Spring Security Basics

Hash Functions

They provide a mapping between an arbitrary length input, and a (usually) fixed length (or smaller length) output. It can be anything from a simple crc32, to a full blown cryptographic hash function such as MD5 or SHA1/2/256/512. The point is that **there's a one-way mapping going on.** It's **always a many:1 mapping** (meaning there will always be collisions) since every function produces a smaller output than it's capable of inputting (If you feed every possible 1mb file into MD5, you'll get a ton of collisions). They are **hard** (or **impossible in practicality**) to **reverse**.

Encryption Functions

They provide a **1:1 mapping** between an arbitrary length input and output. And **they are always reversible**. The important thing to note is that it's reversible using some method. And it's always 1:1 for a given key. Now, there are multiple input:key pairs that might generate the same output (in fact there usually are, depending on the encryption function). Good encrypted data is indistinguishable from random noise. This is different from a good hash output which is always of a consistent format.

Use Cases

Use a **hash function** when you want to compare a value but can't store the plain representation (for any number of reasons). Passwords should fit this use-case very well since you don't want to store them plain-text for security reasons (and shouldn't). But what if you wanted to check a filesystem for pirated music files? It would be impractical to store 3 mb per music file. So instead, take the hash of the file, and store that (md5 would store 16 bytes instead of 3mb). That way, you just hash each file and compare to the stored database of hashes (This doesn't work as well in practice because of re-encoding, changing file headers, etc, but it's an example use-case).

Use a **hash function** when you're checking validity of input data. That's what they are designed for. If you have 2 pieces of input, and want to check to see if they are the same, run both through a hash function. The probability of a collision is astronomically low for small input sizes (assuming a good hash function). That's why it's recommended for passwords. For passwords up to 32 characters, md5 has 4 times the output space. SHA1 has 6 times the output space (approximately). SHA512 has about 16 times the output space. You don't really care what the password was, you care if it's the same as the one that was stored. That's why you should use hashes for passwords.

Hash functions are also great for signing data. For example, if you're using HMAC, you sign a piece of data by taking a hash of the data concatenated with a known but not transmitted value (a secret value). So, you send the plain-text and the HMAC hash. Then, the receiver simply hashes the submitted data with the known value and checks to see if it matches the transmitted HMAC. If it's the same, you know it wasn't tampered with by a party without the secret value. This is commonly used in secure cookie systems by HTTP frameworks, as well as in message transmission of data over HTTP where you want some assurance of integrity in the data.

Hashing is useful if you want to send someone a file. But you are afraid that someone else might intercept the file and change it. So a way that the recipient can make sure that it is the right file is if you post the hash value publicly. That way the recipient can compute the hash value of the file received and check that it matches the hash value.

Encryption is good if you say have a message to send to someone. You encrypt the message with a key and the recipient decrypts with the same (or maybe even a different) key to get back the original message. Credits

Use **encryption** whenever you need to get the input data back out. Notice the word need. If you're storing credit card numbers, you need to get them back out at some point, but don't want to store them plain text. So instead, store the encrypted version and keep the key as safe as possible.

A note on hashes for passwords:

A key feature of cryptographic hash functions is that they should be very fast to create, and very difficult/slow to reverse (so much so that it's practically impossible). This poses a problem with passwords. If you store sha512(password), you're not doing a thing to guard against **rainbow tables** or **brute force attacks**. Remember, the hash function was designed for speed. So it's trivial for an attacker to just run a dictionary through the hash function and test each result.

Adding a salt helps matters since it adds a bit of unknown data to the hash. So instead of finding anything that matches **`md5(foo)`**, they need to find something that when added to the known salt produces **`md5(foo + salt)`** (which is very much harder to do). But it still doesn't solve the speed problem since if they know the salt it's just a matter of running the dictionary through.

So, there are ways of dealing with this. One popular method is called **key strengthening** (or **key stretching**). Basically, you iterate over a hash many times (thousands usually). This does two things. First, it slows down the runtime of the hashing algorithm significantly. Second, if implemented right (passing the input and salt back in on each iteration) actually increases the entropy (available space) for the output, reducing the chances of collisions. A trivial implementation is:

```
var hash = password + salt;
for (var i = 0; i < 5000; i++) {
    hash = sha512(hash + password + salt);
}
```

- The bottom line, hash(password) is not good enough. hash(password + salt) is better, but still not good enough... Use a stretched hash mechanism to produce your password hashes...

Another note on trivial stretching

Do not under any circumstances feed the output of one hash directly back into the hash function:

```
hash = sha512(password + salt);
for (i = 0; i < 1000; i++) {
    hash = sha512(hash); // <-- Do NOT do this!
}
```

Basic access authentication (Http Basic Auth)

In the context of an HTTP transaction, basic access authentication is a method for an HTTP user agent (e.g. a web browser) to provide a **user name** and **password** when making a request. In basic HTTP authentication, a request contains a **header field** of the form `Authorization: Basic <credentials>`, where credentials is the **base64** encoding of id and password joined by a single colon (:).

HTTP Basic authentication (BA) implementation is the simplest technique for enforcing access controls to web resources because **it does not require** cookies, session identifiers, or login pages; rather, HTTP Basic authentication uses standard fields in the HTTP header, removing the need for handshakes.

The BA mechanism provides **no confidentiality protection** for the transmitted credentials. They are merely encoded with Base64 in transit, but not encrypted or hashed in any way. Therefore, Basic Authentication is typically used in conjunction with HTTPS to provide confidentiality.

- Because the BA field has to be sent in the header of each HTTP request, the web browser needs to cache credentials for a reasonable period of time to avoid constantly prompting the user for their username and password. **Caching policy differs between browsers**. Microsoft Internet Explorer caches the credentials for 15 minutes by default.

HTTP does not provide a method for a web server to instruct the client to "**log out**" the user. However, there are a number of methods to clear cached credentials in certain web browsers. One of them is redirecting the user to a URL on the same domain containing credentials that are intentionally incorrect. However, this behavior is inconsistent between various browsers and browser versions. Microsoft Internet Explorer offers a dedicated JavaScript method to clear cached credentials:

```
<script>document.execCommand('ClearAuthenticationCache');</script>
```

Server side

When the server wants the user agent to authenticate itself towards the server, the server must respond appropriately to unauthenticated requests. To unauthenticated requests, the server should return

- a response whose header contains a **HTTP 401 Unauthorized** status and
- a **WWW-Authenticate** field: The WWW-Authenticate field for basic authentication is constructed as following: `WWW-Authenticate: Basic realm="User Visible Realm"` The server may choose to include the charset parameter from RFC 7617: `WWW-Authenticate: Basic realm="User Visible Realm", charset="UTF-8"` This parameter indicates that the server expects the client to use UTF-8 for encoding username and password (see below).

Client side

When the user agent wants to send authentication credentials to the server, it may use the `Authorization` field. The Authorization field is constructed as follows:

1. The **username** and **password** are combined with a **single colon** (:). This means that the username itself cannot contain a colon.
2. The resulting string is encoded into an octet sequence. The character set to use for this encoding is by default unspecified, as long as it is compatible with US-ASCII, but the server may suggest use of UTF-8 by sending the charset parameter.
3. The resulting string is encoded using a variant of **Base64**.
4. The authorization method and a space (e.g. "Basic ") is then prepended to the encoded string.

