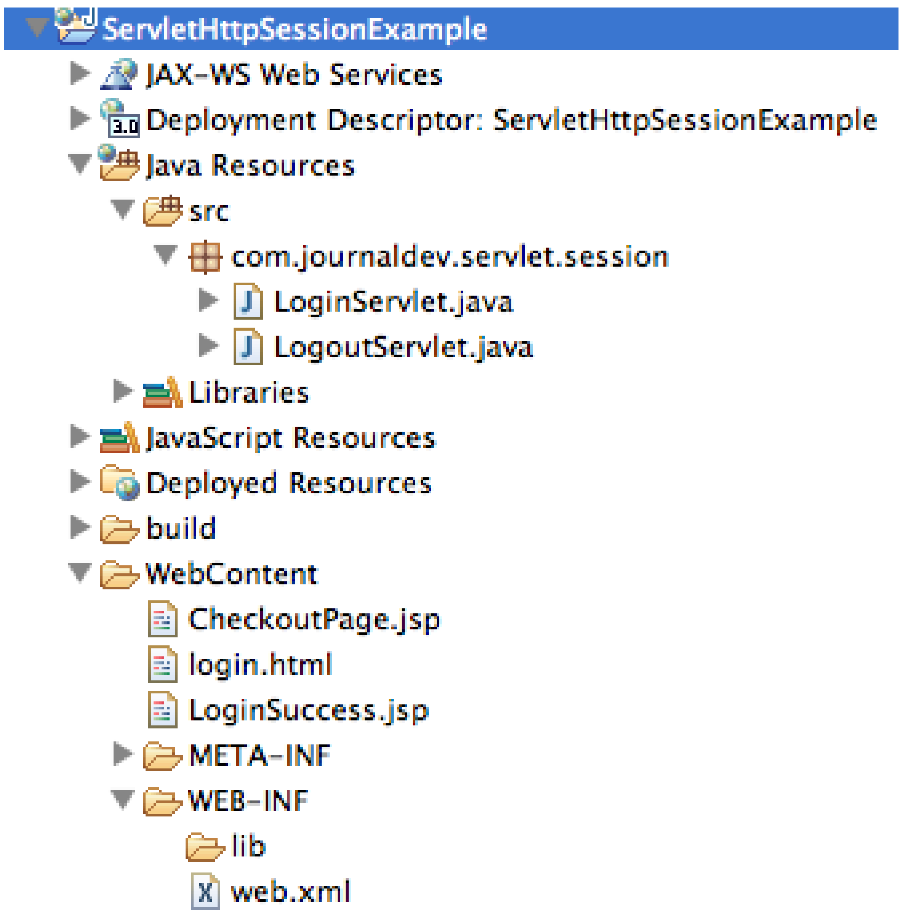
boolean isNew() – Returns true if the client does not yet know about the session or if the client chooses not to join the session.

void invalidate() – Invalidates this session then unbinds any objects bound to it.

Understanding JSESSIONID Cookie

When we use HttpServletRequest getSession() method and it creates a new request, it creates the new HttpSession object and also add a Cookie to the response object with name JSESSIONID and value as session id. This cookie is used to identify the HttpSession object in further requests from client. If the cookies are disabled at client side and we are using URL rewriting then this method uses the jsessionid value from the request URL to find the corresponding session. JSESSIONID cookie is used for session tracking, so we should not use it for our application purposes to avoid any session related issues.

Let’s see example of session management using HttpSession object. We will create a dynamic web project in Eclipse with servlet context as ServletHttpSessionExample. The project structure will look like below image.



login.html is same like earlier example and defined as welcome page for the application in web.xml

LoginServlet servlet will create the session and set attributes that we can use in other resources or in future requests.

Code:

```
package com.journaldev.servlet.session;



import java.io.IOException;

import java.io.PrintWriter;



import javax.servlet.RequestDispatcher;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;
import javax.servlet.http.Cookie;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import javax.servlet.http.HttpSession;


/**

 * Servlet implementation class LoginServlet

 */

@WebServlet("/LoginServlet")

public class LoginServlet extends HttpServlet {

	private static final long serialVersionUID = 1L;

	private final String userID = "admin";

	private final String password = "password";


	protected void doPost(HttpServletRequest request,

			HttpServletResponse response) throws ServletException, IOException {


		// get request parameters for userID and password

		String user = request.getParameter("user");

		String pwd = request.getParameter("pwd");


		if(userID.equals(user) && password.equals(pwd)){

			HttpSession session = request.getSession();
```

```
                        session.setAttribute("user", "Pankaj");

                        //setting session to expiry in 30 mins

                        session.setMaxInactiveInterval(30*60);

                        Cookie userName = new Cookie("user", user);

                        userName.setMaxAge(30*60);

                        response.addCookie(userName);

                        response.sendRedirect("LoginSuccess.jsp");

                }else{

                        RequestDispatcher rd = getServletContext().getRequestDispatcher("/login.html");

                        PrintWriter out= response.getWriter();

                        out.println("<font color=red>Either user name or password is wrong.</font>");

                        rd.include(request, response);

                }


        }


}
```

Our LoginSuccess.jsp code is given below.

Code:

```
<%@ page language="java" contentType="text/html; charset=US-ASCII"

   pageEncoding="US-ASCII"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>
```

```
<head>

<meta http-equiv="Content-Type" content="text/html; charset=US-ASCII">

<title>Login Success Page</title>

</head>

<body>

<%

//allow access only if session exists

String user = null;

if(session.getAttribute("user") == null){

        response.sendRedirect("login.html");

}else user = (String) session.getAttribute("user");

String userName = null;

String sessionID = null;

Cookie[] cookies = request.getCookies();

if(cookies !=null){

for(Cookie cookie : cookies){

        if(cookie.getName().equals("user")) userName = cookie.getValue();

        if(cookie.getName().equals("JSESSIONID")) sessionID = cookie.getValue();

}

}

%>

<h3>Hi <%=userName %>, Login successful. Your Session ID=<%=sessionID %></h3>

<br>

User=<%=user %>
```

```
<br>

<a href="CheckoutPage.jsp">Checkout Page</a>

<form action="LogoutServlet" method="post">

<input type="submit" value="Logout" >

</form>

</body>

</html>
```

When a JSP resource is used, container automatically creates a session for it, so we can’t check if session is null to make sure if user has come through login page, so we are using session attribute to validate request.

CheckoutPage.jsp is another page and it’s code is given below.

Code:

```
<%@ page language="java" contentType="text/html; charset=US-ASCII"

    pageEncoding="US-ASCII"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=US-ASCII">

<title>Login Success Page</title>

</head>

<body>

<%

//allow access only if session exists

if(session.getAttribute("user") == null){
        response.sendRedirect("login.html");
}
String userName = null;
String sessionID = null;
Cookie[] cookies = request.getCookies();
if(cookies !=null){
for(Cookie cookie : cookies){
        if(cookie.getName().equals("user")) userName = cookie.getValue();
}
}
%>
<h3>Hi <%=userName %>, do the checkout.</h3>
<br>
<form action="LogoutServlet" method="post">
<input type="submit" value="Logout" >
</form>
</body>
</html>
```

Our LogoutServlet code is given below.

