

**Jignesh Kakadiya**

160 Followers · About

Follow



Upgrade



# How does basic HTTP authentication work?



Jignesh Kakadiya Apr 1, 2017 · 6 min read

Did you know? Before **cookie** came to browser, user has to provide **userid** and **password** with every single logged in request? If you knew this you should skip this article and if you don't go ahead and read this article to know how it all works from visiting the web page till user logs out of website.

Here I will try to replicate some of the steps that we perform on the browser for example doing signup, login, logout and try to explain how client and server communicates to keep user logged in and give user logged in page to see (HTML) in all of those steps.

Below are the steps that I will try to go through in detail and try to explain how browser (**client**) and **server** behaves on each step.

**step 1.** Go to your favorite browser. Type a link and press enter.

**step 2.** Click on signup and give your userid/password.

**step 3.** login with userid/password

**step 4.** play around, visit other pages as logged in user

**step 5.** Log out

## 1. Go to your favorite browser. Type a link and press Enter ( ).

### On client side:

Let's say you want to visit [www.medium.com/](https://www.medium.com/). Go ahead and open your favorite browser enter above url and press enter. Once you do that client does lot of stuff like finding the server to talk with, DNS resolutions, 2 way handshakes etc.. Once the connection is set up client sends a request to the server for getting the data corresponding to the path "/" for medium.com.

*You should definitely checkout [what-happens-when](#) repository. Which is about "What happens when you type google.com into your browser's address box and press enter?"*

**On server side:**

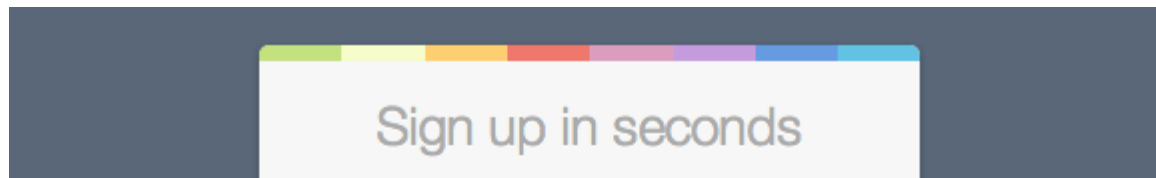
On the other side server gets the request and goes to pull the content for the request, in this case it will fallback to the index file, as “/” is the main file and generally points to index.html (some cases can override this, but this is the most common method). Once server has figured out HTML document it has to send back to the client, server sends response back to the client containing that document which browsers can render on the screen. [More stuff here.](#)

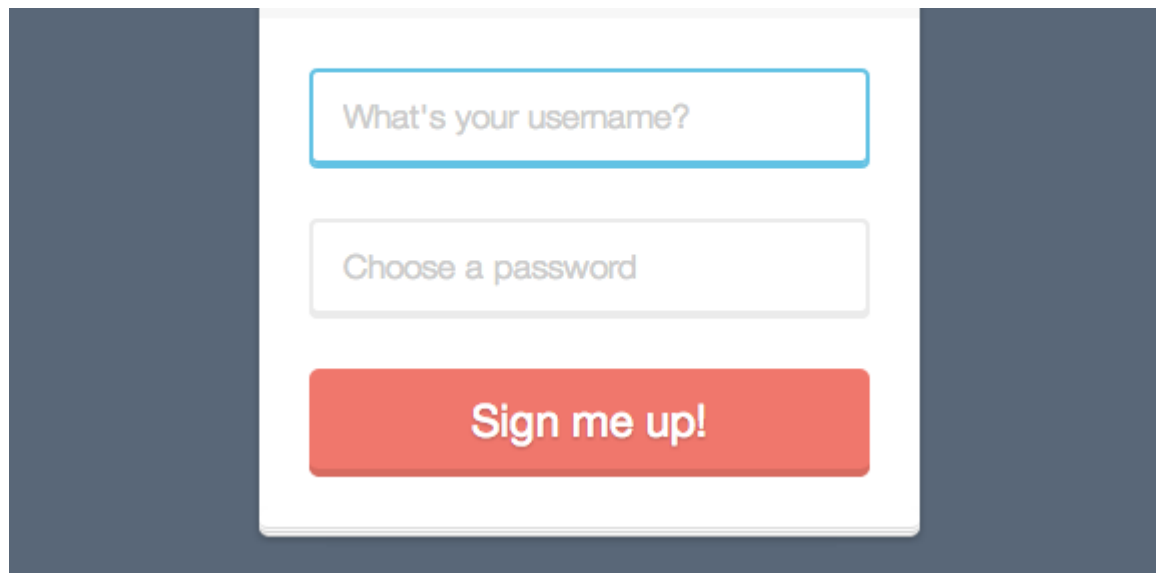
**Requests are stateless:** Means every time you request document from server you have to pass all the information to server in the form of request, Server doesn't know who you are until you tell them.