For example, if the browser uses Aladdin as the username and OpenSesame as the password, then the field's value is the base64-encoding of Aladdin:OpenSesame, or QWxhZGRpbjpPcGVuU2VzYW1l. Then the Authorization header will appear as:

```
Authorization: Basic QWxhZGRpbjpPcGVuU2VzYW1l
```

URL encoding

A client may avoid a login prompt when accessing a basic access authentication by prepending **username:password@** to the hostname in the URL. For example, the following would access the page index.html at the web site www.example.com with the secure HTTPS protocol and provide the username Aladdin and the password OpenSesame credentials via basic authorization:

```
https://Aladdin:OpenSesame@www.example.com/index.html
```

This has been deprecated by RFC 3986: Use of the format "user:password" in the user info field is deprecated. **Some modern browsers therefore no longer support URL encoding of basic access credentials**. This prevents passwords from being sent and seen prominently in plain text, and also eliminates (potentially deliberately) confusing URLs.

Created a basic spring app with some tymeleaf templates.

Add Spring Boot Dependencies

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>

<dependency>
  <groupId>org.thymeleaf.extras</groupId>
  <artifactId>thymeleaf-extras-springsecurity5</artifactId>
</dependency>
```

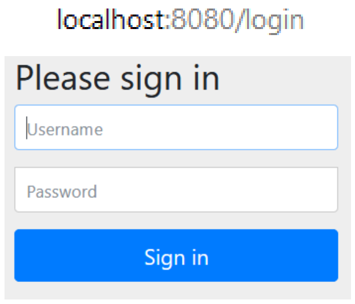
The main dependency we need. Even just by adding this to our pom, some default security features will be enabled.

Second dependency is not mandatory but it is enabling some security features in thymeleaf. For example showing the authenticated user, or if the user is auth, you want to display a certain template if the user has the correct role / authority.

Default Spring Boot Security Configuration

We now have a line printed:

```
Using generated security password: 76c4f29f-a2f0-4add-b5e4-2d9431a4ad56
```


localhost:8080/login

Please sign in

Username

Password

Sign in

This was not there before. So now when I try to access the homepage of the app, it redirects me to `http://localhost:8080/login` and a login page shows up. This page was generated by the dependency, I did not create this.

We can login by typing `user` (spring default username) and give the generated password.

So this is a basic auth mechanism. You can change the configuration from `application.properties` file. Now when I refresh I can login with the updated user credentials.

```
spring.security.user.name=can
spring.security.user.password=can
spring.security.user.roles=...
```

HTTP Basic Authentication using Spring Security

Each time you need a custom security implementation you need to create a class like this.

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
```

2 users: admin, can and we will configure http basic auth.

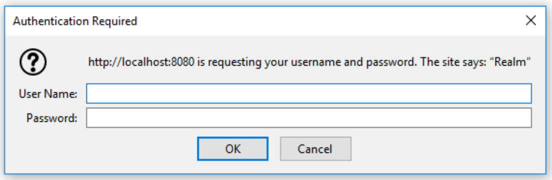
Usually the structure is the same no matter what kind of security you will use.

1. You need to define a data source for your users.
2. Then you need to authorize requests.
3. Provide password encoder

```
// need to still protect our resources
@Override
protected void configure(HttpSecurity http) throws Exception {
  // require authentication for any resource: all views or api
  // and I want to use http basic
  http.authorizeRequests().anyRequest().authenticated()
      .and()
      .httpBasic();
}
```

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

  @Override
  protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    // user credentials will be stored in memory
    // each time our application has started
    // later we will use a database
    auth.inMemoryAuthentication()
        .withUser("admin").password("admin123").roles("ADMIN")
        .and()
        .withUser("can").password("can123").roles("USER");
  }
```

When I try to login this browser will pop-up:


Authentication Required

http://localhost:8080 is requesting your username and password. The site says: "Realm"

User Name:
Password:

When we try to login with the 2 users we created we will get an exception and won't be able to login.

```
java.lang.IllegalArgumentException: There is no PasswordEncoder mapped for the id "null"
```

Password encoder will hash your passwords when they will be stored. They won't be stored in plain text. Doesn't matter if we use an in memory auth or a database its always to good idea to use it.

- In previous versions this was not mandatory.

```
@Bean
public PasswordEncoder passwordEncoder() {
  return new BCryptPasswordEncoder();
}
```

We also need to encode our plain text password string with this encoder when creating users.

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
  // user credentials will be stored in memory each time our application has started
  // later we will use a database
  auth.inMemoryAuthentication()
      .withUser("admin").password(passwordEncoder().encode("admin123")).roles("ADMIN")
      .and()
      .withUser("can").password(passwordEncoder().encode("can123")).roles("USER");
```

Now login works! Both users and role types can login. We will also see how we can refine that.
Provide a finer level of authorization, let ADMIN only access some resources etc.

Configure Role Based Authorization

The homepage is open for access to everyone.

- `/profile` -> Every authenticated user.
- `/admin` -> only ADMIN role can access
- `/management` -> ADMIN and MANAGEMENT role can access it.

Atm this tells us that any role can access any resource if authenticated. We want to change that and implement role based authentication.

```
http.authorizeRequests().anyRequest().authenticated()
```

- Generally we will choose to secure routes or folders, not individual pages.

These won't work!

```
.antMatchers("/profile").aut
.antMatchers("/admin").hasRo
.antMatchers("/management").
```

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http.authorizeRequests()
      .antMatchers("/").permitAll() // no auth needed to access
      .antMatchers("/profile/").authenticated()
      .antMatchers("/admin/").hasRole("ADMIN")
      .antMatchers("/management/").hasAnyRole("ADMIN", "MANAGER")
      .and()
      .httpBasic();
```

So right now the homepage is open for everyone. `/profile` is asking for username and pass. After login as a USER, you can only access `/profile`. If you try to access other pages you will get a **403 Forbidden error**.

```
Tue Jul 16 23:19:03 CEST 2019
There was an unexpected error (type=Forbidden, status=403).
Forbidden
```

Protecting views and resources are basically the same:

```
.antMatchers("/api/**").authenticated()
.antMatchers("/api/**").hasAnyRole("ADMIN", "MANAGER")
```

- ➢ Strings passed to antMatchers is important! If you write "/profile" then only the resource /profile will be protected and you can still access "/profile/" without auth.
- ➢ If you want to cover both '/profile' and '/profile/' and the sub-resources, you can use wildcards. What is the best practice for this in general?

We can also use a shortcut to give certain roles **access to all sub-resources under a path**:

```
http.authorizeRequests()
    .antMatchers("/").permitAll() // no auth needed to access
    .antMatchers("/profile/**").authenticated()
    .antMatchers("/admin/**").hasRole("ADMIN")
    .antMatchers("/management/**").hasAnyRole("ADMIN", "MANAGER")
    .and()
    .httpBasic();
```

- ➢ The order that you put these `.antMatchers()` is really important. With each request, these are traversed from top to bottom as seen above, so the first match will be picked and processed. So if we write `.anyRequest().permitAll()` at the top, then each request will be allowed and others will be ignored.

  You don't want to give more / less access than you normally want. Be careful!

This will give a 404! localhost:8080/profile

This works! localhost:8080/profile/