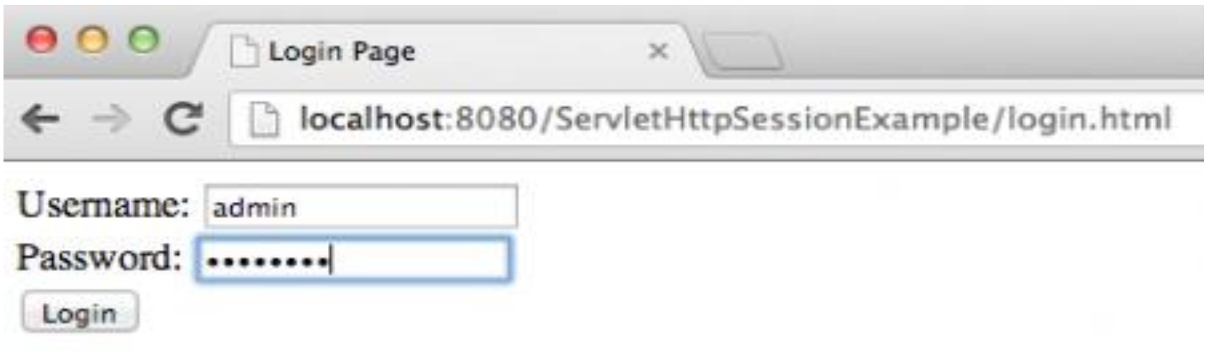
Code:

```
package com.journaldev.servlet.session;


import java.io.IOException;
import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.Cookie;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import javax.servlet.http.HttpSession;


/**

 * Servlet implementation class LogoutServlet

 */

@WebServlet("/LogoutServlet")

public class LogoutServlet extends HttpServlet {

        private static final long serialVersionUID = 1L;


   protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {

        response.setContentType("text/html");

        Cookie[] cookies = request.getCookies();

        if(cookies != null){

        for(Cookie cookie : cookies){

                if(cookie.getName().equals("JSESSIONID")){

                        System.out.println("JSESSIONID="+cookie.getValue());

                        break;
```

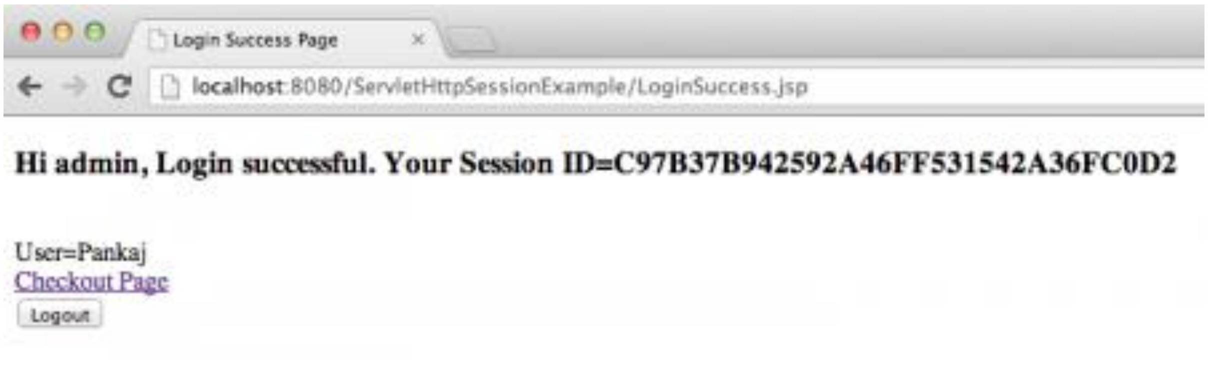
```
            }

        }

        }

        //invalidate the session if exists

        HttpSession session = request.getSession(false);

        System.out.println("User="+session.getAttribute("user"));

        if(session != null){

            session.invalidate();

        }

        response.sendRedirect("login.html");

    }


}
```

Notice that I am printing JSESSIONID cookie value in logs, you can check server log where it will be printing the same value as Session Id in LoginSuccess.jsp

Below images shows the execution of our web application.

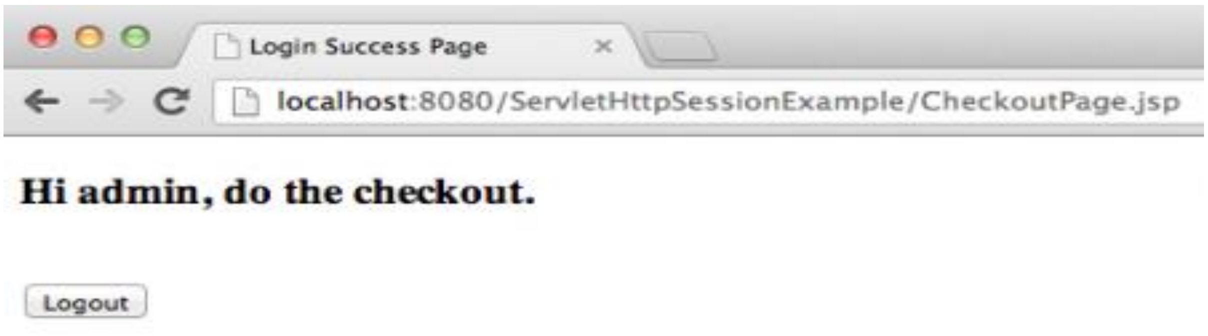


Username: admin
Password: ••••••••



Hi admin, Login successful. Your Session ID=C97B37B942592A46FF531542A36FC0D2

User=Pankaj
Checkout Page



Hi admin, do the checkout.

Session Management in Java Servlet – URL Rewriting

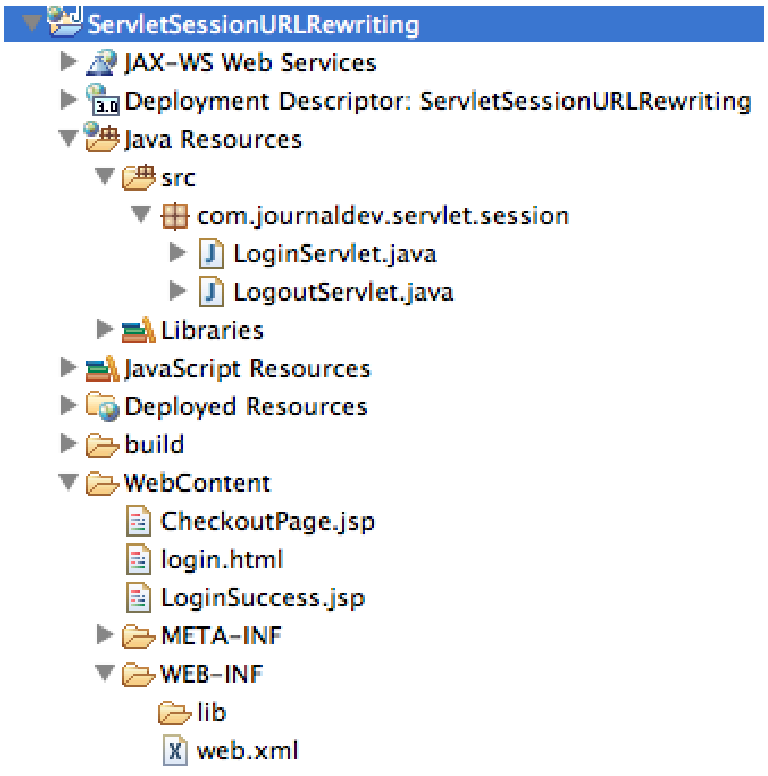
As we saw in last section that we can manage a session with HttpSession but if we disable the cookies in browser, it won’t work because server will not receive the JSESSIONID cookie from client. Servlet API provides support for URL rewriting that we can use to manage session in this case.