**For example:** If you want to get your personal page as logged in user, you will have to send your id/password with each request to tell server who you are.

## 2. Click on Signup and give your userId/password.

On client side:



A screenshot of a simple web form for signing up. The form is centered on a dark blue background. It consists of two white input fields with rounded corners. The first field has a blue border and contains the placeholder text "What's your username?". The second field has a grey border and contains the placeholder text "Choose a password". Below these fields is a red button with white text that says "Sign me up!".

Signup form from [here](#)

The simplest signup form has two fields **user id** and **password** to identify user. When you submit the form, browser sends a request containing your id and password to the server. More info on [MDN](#).

The HTTP request looks like this:

-----

```
request: POST / HTTP/1.1
Host: medium.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 30

userid=foo&password=mypassword
```

## On server side:

Server receives the request, extracts the data (credentials) and creates an entry in their **database** with **userid** and **password**. If database entry is created successfully without any userid conflict Server redirects user to **login page**. Otherwise it will throw some error saying userid already exists etc.

*To protect your password from anonymous user accessing the database, Server creates hash of password and stores it against **userid** instead of actual password. Here is an simple examples of how you can generate hashed password with Node.js.*

```
password_hash = hash.create('mypassword', 'sha-1');  
  
// password_hash = 2ef5aa5a037ae1be9c7cdd15649cf9fc686dde2  
// used sha-1 (Secure Hash Algorithm 1) for generating hashed  
password
```

Results	
Original text	mypassword
Original bytes	6d:79:70:61:73:73:77:6f:72:64 (length=10)
Adler32	1816045a
CRC32	cb730b84
Haval	a87bfd9ee9f6676ccce8998902ba88f

MD2	398b7381b88d8697d8c48359152548d0
MD4	4bd44ba402231f35390c1ae3b76f0154
MD5	34819d7beeabb9260a5c854bc85b3e44
RipeMD128	7b694600c8a2a2b0897c719958713619
RipeMD160	0a70cb6f99916d90685ed888922b2bc501838c14
SHA-1	91dfd9ddb4198affc5c194cd8ce6d338fde470e2
SHA-256	89e01536ac207279409d4de1e5253e01f4a1769e696db0d6062ca9b8f56767c8
SHA-384	95b2d3b2ad7c2759bf3daa53424e2a472bc932798dae30b982621833a449492883b7ae9d31d30d32372f98abdbb256ae
SHA-512	a336f671080fb4f2a230f313560ddf0d0c12dfc1741e49e8722a234673037dc493caa8d291d8025f71089d63cea809cc8ae53e5b17054806837dbe4099c4ca
Tiger	8741a3ddd460c10dd8008b73021a25b033bae8f8977f82de
Whirlpool	222d4656bb927915ff26281640041c03f720776532daed15e2a3da1b74cbee7d2e8c8c5bcf4706dc44f4804f2842ff3860029b7a925012742d61f63b349d1bc5

Hashes created with different type of hash algorithms for password = “mypassword”

### 3. You login with credentials (id, password)

#### On client side:

To log in to the system Browser has to send the credentials (**userid**, **password**) via login form (similar form like we used for signup) which points to different route (example: “/login”) which allows us to login. Browser sends request to that route with user’s credentials when you submit (click on submit) that form.