```
@Controller
@RequestMapping("/profile")
public class ProfileController {

  //  @GetMapping("/") -> problem
  @GetMapping("")
  public String index() {
    return "profile/index";
  }
```

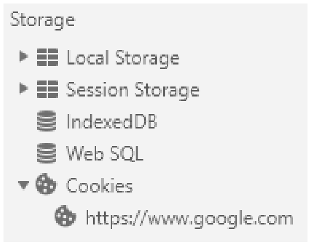
The reason was the string in my controllers. When you give an empty string inside @GetMapping then both will work.

If we login with an admin account then we can access every page.

- ➢ I can’t logout. No cache, no cookies but I’m still logged in always, how?
- ➢ Short answer: when I login, my credentials are saved somewhere on my file system and it is appended to every request as a request header. I also get a session id back from the server and that alone can also be used to login. So deleting one of them isn’t enough. Somehow both should be deleted for a complete logout. Or the browser window must completely be closed.

Chrome and Firefox:

- Keeps session cookies until the window is closed, not the tab! Meaning that if you login to 8080, you will be logged in till you close the whole window.
- In **Developer Console –> Network**, you can only see the cookies that are related to the domain of the opened tab!
- To see all cookies for every domain: (Chrome) **Settings –> Advanced Settings –> Site Settings –> Cookies –> See all cookies and site data**.



Where is basic auth credentials stored in the browser?

They are sent as an 'Authorization' header on each request (with base65 encoded value). But where are these values stored on the browser? They don't appear to be in cookies / sessions / storage etc.

- They are stored somewhere on your file system, under

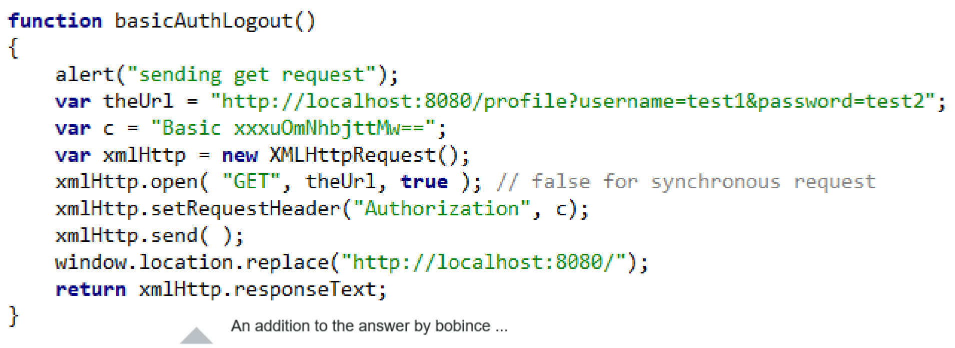
```
C:\Users\<username>\Appdata\Local\Google\Chrome\User Data\Default\….
```

Cookie and Basic Auth Credential Lifecycle

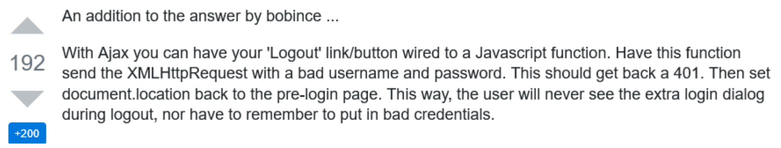
- Basic Auth credentials are cached until the **browser is closed** (Chrome and Firefox). The problem with logging out from Basic Auth is simple: the browser will cache your credentials by default until the browser windows is closed. There is no standard mechanism to invalidate them. So Basic Auth doesn't allow a log-out!
- A session cookie is also deleted usually when the **browser is closed**. If a cookie has neither the **Max-Age** nor the **Expires** attribute, the user agent will retain the cookie until "the current session is over" (as defined by the user agent). The user agent is not required to retain the cookie for the specified duration. In fact, user agents often evict cookies due to memory pressure or privacy concerns.

So in order to logout from my spring application, I have to tell my app that I no longer want to get cookies, meaning that it should log me out and also to somehow remove the basic auth credentials when clicked on the logout button.

I first deleted the cookie then clicked the button which invokes this method (as stackoverflow suggested), but it didn't work. I was still logged in.



```
function basicAuthLogout()
{
    alert("sending get request");
    var theUrl = "http://localhost:8080/profile?username=test1&password=test2";
    var c = "Basic xxxuOmNhbjttMw==";
    var xmlHttp = new XMLHttpRequest();
    xmlHttp.open( "GET", theUrl, true ); // false for synchronous request
    xmlHttp.setRequestHeader("Authorization", c);
    xmlHttp.send( );
    window.location.replace("http://localhost:8080/");
    return xmlHttp.responseText;
}
```



192

+200

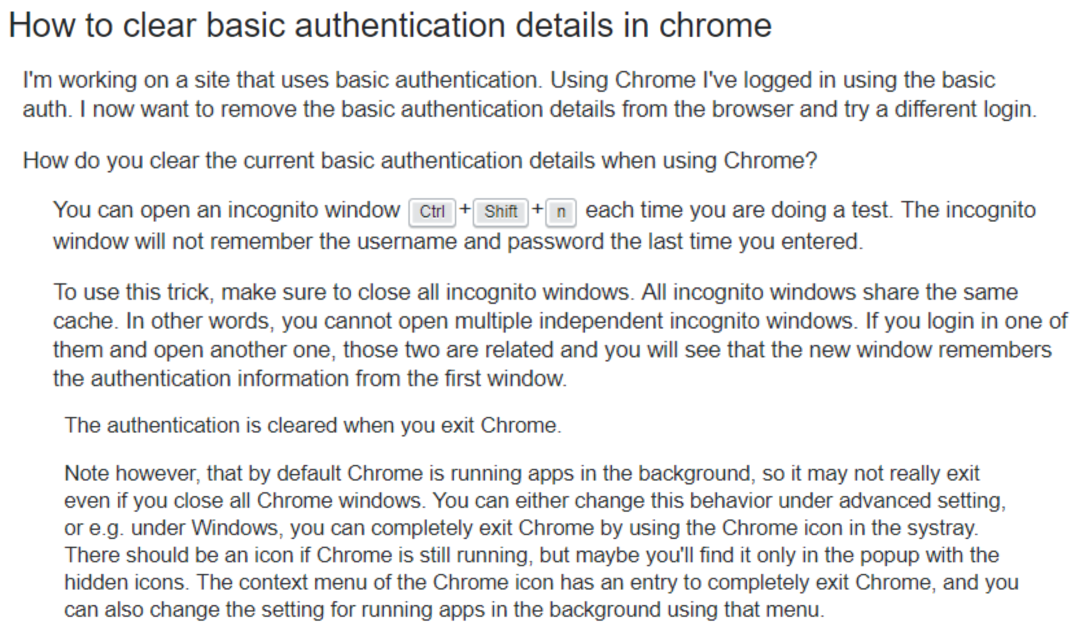
An addition to the answer by bobince ...

With Ajax you can have your 'Logout' link/button wired to a Javascript function. Have this function send the XMLHttpRequest with a bad username and password. This should get back a 401. Then set document.location back to the pre-login page. This way, the user will never see the extra login dialog during logout, nor have to remember to put in bad credentials.

https://log:out@example.com/.

Chrome and Firefox do not always clear session cookies on exit

Cookies without an expiration timestamp are called "**session cookies**". They should only be kept until the end of the browsing session. However, when Chrome or Firefox are configured to reopen tabs from last time upon start, they will keep session cookies when closing the browser. This even applies to tabs that were closed before shutting down the browser.



How to clear basic authentication details in chrome

I'm working on a site that uses basic authentication. Using Chrome I've logged in using the basic auth. I now want to remove the basic authentication details from the browser and try a different login.

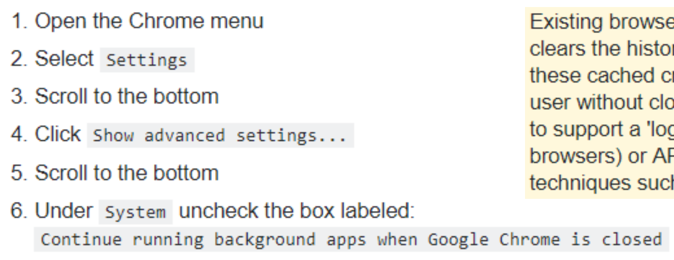
How do you clear the current basic authentication details when using Chrome?

You can open an incognito window Ctrl + Shift + n each time you are doing a test. The incognito window will not remember the username and password the last time you entered.

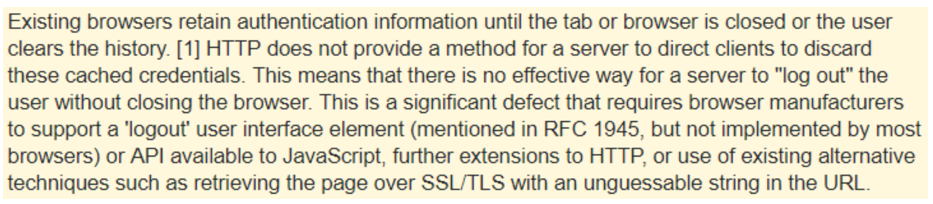
To use this trick, make sure to close all incognito windows. All incognito windows share the same cache. In other words, you cannot open multiple independent incognito windows. If you login in one of them and open another one, those two are related and you will see that the new window remembers the authentication information from the first window.

The authentication is cleared when you exit Chrome.

Note however, that by default Chrome is running apps in the background, so it may not really exit even if you close all Chrome windows. You can either change this behavior under advanced setting, or e.g. under Windows, you can completely exit Chrome by using the Chrome icon in the systray. There should be an icon if Chrome is still running, but maybe you'll find it only in the popup with the hidden icons. The context menu of the Chrome icon has an entry to completely exit Chrome, and you can also change the setting for running apps in the background using that menu.



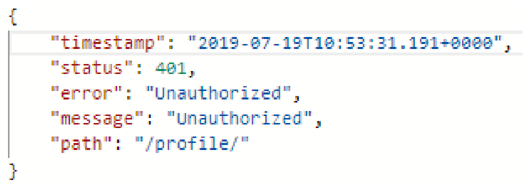
1. Open the Chrome menu
2. Select `Settings`
3. Scroll to the bottom
4. Click `Show advanced settings...`
5. Scroll to the bottom
6. Under `System` uncheck the box labeled:
   `Continue running background apps when Google Chrome is closed`



Existing browsers retain authentication information until the tab or browser is closed or the user clears the history. [1] HTTP does not provide a method for a server to direct clients to discard these cached credentials. This means that there is no effective way for a server to "log out" the user without closing the browser. This is a significant defect that requires browser manufacturers to support a 'logout' user interface element (mentioned in RFC 1945, but not implemented by most browsers) or API available to JavaScript, further extensions to HTTP, or use of existing alternative techniques such as retrieving the page over SSL/TLS with an unguessable string in the URL.

Testing with Postman

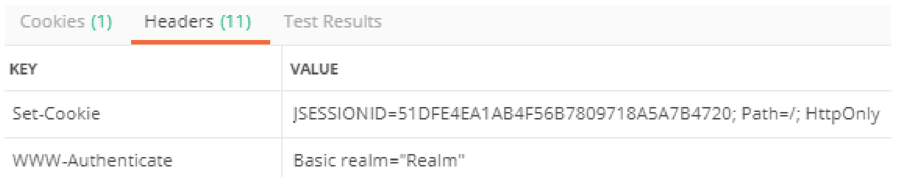
Sending a GET request with postman (without any request headers) to /profile gives me as expected a 401 response.