The best part is that from coding point of view, it’s very easy to use and involves one step – encoding the URL. Another good thing with Servlet URL Encoding is that it’s a fallback approach and it kicks in only if browser cookies are disabled.

We can encode URL with HttpServletResponse encodeURL() method and if we have to redirect the request to another resource and we want to provide session information, we can use encodeRedirectURL() method.

We will create a similar project like above except that we will use URL rewriting methods to make sure session management works fine even if cookies are disabled in browser.

ServletSessionURLRewriting project structure in eclipse looks like below image.



```
ServletSessionURLRewriting
  JAX-WS Web Services
  Deployment Descriptor: ServletSessionURLRewriting
  Java Resources
    src
      com.journaldev.servlet.session
        LoginServlet.java
        LogoutServlet.java
    Libraries
  JavaScript Resources
  Deployed Resources
  build
  WebContent
    CheckoutPage.jsp
    login.html
    LoginSuccess.jsp
    META-INF
    WEB-INF
      lib
      web.xml
```

Code:

```
package com.journaldev.servlet.session;



import java.io.IOException;

import java.io.PrintWriter;



import javax.servlet.RequestDispatcher;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.Cookie;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;


/**
 * Servlet implementation class LoginServlet
 */
@WebServlet("/LoginServlet")
public class LoginServlet extends HttpServlet {
	private static final long serialVersionUID = 1L;
	private final String userID = "admin";
	private final String password = "password";

	protected void doPost(HttpServletRequest request,
			HttpServletResponse response) throws ServletException, IOException {

		// get request parameters for userID and password
		String user = request.getParameter("user");
		String pwd = request.getParameter("pwd");

		if(userID.equals(user) && password.equals(pwd)){
			HttpSession session = request.getSession();
			session.setAttribute("user", "Pankaj");
			//setting session to expiry in 30 mins
			session.setMaxInactiveInterval(30*60);
			Cookie userName = new Cookie("user", user);
```

```
                        response.addCookie(userName);

                        //Get the encoded URL string

                        String encodedURL = response.encodeRedirectURL("LoginSuccess.jsp");

                        response.sendRedirect(encodedURL);

                }else{

                        RequestDispatcher rd = getServletContext().getRequestDispatcher("/login.html");

                        PrintWriter out= response.getWriter();

                        out.println("<font color=red>Either user name or password is wrong.</font>");

                        rd.include(request, response);

                }


        }


}
```

Code:

```
<%@ page language="java" contentType="text/html; charset=US-ASCII"

    pageEncoding="US-ASCII"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=US-ASCII">

<title>Login Success Page</title>

</head>
```

```
<body>

<%

//allow access only if session exists

String user = null;

if(session.getAttribute("user") == null){

        response.sendRedirect("login.html");

}else user = (String) session.getAttribute("user");

String userName = null;

String sessionID = null;

Cookie[] cookies = request.getCookies();

if(cookies !=null){

for(Cookie cookie : cookies){

        if(cookie.getName().equals("user")) userName = cookie.getValue();

        if(cookie.getName().equals("JSESSIONID")) sessionID = cookie.getValue();

}

}else{

        sessionID = session.getId();

}

%>

<h3>Hi <%=userName %>, Login successful. Your Session ID=<%=sessionID %></h3>

<br>

User=<%=user %>

<br>

<!-- need to encode all the URLs where we want session information to be passed -->
```

```
<a href="<%=response.encodeURL("CheckoutPage.jsp") %>">Checkout Page</a>

<form action="<%=response.encodeURL("LogoutServlet") %>" method="post">

<input type="submit" value="Logout" >

</form>

</body>

</html>
```

Code:

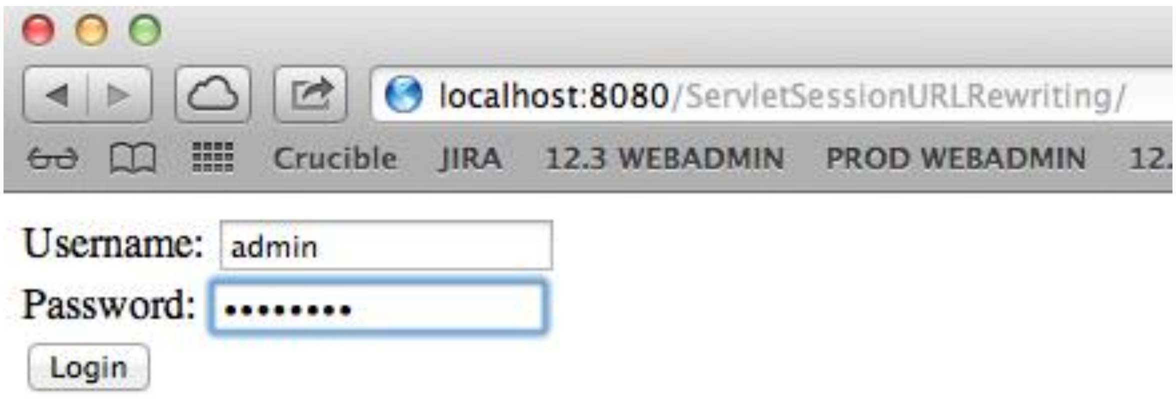
```
<%@ page language="java" contentType="text/html; charset=US-ASCII"

    pageEncoding="US-ASCII"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=US-ASCII">

<title>Login Success Page</title>

</head>

<body>

<%

String userName = null;

//allow access only if session exists

if(session.getAttribute("user") == null){

	response.sendRedirect("login.html");

}else userName = (String) session.getAttribute("user");

String sessionID = null;
```