#### On server side:

Server gets the request. Extracts **userid** and **password** from the **request**. Searches for that **userid** in the database, extracts the stored password against that **userid** and tries to compare stored **password** against

**password** received from request (assuming we haven't stored hashed password). If the server stored hashed password in their database it **checks by comparing hash value** of the password. (example: 2ef5aa5a037ae1be9c7cdd15649cf9fc686ddee2).

*If password/hashed password matches then server creates the **token**. Token is nothing but a string that is used to identify user so that user won't have to send id and password with every request in future.*

## 4. Generating token and tell client to use that token for future requests.

On server side:

- Server creates a random token **string** (example: "xyztoken") and puts *it against userid in the database*. You can generate any random tokens by yourself but there are modules which can generate tokens without repeating it.
- Now server has to send this token to the client and tell *client to store this token somewhere and use it for future request to identify user*. Server does this by setting response header **Set-cookie**. Don't forget server also sends the html document along with it.

HTTP response looks like this:

```
=====
HTTP/1.1 200 OK
Set-Cookie: access_token=xyztoken; Path=/; Domain=foo.com;
expires=Thu, 01 Jan 2050 00:00:00 GMT;
-----
<html>
<body>
<h1>Hello User!</h1>
  (more user info)
  .
  .
  .
</body>
</html>
=====
```

**Set-Cookie header tells client to store access\_token in cookie for path "/" and domain "foo.com" which "expires" on 1st day of 2050.**

***Response** contains **headers** and **body** sections. Headers contains set of commands/information that server would like to send to client and body generally contains the html document or JSON depending on the requirements. If you are more curious on how it works go ahead and read [HTTP made really easy](#) by James Marshall.*

### On client side:

Now the client gets the response. Client uses **data (HTML)** to render it on



screen and **value** of **set-cookie** to set as a **cookie**. **Cookie** is nothing but small (key, value) persistent storage which browsers allowed to keep in order to provide stateful behavior. Browsers can store Cookie up to 5MB.

*There is popular chrome extension **EditThisCookie** Which is highly recommended and popular among web developers for cookie management.*

## 5. You play around, visit other pages as logged in user

### On client side:

Now user is logged in. It means client has cookie which contains **access\_token=xyztoken**. When you navigate to other pages on the same domain, browser will send back that cookie to the server by setting it to request header cookie like below.

HTTP request looks like this:

```
GET https://www.foo.com/xyz HTTP/1.1  
Cookie: access_token=xyztoken;
```

Client sends back that cookie to the server to identify current userid.

### On Server side:

Server receives a request. extracts **access\_token** from cookie and searches for that token in database to see which **user id** it points to. Once we have **user id** its easy to get all the information about the user and create specific HTML document for that user. If token doesn't match server will redirect client to the login page or show errors indicating password doesn't match. Remember while sending data back to the client, server doesn't have to send the Set-Cookie as a header again and again because client already have that cookie stored in a persistent storage.

## 5. You log out

### On client side:

When click on the logout. There is a separate route for logout (example: /logout). Browser sends request to the server on that route with existing token set as a cookie.

### On server side:

Server extracts the token, Finds the **userid** corresponding to that token, deletes the access token against that **userid** and redirects user to login page. Now that we have removed the token from database the server will have to tell **client** to remove that token from cookie because that token doesn't exist anymore. To do that while redirecting user to login page server

uses **Set-Cookie** header again, but sets **access\_token** as **empty string** to tell browser to remove token from the cookie.

```
Set-Cookie: token=''; path=/; expires=Thu, 01 Jan 1970 00:00:00 GMT
```

*There is a flag **HTTPOOnly** cookie used to not allow browsers to access cookie via JavaScript to prevent XSS (cross site scripting) attacks. More information on flags here [https://en.wikipedia.org/wiki/HTTP\\_cookie#Terminology](https://en.wikipedia.org/wiki/HTTP_cookie#Terminology)*

I hope it helps. This is just the basic cookie/session management. It doesn't cover lots of loopholes this approach has. But now if you go ahead and read those documents or codes it will be more clear to understand how sessions and attack prevention works.

*If there's anything I can help you with, don't hesitate to [hit me up on Twitter!](#)*

[JavaScript](#)[Web Development](#)[Http](#)[Nodejs](#)



[About](#)

[Help](#)

[Legal](#)