```
{
    "timestamp": "2019-07-19T10:53:31.191+0000",
    "status": 401,
    "error": "Unauthorized",
    "message": "Unauthorized",
    "path": "/profile/"
}
```

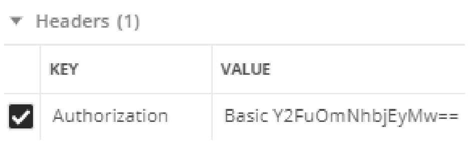
We can see the response headers here:

Status: 401 Unauthorized



KEY	VALUE
Set-Cookie	JSESSIONID=51DFE4EA1AB4F56B7809718A5A7B4720; Path=/; HttpOnly
WWW-Authenticate	Basic realm="Realm"

Now also sending the required credentials inside an '**Authorization**' header. I copied the base64 string from the browser where I was already logged in.

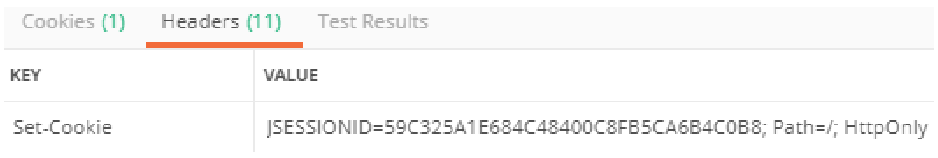


	KEY	VALUE
✓	Authorization	Basic Y2FuOmNhbjEyMw==

Notice that I only send one header, not any cookies (JSESSIONID)

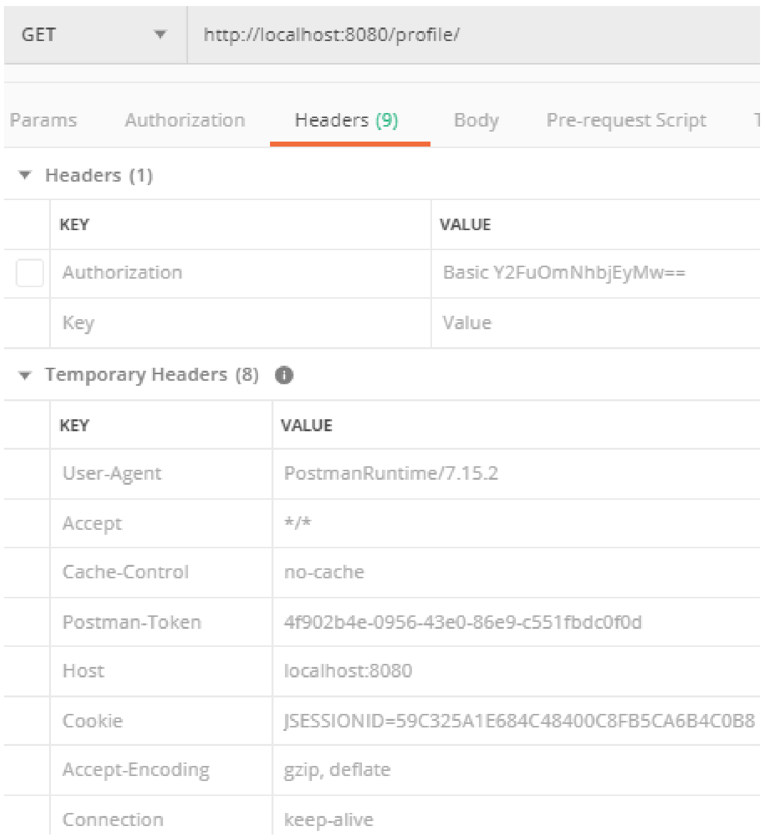
- This worked! Now I got a 200 response

Status: 200 OK



KEY	VALUE
Set-Cookie	JSESSIONID=59C325A1E684C48400C8FB5CA6B4C0B8; Path=/; HttpOnly

IMPORTANT: When I got the response, postman saved the JsessionId cookie under 'Temporary Headers'.



GET http://localhost:8080/profile/

Headers (1)

	KEY	VALUE
☐	Authorization	Basic Y2FuOmNhbjEyMw==
	Key	Value

Temporary Headers (8)

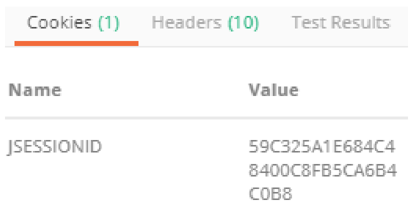
KEY	VALUE
User-Agent	PostmanRuntime/7.15.2
Accept	*/*
Cache-Control	no-cache
Postman-Token	4f902b4e-0956-43e0-86e9-c551fbdc0f0d
Host	localhost:8080
Cookie	JSESSIONID=59C325A1E684C48400C8FB5CA6B4C0B8
Accept-Encoding	gzip, deflate
Connection	keep-alive

Now I make another request to /profile but this time **without** 'Authorization' and **with** JsessionId cookie!

This gives me a 200 response back, meaning that it sees me still as logged in. So I didn't have to send the credentials again since now I have a session on the server side. Sending the session cookie is enough to authenticate yourself.

Status: 200 OK

Notice that the server sends the same cookie back. Meaning the same session will be used to authentication as expected



Name	Value
JSESSIONID	59C325A1E684C48400C8FB5CA6B4C0B8

What is JSESSIONID

JSESSIONID is a **cookie** generated by Servlet containers like Tomcat or Jetty and used for session management in J2EE web application for HTTP protocol. Since HTTP is a stateless protocol there is no way for Web Server to relate two separate requests coming from the same client and Session management is the process to track user session using different session management techniques like **Cookies** and **URL Rewriting**.

If a Web server is using a cookie for session management it creates and sends JSESSIONID cookie to the client and then the client sends it back to the server in subsequent HTTP requests.

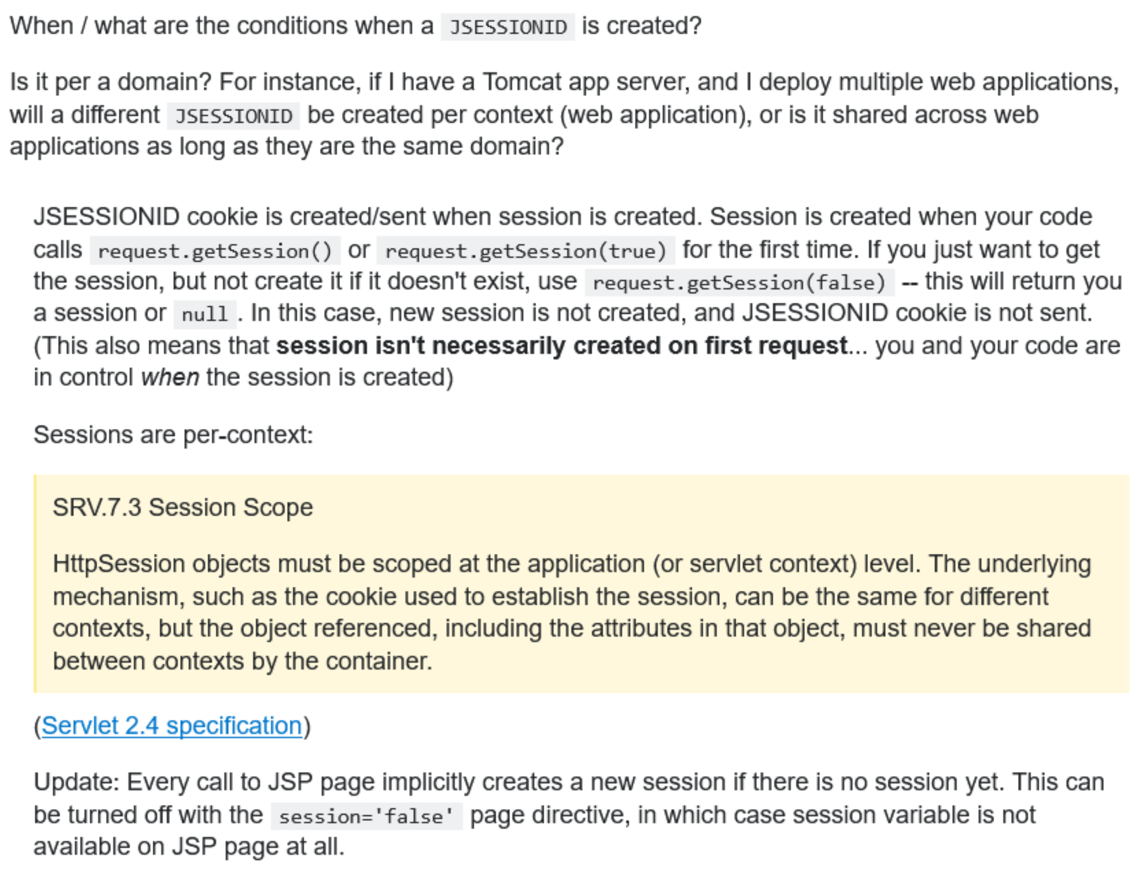
When is JSESSIONID created in a Web application?

In Java, a J2EE application container is responsible for Session management and by default uses cookies. When a user first time access your web application, session is created based on whether its accessing HTML, JSP or Servlet.

If user request is served by **Servlet** than session is created by calling `request.getSession(true)` method. it accepts a boolean parameter which instruct to create session if its not already exists. If you call request.getSession(false) then it will either return null if no session is associated with this user or return the associated HttpSession object. If HttpRequest is for a JSP page then the container automatically creates a new Session with JSESSIONID if this feature is not disabled explicitly by using page directive %@ page session="false" %>.