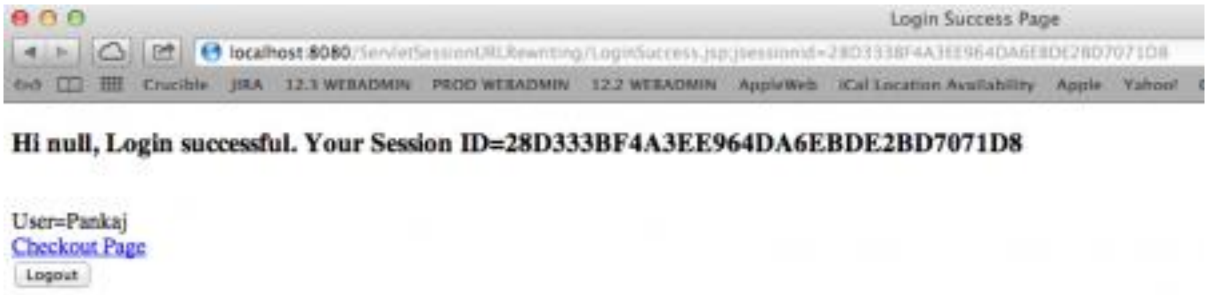
```
Cookie[] cookies = request.getCookies();

if(cookies !=null){

for(Cookie cookie : cookies){

	if(cookie.getName().equals("user")) userName = cookie.getValue();

}

}

%>

<h3>Hi <%=userName %>, do the checkout.</h3>

<br>

<form action="<%=response.encodeURL("LogoutServlet") %>" method="post">

<input type="submit" value="Logout" >

</form>

</body>

</html>
```

Code:

```
package com.journaldev.servlet.session;


import java.io.IOException;


import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import javax.servlet.http.HttpSession;


/**

 * Servlet implementation class LogoutServlet

 */

@WebServlet("/LogoutServlet")

public class LogoutServlet extends HttpServlet {

        private static final long serialVersionUID = 1L;


   protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {

        response.setContentType("text/html");

        Cookie[] cookies = request.getCookies();

        if(cookies != null){

        for(Cookie cookie : cookies){

                if(cookie.getName().equals("JSESSIONID")){

                        System.out.println("JSESSIONID="+cookie.getValue());

                }

                cookie.setMaxAge(0);

                response.addCookie(cookie);

        }

        }
        //invalidate the session if exists
        HttpSession session = request.getSession(false);
        System.out.println("User="+session.getAttribute("user"));
        if(session != null){
                session.invalidate();
        }
        //no encoding because we have invalidated the session
        response.sendRedirect("login.html");
    }

}
```

When we run this project keeping cookies disabled in the browser, below images shows the response pages, notice the jsessionid in URL of browser address bar. Also notice that on LoginSuccess page, user name is null because browser is not sending the cookie send in the last response.

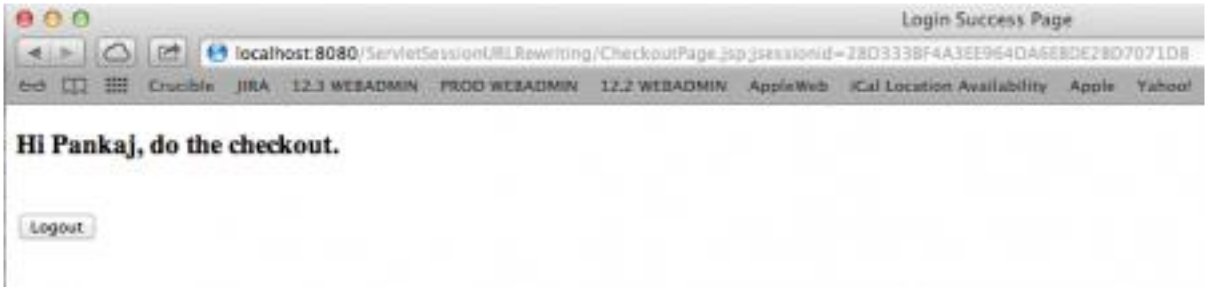



Login Success Page
localhost:8080/ServletSessionURLRewriting/LoginSuccess.jsp;jsessionid=28D333BF4A3EE964DA6EBDE2BD7071D8
Crucible
JIRA
12.3 WEBADMIN
PROD WEBADMIN
12.2 WEBADMIN
AppleWeb
iCal Location Availability
Apple
Yahoo!

Hi null, Login successful. Your Session ID=28D333BF4A3EE964DA6EBDE2BD7071D8

User=Pankaj

Checkout Page


Login Success Page
localhost:8080/ServletSessionURLRewriting/CheckoutPage.jsp;jsessionid=28D333BF4A3EE964DA6EBDE2BD7071D8
Crucible
JIRA
12.3 WEBADMIN
PROD WEBADMIN
12.2 WEBADMIN
AppleWeb
iCal Location Availability
Apple
Yahoo!

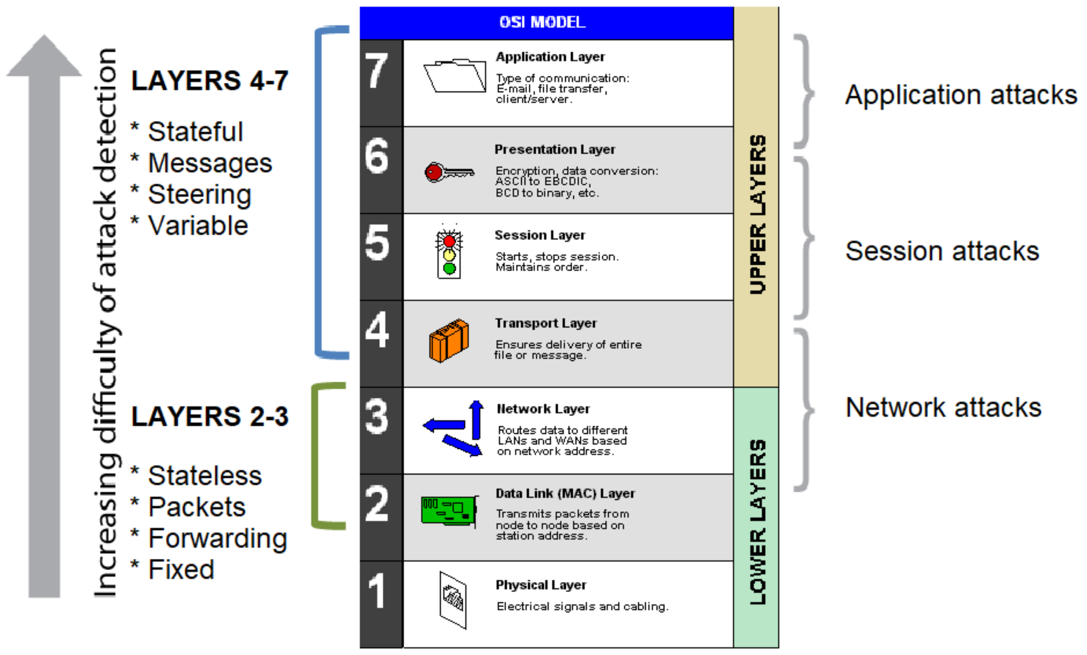
Hi Pankaj, do the checkout.

If cookies are not disabled, you won't see j_session_id in the URL because Servlet Session API will use cookies in that case.