Once Session is created the container sends JSESSIONID cookie into response to the client. In case of HTML access, no user session is created. If client has disabled cookies, then the Container uses URL rewriting for managing session on which jsessionid is appended into URL as shown below: https://localhost:8443/supermart/login.htm;jsessionid=1A530637289A03B07199A44E8D531427

When HTTP session is invalidated(), mostly when the user logged off, old JSESSIONID destroyed and a new JSESSIONID is created when the user further login.



When / what are the conditions when a `JSESSIONID` is created?

Is it per a domain? For instance, if I have a Tomcat app server, and I deploy multiple web applications, will a different `JSESSIONID` be created per context (web application), or is it shared across web applications as long as they are the same domain?

JSESSIONID cookie is created/sent when session is created. Session is created when your code calls `request.getSession()` or `request.getSession(true)` for the first time. If you just want to get the session, but not create it if it doesn't exist, use `request.getSession(false)` -- this will return you a session or `null`. In this case, new session is not created, and JSESSIONID cookie is not sent. (This also means that **session isn't necessarily created on first request**... you and your code are in control *when* the session is created)

Sessions are per-context:

> SRV.7.3 Session Scope
>
> HttpSession objects must be scoped at the application (or servlet context) level. The underlying mechanism, such as the cookie used to establish the session, can be the same for different contexts, but the object referenced, including the attributes in that object, must never be shared between contexts by the container.

(Servlet 2.4 specification)

Update: Every call to JSP page implicitly creates a new session if there is no session yet. This can be turned off with the `session='false'` page directive, in which case session variable is not available on JSP page at all.

Configure Permission Based Authorization

In Spring Security, granted authorities and roles are a form of expressing a privilege / permission for an authenticated user. Both are expressed with plain names.

Granted authorities: PERMISSION_PROFILE_READ, PERMISSION_PROFILE_EDIT, PERMISSION_PROFILE_DELETE

Roles: ROLE_ADMIN, ROLE_USER, ROLE_SALES, ROLE_MANAGEMENT.

- In Spring Security we can name permissions anything we want, but for roles **we have to prefix the name** with '**ROLE**' prefix.

The only difference between Roles and Permissions are their granularity level.

Granularity
Granted Authorities / Permissions
Roles

Roles are **big chunks** used for authorization: MANAGER_ROLE, ADMIN_ROLE

Permissions are more specific and granular: read users, modify users, access api's, modify api's etc.

In a real world app, only a role based authorization system might not be flexible enough. Lots of features and if we need to allow only certain ones for users then it might be difficult with only roles. Different users might have the same role but different privileges. In those scenarios it's more common to define permission, instead of creating new roles.

For demo, we have rest api endpoints with different access restrictions

- .authorities(String... authorities) is used to define authorities.
- Roles and GrantedAuthorities are basically the same thing.
- Roles() is just an authority with the prefix "ROLE_" appended.