CHAPTER 3: DEVELOP APPLICATIONS IN A STATELESS ENVIRONMENT

3.1. Develop web applications that keep track of data between browser requests

3.2. Document the web application, with particular reference to its management of statelessness


Increasing difficulty of attack detection
LAYERS 4-7
* Stateful
* Messages
* Steering
* Variable
LAYERS 2-3
* Stateless
* Packets
* Forwarding
* Fixed
OSI MODEL
7 Application Layer Type of communication: E-mail, file transfer, client/server.
6 Presentation Layer Encryption, data conversion: ASCII to EBCDIC, BCD to binary, etc.
5 Session Layer Starts, stops session. Maintains order.
4 Transport Layer Ensures delivery of entire file or message.
3 Network Layer Routes data to different LANs and WANs based on network address.
2 Data Link (MAC) Layer Transmits packets from node to node based on station address.
1 Physical Layer Electrical signals and cabling.
UPPER LAYERS
LOWER LAYERS
Application attacks
Session attacks
Network attacks

The stateful and stateless models of software application behaviour define how a user's web browser communicates with a web server. In the earliest years of the Web, sites tended to be stateless. Pages were static, not varying from user to user. Later, websites included the stateful model, which delivered pages with information unique to each user. Stateful web applications are essential for modern e-commerce such as online retailers and banks but require sophisticated programming to work effectively.

Stateless Model

In the pure form of the stateless model, a client program makes a request to an application server, which sends data back to the client. The server treats all client connections equally and saves no information from prior requests or sessions. A website that serves up a simple static web page is a good example of the stateless model. The server receives requests for pages it hosts and sends the page data to requesting browsers, much like a short-order cook making meals for diners.

Stateless

The answer to these issues is statelessness. Stateless is the polar opposite of stateful, in which any given response from the server is independent of any sort of state.

Let’s go back to that binary room theoretical. You are given the same binary clock, only this time, the paper simply has a name — “Jack” — and the instructions are to respond when someone says the password “fish”. You sit watching the clock slowly change, and each time someone says the special password, you say the name “Jack”.

This is statelessness — there’s no need to even reference the clock, because the information is stored locally in such a way that the requests are self contained — it’s dependent only on the data you hold. The speaker could easily say the secret word, tell you to change the name, then walk away. He can then come back an hour later, say the secret password, and get the new name — everything is contained within the request, and handled in two distinct phases, with a “request” and a “response”.

This is a stateless system. Your response is independent of the “0” or “1”, and each request is self-contained.

Stateless Web Services

Statelessness is a fundamental aspect of the modern internet — so much so that every single day, you use a variety of stateless services and applications. When you read the news, you are using HTTP to connect in a stateless manner, utilizing messages that can be parsed and worked with in isolation of each other and your state.

If you have Twitter on your phone, you are constantly utilizing a stateless service. When the service requests a list of recent direct messages using the Twitter REST API, it issues the following request:

GET https://api.twitter.com/1.1/direct_messages.json?since_id=240136858829479935&amp;count=1

The response that you will get is entirely independent of any server state storage, and everything is stored on the client’s side in the form of a cache.

Let’s take a look at another example. In the example below, we are invoking a POST command, creating a record on HypotheticalService:

```
POST http://HypotheticalService/Entity

Host: HypotheticalService

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

Content-Length: 123

<!--?xml version="1.0" encoding="utf-8"?-->

<entity>

  <id>1</id>

  <title>Example</title>
  <content>This is an example</content>

</entity>
```

In this example, we are creating an entry, but this entry does not depend on any matter of state. Do keep in mind that this is a simple use case, as it does not pass any authorization/authentication data, and the POST issuance itself contains only very basic data.

Even with all of this in mind, you can plainly see that doing a POST issuance in a stateless manner means that you do not have to wait for server synchronization to ensure the process has been properly completed, as you would with FTP or other stateful services. You receive a confirmation, but this confirmation is simply an affirmative response, rather than a mutual shared state.

As a quick note, it must be said that REST is specifically designed to be functionally stateless. The entire concept of Representational State Transfer (from which REST gets its name) hinges on the idea of passing all data to handle the request in such a way as to pair the data within the request itself. Thus, REST should be considered stateless (and, in fact, that is one of the main considerations as to whether something is RESTful or not as per the original dissertation by Roy Fielding which detailed the concept).

Stateful Model

When an application operates in a stateful mode, the server keeps track of who users are and what they do from one screen to the next. Preserving the state of users' actions is fundamental to having a meaningful, continuous session. It typically begins with a login with user ID and password, establishing a beginning state to the session. As a user navigates through the site, the state may change. The server maintains the state of the user's information throughout the session until logout.

Stateful

To understand statelessness, one must understand statefulness. When we talk about computer systems, a "state" is simply the condition or quality of an entity at an instant in time, and to be stateful is to rely on these moments in time and to change the output given the determined inputs and state.

If that's unclear, don't worry — it's a hard concept to grasp, and doubly so with APIs. We can break this down even further — consider binary, a language of 1's and 0's. What this functionally represents is either "on" or "off" — a binary system cannot be both 1 and 0, and are so mutually exclusive.

Now, consider a theoretical situation in which you are given a piece of paper with these simple instructions — "if the number is 0, say no, if 1, say yes" — and you were placed into a room with a binary display which changed between the number 0 and 1 every five seconds.

This is a stateful system. Your answer will depend entirely on whether or not that clock says "0" or "1" — you cannot answer independently of the state of the grand machine. This is statefulness.

Stateful Web Services

With this in mind, what does a stateful web service looks like? Let's say you log into a resource, and in doing so, you pass your password and username. If the web server stores this data in a backend manner and uses it to identify you as a constantly connected client, the service is stateful. Do keep in mind that this is a very specific example that exists in other forms, so what seems stateful may not necessarily be stateful — more on this later.

As you use the web service, everything you do is referenced back to this stored state. When you request an account summary, the web service asks two things:

Who is making this request?

Using the ID stored for who is making this request, what should their page look like?

In a stateful web service like this, the response formed from a simple GET request is entirely dependent on the state registered by the server. Without knowledge of that state, your request cannot be returned properly.

Another great example is FTP. When a user logs in to a traditional FTP server, they are engaging in an active connection with the server. Each change to the state of the user, such as active directory, is stored on the server as a client state. Each change made to the server is registered as a change of state, and when the user disconnects, their state is further changed to disconnected.

So far so good, right? Well, not quite. Stateful programming is fine in some very limited applications, but it has a lot of issues. First and foremost, when you have to reference a state, you're opening yourself up to a lot of incomplete sessions and transactions. Let's say you make a call to present a piece of data. In a stateful system where the state is determined by the client, how long is the system supposed to leave this connection open? How do we verify if the client has crashed or disconnected? How do we track the actions of the user while maintaining the ability to document changes and roll back when necessary?

While there are certainly workarounds for all of these questions, more often than not, statefulness is only really useful when the functions themselves depend on the statefulness quality. Most consumers are able to respond to the server in intelligent, dynamic ways, and because of this, maintaining server state independent of the consumer as if the consumer was simply a "dumb" client is wasteful and unnecessary.

Getting States into Stateless

Many websites generate pages dynamically, relying less on static HTML files. The user's browser receives page data from a web server and renders it as it would a static page. In addition, dynamic pages allow the server to "remember" the user and create continuity from page to page. To make continuity work, a developer can employ a number of tricks, adding stateful features to the stateless model. For example, when a user logs into a bank's website, the server creates information about this session. The information may reside in the server's memory, but it may also be stored in the browser. The server keeps track of your session by recording unique information about the session, such as a user's IP address. In addition, the developer can create additional server-side variables used by the programs on the server. These variables remain intact as long as a session is active, vanishing upon logout or session expiration.

Browser-Side Data

A few different options exist for keeping session data in the browser. One is the cookie, a special type of browser token. A web server may create a cookie and store it in the user's browser so that it can query it during the session. A browser stores these cookies and returns them upon server request. Cookies can optionally expire with the session or persist nearly permanently. If they remain, however, the data can be read by any site, potentially revealing information the user might not otherwise share. Many companies' indiscriminate and unauthorized data-gathering from the use of cookies has led to a sense of distrust from many Internet users. Many users, in turn, have responded by setting their browsers to block cookies, making cookies less available for legitimate purposes.

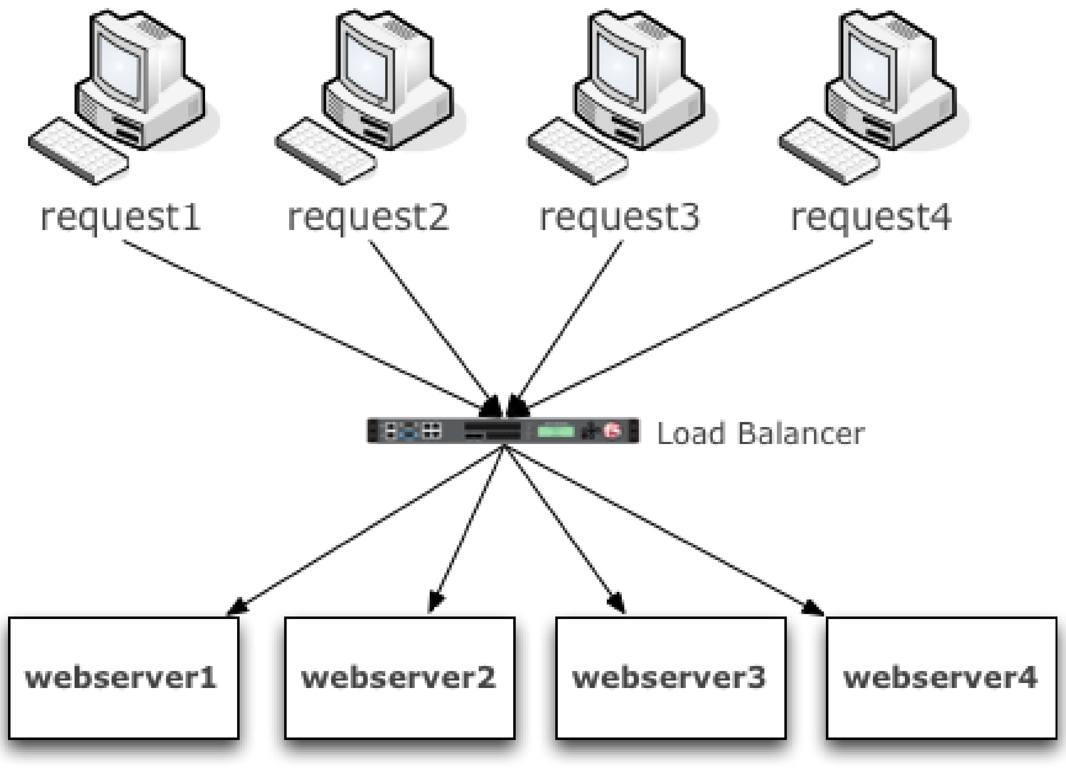
In addition to cookies, you can communicate session data to a server by putting variables in the URL, set off with a question mark, such as in this example:

http://mysite.net/anypage.php?userID=54321&orderID=51926

Note that the very first variable is set off with a question mark and subsequent variables use an ampersand.

Many sites use URLs to pass data from client to server. One drawback is that the URL data is visible to anyone. It is possible in some instances to exploit URL data to obtain unauthorized information or otherwise subvert a site's security. Also, a hacker who monitors your Internet traffic can intercept URL data easily. You can, however, obscure URL data by passing it in encrypted or hashed form, improving its security.

Server Load



When running websites that use the stateless mode, a cloud server has a job that is relatively easy, serving up pages without the need to save the user's data. A stateful site has additional memory overhead for each user, needed to store basic session data plus user responses and other data required by the application. For sites that handle thousands of simultaneous sessions, the additional memory requirements add up. A stateless site that relies primarily on static pages is also easier on the server’s CPU. By contrast, a stateful site depends more heavily on dynamic, program-generated content, requiring much more CPU horsepower per session. A savvy operations staff keeps a close watch on server memory and CPU performance data, looking for bottlenecks that can affect a site’s responsiveness.

Smoke and Mirrors

We need to be somewhat careful when we talk about web services as examples of stateful or stateless, though, because what seems to fall in one category may not actually be so. This is largely because stateless services have managed to mirror a lot of the behaviour of stateful services without technically crossing the line.

Statelessness is, just like our example above, all about self-contained state and reference rather than depending on an external frame of reference. The difference between it and statefulness is really where the state is stored. When we browse the internet or access our mail, we are generating a state — and that state has to go somewhere.