```java
@RestController
@RequestMapping("/api")
public class PublicRestApiController {

  @GetMapping("/test1")
  public String test1() {
    return "api test 1";
  }

  @GetMapping("/test2")
  public String test2() {
    return "api test 2";
```

```java
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
  // user credentials will be stored in memory each time our application has started
  // later we will use a database
  auth.inMemoryAuthentication()
      .withUser("admin")
      .password(passwordEncoder().encode("admin123"))
      .roles("ADMIN")
      .authorities("ACCESS_TEST1", "ACCESS_TEST2")
      .and()
      .withUser("can")
      .password(passwordEncoder().encode("can123"))
      .roles("USER")
      .and()
      .withUser("manager")
      .password(passwordEncoder().encode("manager123"))
      .roles("MANAGER")
      .authorities("ACCESS_TEST1");
```

```java
@Override
protected void configure(HttpSecurity http) throws Exception {
  http.authorizeRequests()
      .antMatchers("/").permitAll() // no auth needed to access
      .antMatchers("/profile/**").authenticated()
      .antMatchers("/admin/**").hasRole("ADMIN")
      .antMatchers("/management/**").hasAnyRole("ADMIN", "MANAGER")
      .antMatchers("/api/test1").hasAuthority("ACCESS_TEST1")
      .antMatchers("/api/test2").hasAuthority("ACCESS_TEST2")
      .and()
      .httpBasic();
}
```

If we try to implement it like this, thinking of roles and permissions (authorities) as different things, **it will fail**!

Every **.roles()** or **.authorities()** will create a new list of authorities! This means if we chain it like this, the first invocation of each method will be ignored. In this example **.roles("ADMIN")** and **.roles("MANAGER")** are completely ignored because they are overwritten by **.authorities(…)**. So at the end, admin and manager users will only have test1, test2 authorities.

```java
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
  auth.inMemoryAuthentication()
      .withUser("admin")
      .password(passwordEncoder().encode("admin123"))
      .authorities("ACCESS_TEST1", "ACCESS_TEST2", "ROLE_ADMIN")
      .and()
      .withUser("can")
      .password(passwordEncoder().encode("can123"))
      .roles("USER")
      .and()
      .withUser("manager")
      .password(passwordEncoder().encode("manager123"))
      .authorities("ACCESS_TEST1", "ROLE_MANAGER");
}
```

One solution is to write the roles, with the "ROLE_" prefix as authorities for each user.

Maybe there is a better way, need to check internet.

HTTPS/SSL

Hypertext Transfer Protocol Secure (HTTPS) is an extension of the Hypertext Transfer Protocol (HTTP). It is used for secure communication over a computer network, and is widely used on the Internet. In HTTPS, the communication protocol is encrypted using **Transport Layer Security** (TLS), or, formerly, its predecessor, **Secure Sockets Layer** (SSL). The protocol is therefore also often referred to as HTTP over TLS, or HTTP over SSL. Today, more than 50% of all websites are HTTPS.

The principal motivation for HTTPS is authentication of the accessed website and protection of the privacy and integrity of the exchanged data while in transit. It protects against man-in-the-middle attacks. The bidirectional encryption of communications between a client and server protects against eavesdropping and tampering of the communication. In practice, this provides a reasonable assurance that one is communicating without interference by attackers with the website that one intended to communicate with, as opposed to an impostor.

HTTPS creates a secure channel over an insecure network. This ensures reasonable protection from eavesdroppers and man-in-the-middle attacks, provided that adequate cipher suites are used and that the server certificate is verified and trusted.

Because HTTPS piggybacks HTTP entirely on top of TLS, the entirety of the underlying HTTP protocol can be encrypted. This includes the request URL (which particular web page was requested), query parameters, headers, and cookies (which often contain identity information about the user). However, because host (website) addresses and port numbers are necessarily part of the underlying TCP/IP protocols, HTTPS cannot protect their disclosure. In practice this means that even on a correctly configured web server, eavesdroppers can infer the IP address and port number of the web server (sometimes even the domain name e.g. www.example.org, but not the rest of the URL) that one is communicating with, as well as the amount (data transferred) and duration (length of session) of the communication, though not the content of the communication.

Web browsers know how to trust HTTPS websites based on certificate authorities that come pre-installed in their software. Certificate authorities (such as Let's Encrypt, Digicert, Comodo, GoDaddy and GlobalSign) are in this way being trusted by web browser creators to provide valid certificates.

Why SSL exists

- **Encryption**: Higing what is sent from one computer to another
- **Identification**: Making sure the computer you are speaking to is the one you trust.

Encryption

1. **Computers agree on how to encrypted**: Sends a Hello message to the other side, containing information about the key exchange method (eg RSA, Diffie-Hellman, DSA), Cipher (RC4, Triple DES, AES), Hash (HMAC-MD5, HMAC-SHA), version of SSL (3.3 inticating TLS), a random number which is used to compute the master secret which is from then on used to calculate the encryption keys. The client sends 'this is what I can do' and the server agrees / chooses.
2. **Server sends certificate**: Then the server send a certificate to the client which contains information like who the server belongs to, how long cert valid, public key, …
3. **Your computer says 'start encrypting'**: Both computers calculate a master secret code which is used to encrypt all the communication. Client Key Exchange, Change Cipher Spec, Finished
4. **The server says 'start encrypting'**
5. **All messages are now encrypted**

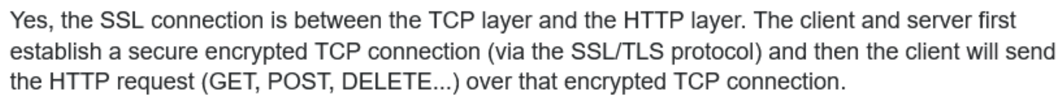
Identification

You can have an SSL certificate but that doesn't automatically means that the other side is not faking their identity.

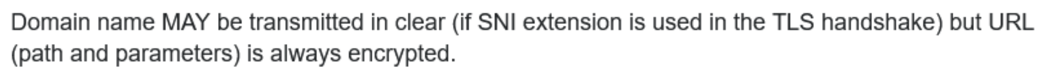
1. **Company asks CA for a certificates**
2. **CA creates certificate and signs it**: Signature created by condensing all details of the certificate into a number through hashing and then the CA encrypts that number with its private key, so anyone holding the public key of the CA can verify its correct (?). Cert also contains an algorithm and key used for the actual encryption of messages.
3. **Certificate installed in server**
4. **Browser issued with root certificates**: Browsers come with knows certificates, so it can compare and see if the cert is really valid.
5. **Browser trusts correctly signed certs**: Since your browser has the public key of lots of certs, when a request comes in, it is able to verify the certificate using the public key (?)

https://tiptopsecurity.com/how-does-https-work-rsa-encryption-explained/

Are all URLs encrypted when using TLS/SSL (HTTPS) encryption?



Yes, the SSL connection is between the TCP layer and the HTTP layer. The client and server first establish a secure encrypted TCP connection (via the SSL/TLS protocol) and then the client will send the HTTP request (GET, POST, DELETE...) over that encrypted TCP connection.



Domain name MAY be transmitted in clear (if SNI extension is used in the TLS handshake) but URL (path and parameters) is always encrypted.

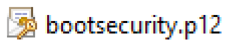
Enable HTTPS/SSL

3 Steps:

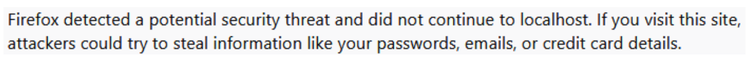
1. **Certificate**: Can be self signed or we can buy one. For production don't use self signed certificates!
2. **Modify app.properties:** SSL ports, name of the cert etc
3. **Add @Bean for ServletWebServerFactory**: Our traffic is handled by a web server and we are using an embedded servlet container, we need to tell it that we want to use http and redirect all http traffic to the https port.

cd to JDK/bin directory

Run as Admin or su: `.\keytool.exe -genkeypair -alias bootsecurity -keyalg RSA -keysize 2048 -storetype PKCS12 -keystore bootsecurity.p12 -validity 3650`


bootsecurity.p12