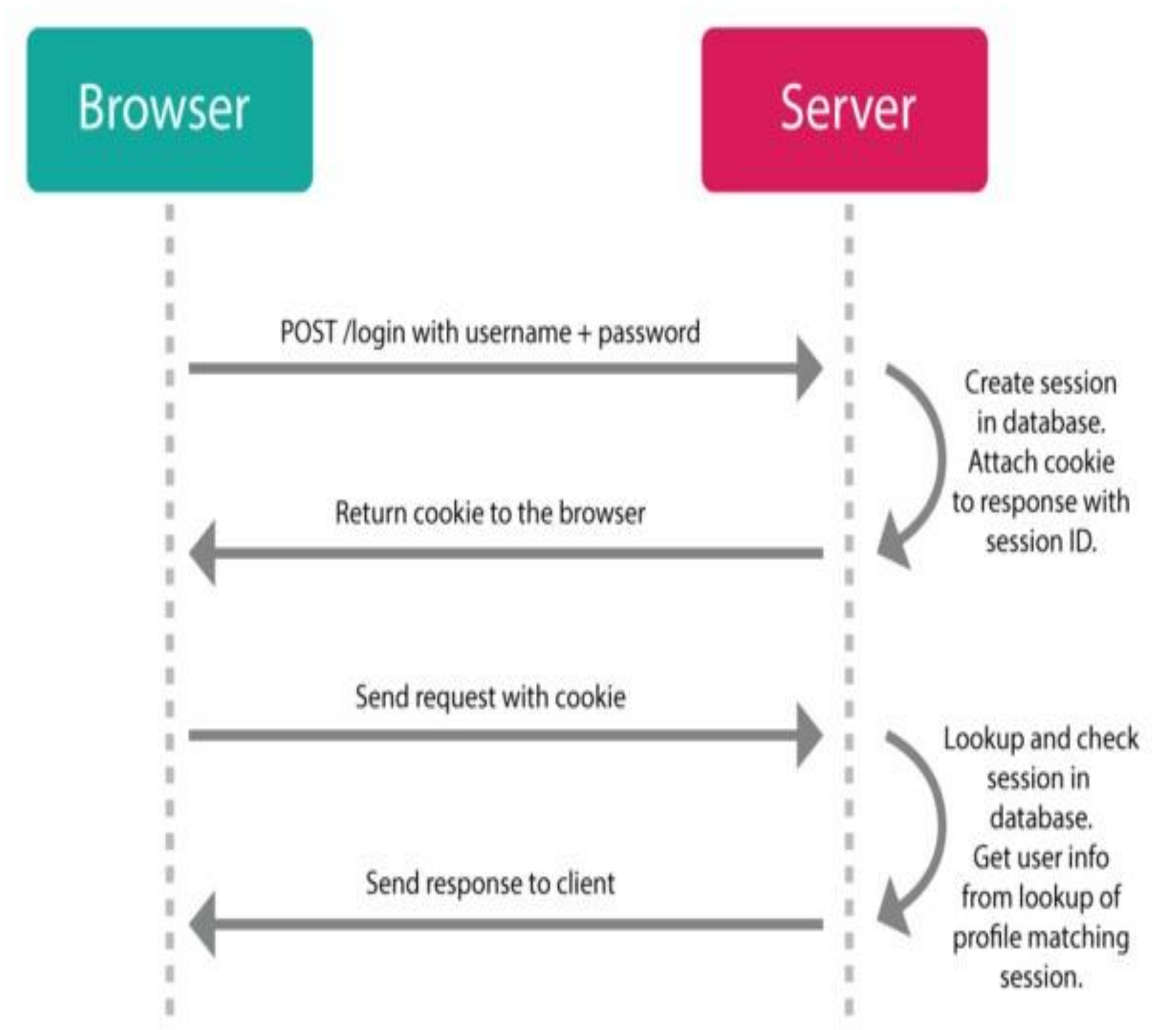
When the state is stored by the server, it generates a session. This is stateful computing. When the state is stored by the client, it generates some kind of data that is to be used for various systems — while technically “stateful” in that it references a state, the state is stored by the client so we refer to it as stateless.

This seems confusing, but it’s actually the best way to work around the limitations of statelessness. In a purely stateless system, we’re essentially interacting with a limited system — when we would order an online good, that’d be it for us, it wouldn’t store our address, our payment methods, even a record of our order, it would simply process our payment and, as far as the server was concerned, we’d cease to be.

That’s obviously not the best-case scenario, and so, we made some concessions. In the client cookie, we store some basic authentication data. On the server side, we create some temporary client data or store on a database, and reference it to an external piece of data. When we return to make another payment, it’s our cookie that establishes the state, not the non-existent session.

The use of Sessions

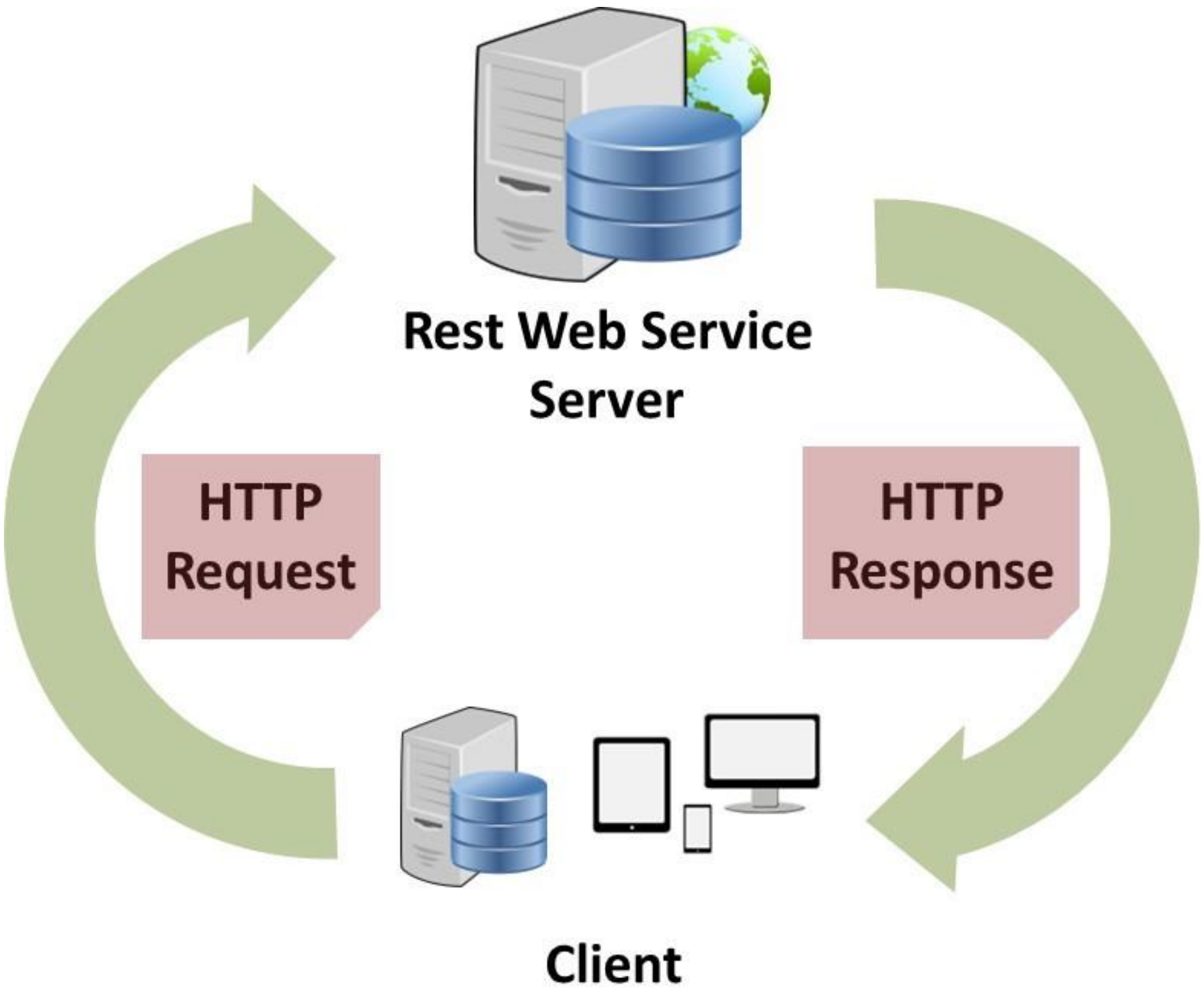
The Web was originally designed to be stateless, that is, not to rely on information about users being stored between interactions. Sessions are a way to save the state of the application for the user between requests (shopping carts being the classic example of this). Programmers, feeling that they need to save information about particular users between interactions, find all sorts of ways to use sessions that are not really appropriate. As a result, the concept of sessions gets abused and the scalability of the applications suffers due to requiring sessions. If you can design your application not to use sessions at all and be completely stateless, it can be scaled horizontally much more easily and the application will run much better under high load than if you try to keep track of which user needs to be reconnected with which session settings.