```
server.port=443
server.ssl.enabled=true
server.ssl.key-store=src/main/resources/bootsecurity.p12
server.ssl.key-store-password=rootroot
server.ssl.key-store-type=PKCS12
server.ssl.key-alias=bootsecurity
```


Firefox detected a potential security threat and did not continue to localhost. If you visit this site, attackers could try to steal information like your passwords, emails, or credit card details.

localhost:8080

redirects to

https://localhost

```
@SpringBootApplication
public class CourseApiApp {
  public static void main(String[] args) throws Exception {
    SpringApplication.run(CourseApiApp.class, args);
  }

  @Bean
  public ServletWebServerFactory servletContainer() {
    // Enable SSL Trafic
    TomcatServletWebServerFactory tomcat = new TomcatServletWebServerFactory() {
      @Override
      protected void postProcessContext(Context context) {
        SecurityConstraint securityConstraint = new SecurityConstraint();
        securityConstraint.setUserConstraint("CONFIDENTIAL");
        SecurityCollection collection = new SecurityCollection();
        collection.addPattern("/*");
        securityConstraint.addCollection(collection);
        context.addConstraint(securityConstraint);
      }
    };

    // Add HTTP to HTTPS redirect
    tomcat.addAdditionalTomcatConnectors(httpToHttpsRedirectConnector());

    return tomcat;
  }

  /*
  We need to redirect from HTTP to HTTPS. Without SSL, this application used
  port 8080. With SSL it will use port 443. So, any request for 8080 needs to be
  redirected to HTTPS on 443.
   */
  private Connector httpToHttpsRedirectConnector() {
    Connector connector = new Connector(TomcatServletWebServerFactory.DEFAULT_PROTOCOL);
    connector.setScheme("http");
    connector.setPort(8080);
    connector.setSecure(false);
    connector.setRedirectPort(443);
    return connector;
  }
```

Authentication with Database

These are the steps we need to do in order to store user information in a database:

1. **Create a User entity to store user information**: We could store whatever we want, roles etc.
2. **Store the User in our database**: We need to create a repository for that. In this example it will be a JPA repository but you can create whatever class you need. This repository is just an interface for where we can extract users, it can be from a db, an excel file, in memory etc. The impl of the repo is up to us.
3. **Link our User entity with the built in classes in Spring Security**: This is the most tricky part. Spring security gives us the flexibility to define our own users and also how to retrieve it. We are not bound to a specific technology, that will be our choice.
   a) Link `User` with `UserDetails` interface
   b) Link `UserRepository` with `UserDetailsService` interface
4. **Integrate Database Auth in our configuration**

1. Create a User entity to store user information

2. Store the User in our database

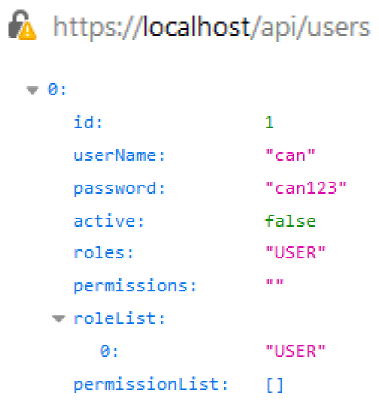
```java
@Entity
public class User {
  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private long id;

  @Column(nullable = false)
  private String userName;

  @Column(nullable = false)
  private String password;

  private boolean active = true;
  private String roles = "";
  private String permissions = "";

  public User() { // Don't forget the default constructor
  }

  public User(String userName, String password,
      String roles, String permissions) {
    super();
    this.userName = userName;
    this.password = password;
    this.roles = roles;
    this.permissions = permissions;
  }
```

```java
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface UserRepository extends JpaRepository<User, Long> {

  User findByUserName(String userName);
}
```

```java
@Service
public class DatabaseInit implements CommandLineRunner {

  private UserRepository userRepository;

  public DatabaseInit(UserRepository userRepository) {
    this.userRepository = userRepository;
  }

  @Override
  public void run(String... args) throws Exception {

    User can = new User("can", "can123", "USER", "");
    User admin = new User("admin", "admin123", "ADMIN", "");
    User manager = new User("manager", "manager123", "MANAGER", "");

    List<User> users = Arrays.asList(can, admin, manager);
    this.userRepository.saveAll(users);
```

For testing:

```java
@RestController
@RequestMapping("/api")
public class PublicRestApiController {

  private UserRepository userRepository;

  public PublicRestApiController(UserRepository userRepository) {
    this.userRepository = userRepository;
  }

  // to test if everything works
  @GetMapping("/users")
  public List<User> allUsers() {
    return userRepository.findAll();
  }
```



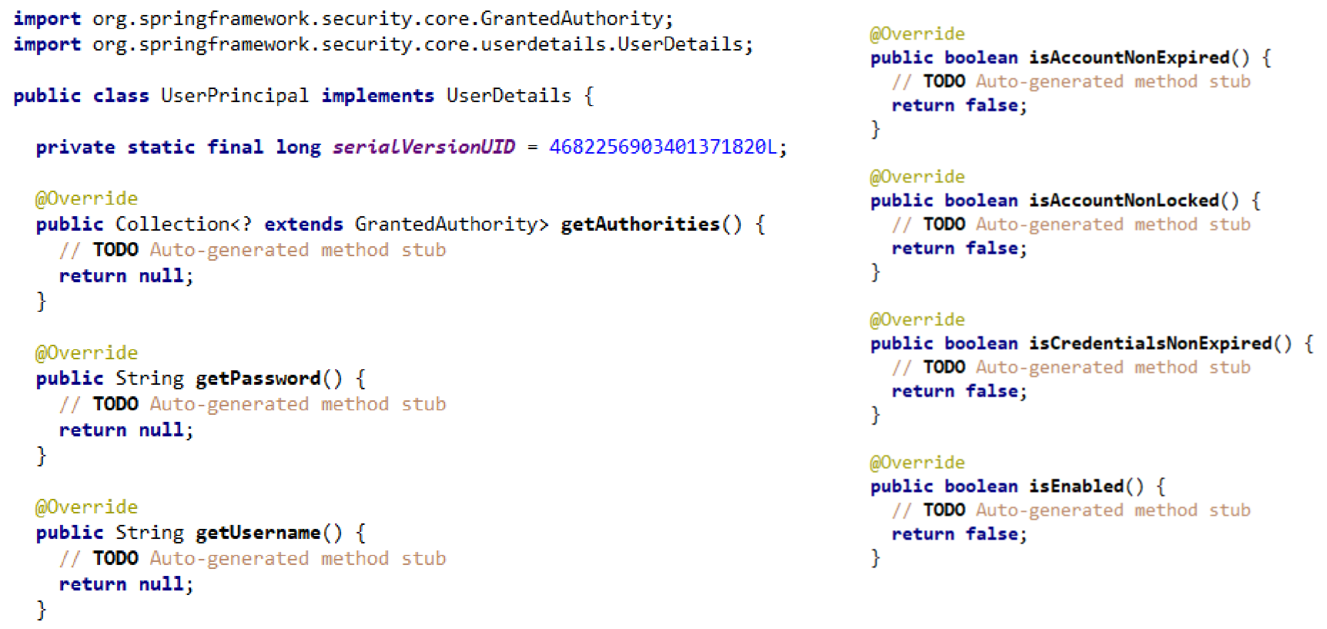
```
https://localhost/api/users