Browser
Server
POST /login with username + password
Create session in database. Attach cookie to response with session ID.
Return cookie to the browser
Send request with cookie
Lookup and check session in database. Get user info from lookup of profile matching session.
Send response to client

The error that we often see is that the application programmer makes a dependency on sessions central to the design of the software, and later when it becomes identified as a scalability bottleneck, it cannot be removed from the application because it was designed in at such a low level.

The reason why sessions are so evil is that most developers store the session information in the database. To get the information you need out of the session, you need to query the database and read the information. And because sessions change so often, such applications require lots and lots of writing to the database to keep the state of the session. You can end up with a read/write ratio of about 50/50 because of sessions, which kills scalability.

Working with REST

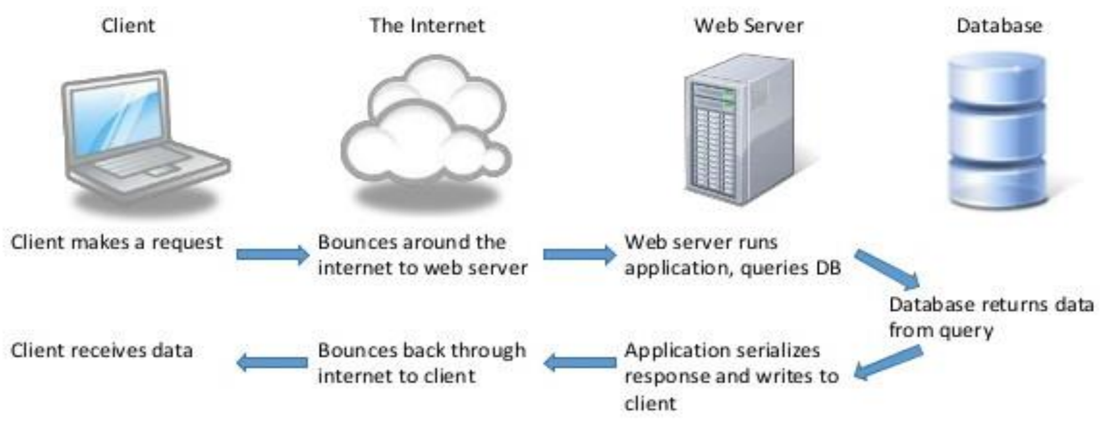


One way to make sure you don't rely on sessions is to use Representational State Transfer (REST) architectures, which are by definition stateless.

If your application can work with REST, sessions are a non-issue. If you have a REST architecture, you can put Ajax on top of REST and have a very useful application. Ajax can maintain information on the client side, which is fine for many applications. Going back to the shopping cart example, you could implement a shopping cart using Ajax. You could work around a lot of the storage that would be required by storing it in the JavaScript app that's running on the client and give it to the server at the end of the interaction.

Some types of information are stateful by nature, such as something that requires an open network connection to monitor progress of a remote operation that does not have stateless features. In such cases, using a stateless design may not be practical, or even possible. When you have more clients than servers it makes sense to do as much work on the clients as you reasonably can. If your application must save state, save it on the client rather than on the server. Consider using a browser cookie, or some Ajax. This will greatly improve your scalability. Furthermore, you pay for the server-side resources you consume, but client-side resources are free!

Principles of REST


Client
The Internet
Web Server
Database
Client makes a request
Bounces around the internet to web server
Web server runs application, queries DB
Database returns data from query
Application serializes response and writes to client
Bounces back through internet to client
Client receives data

There are mainly five REST principles.

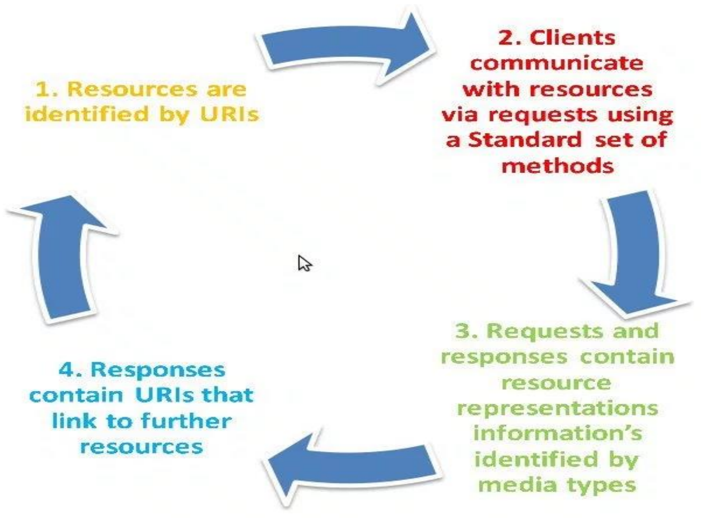
- Give everything an ID
- Standard set of methods
- Link things together
- Multiple representations
- Stateless communications

REST services characteristics

The following list summarizes the key characteristics of the REST architecture:

- Uses a client-server system.
- Stateless.
- Supports caching of resources.
- Proxy servers are supported.
- Uses logical URLs to identify resources.

RESTful Application Cycle


1. Resources are identified by URIs
2. Clients communicate with resources via requests using a Standard set of methods
3. Requests and responses contain resource representations information's identified by media types
4. Responses contain URIs that link to further resources

Conclusion

It's true that web applications should be stateless. However, session variables, cookies, and tokens don't violate this when they are all stored on the client (web browser). They can be parameters in the request.

Here's a simplified model:

Web Browser (has state) <-> Web Server (stateless) <-> Database (has state)

Web applications should be stateless" should be understood as "web applications should be stateless unless there is a very good reason to have state". A "shopping cart" is a stateful feature by design and denying that is quite counter-productive. The whole point of the shopping cart pattern is to preserve the state of the application between requests.

An alternative which I could imagine as a stateless website which implements a shopping cart would be a single-page-application which keeps the shopping cart completely client-sided, retrieves product information with AJAX calls and then sends it to the server all at once when the user does a checkout. But I doubt I have ever seen someone actually do that, because it doesn't allow the user to use multiple browser tabs and doesn't preserve state when they accidentally close the tab. Sure, there are workarounds like using local storage, but then you do have state again, just on the client instead of on the server.

Whenever you have a web application which requires to persist data between pageviews, you usually do that by introducing sessions. The session a request belongs to can be identified by either a cookie or by a URL parameter you add to every link. Cookies should be preferred because they keep your URLs handier and prevent your user from accidentally sharing an URL with their session-id in it. But having URL tokens as a fall back is also vital for users which deactivate cookies. Most web development frameworks have a session handling system which can do this out-of-the-box.

On the server-side, session information is usually stored in a database. Server-side in-memory caching is optional. It can greatly improve response time but won't allow you to transfer sessions between different servers. So, you will need a persistent database as a fall back.