▼ 0:
    id:               1
    userName:         "can"
    password:         "can123"
    active:           false
    roles:            "USER"
    permissions:      ""
  ▼ roleList:
      0:              "USER"
    permissionList:   []
```

3. Implement UserDetails and UserDetailsService

We need to provide implementations for both UserDetails and UserDetailsService.

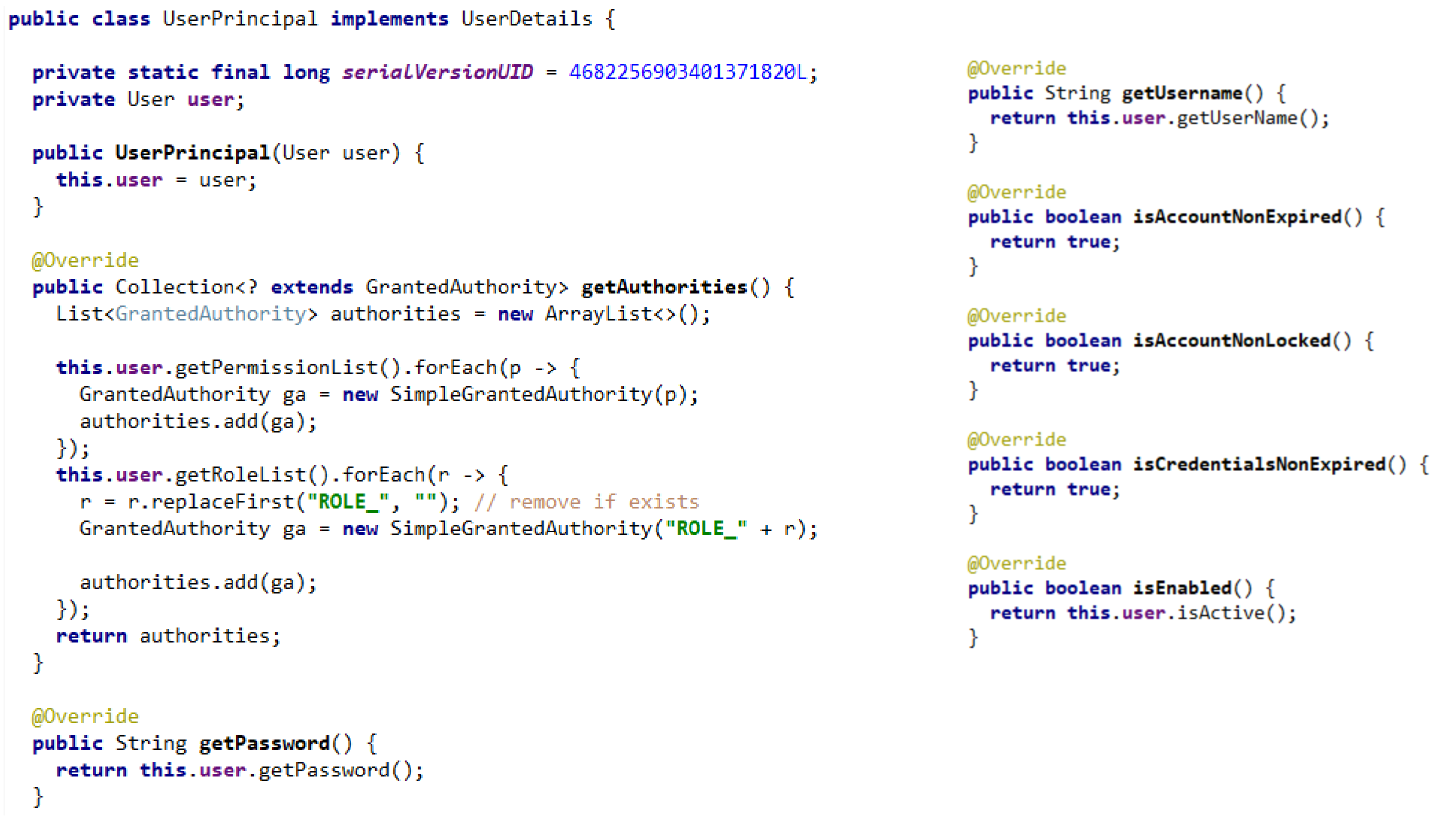
Spring Security needs to know from a User from security point of view, what are his authorities, username, password, locked etc. But inside our own application we can have many more properties like email address, phone number, bday, department etc. So these are the minimums to configure security but it can be extended to suite specific needs.

We have to somehow map this **`UserPrincipal`** class to our **`User`** model.



```java
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;

public class UserPrincipal implements UserDetails {

  private static final long serialVersionUID = 4682256903401371820L;

  @Override
  public Collection<? extends GrantedAuthority> getAuthorities() {
    // TODO Auto-generated method stub
    return null;
  }

  @Override
  public String getPassword() {
    // TODO Auto-generated method stub
    return null;
  }

  @Override
  public String getUsername() {
    // TODO Auto-generated method stub
    return null;
  }

@Override
public boolean isAccountNonExpired() {
  // TODO Auto-generated method stub
  return false;
}

@Override
public boolean isAccountNonLocked() {
  // TODO Auto-generated method stub
  return false;
}

@Override
public boolean isCredentialsNonExpired() {
  // TODO Auto-generated method stub
  return false;
}

@Override
public boolean isEnabled() {
  // TODO Auto-generated method stub
  return false;
}
```

One way is to use the **decorator pattern (?)** -> Use UserPrincipal as a wrapper for User.



```java
public class UserPrincipal implements UserDetails {

  private static final long serialVersionUID = 4682256903401371820L;
  private User user;

  public UserPrincipal(User user) {
    this.user = user;
  }

  @Override
  public Collection<? extends GrantedAuthority> getAuthorities() {
    List<GrantedAuthority> authorities = new ArrayList<>();

    this.user.getPermissionList().forEach(p -> {
      GrantedAuthority ga = new SimpleGrantedAuthority(p);
      authorities.add(ga);
    });
    this.user.getRoleList().forEach(r -> {
      r = r.replaceFirst("ROLE_", ""); // remove if exists
      GrantedAuthority ga = new SimpleGrantedAuthority("ROLE_" + r);

      authorities.add(ga);
    });
    return authorities;
  }

  @Override
  public String getPassword() {
    return this.user.getPassword();
  }

@Override
public String getUsername() {
  return this.user.getUserName();
}

@Override
public boolean isAccountNonExpired() {
  return true;
}

@Override
public boolean isAccountNonLocked() {
  return true;
}

@Override
public boolean isCredentialsNonExpired() {
  return true;
}

@Override
public boolean isEnabled() {
  return this.user.isActive();
}
```

```java
public class UserPrincipalDetailsService implements UserDetailsService {

  private UserRepository userRepository;

  public UserPrincipalDetailsService(UserRepository userRepository) {
    this.userRepository = userRepository;
  }

  @Override
  public UserDetails loadUserByUsername(String username)
      throws UsernameNotFoundException {

    User user = this.userRepository.findByUserName(username);
    UserPrincipal userPrincipal = new UserPrincipal(user);
    return userPrincipal;
  }
```

4. Integrate Database Auth in our configuration

The next step will be to switch using authentication with the database. Right now we still use the in memory security configuration. We need to switch that to the db auth mode.

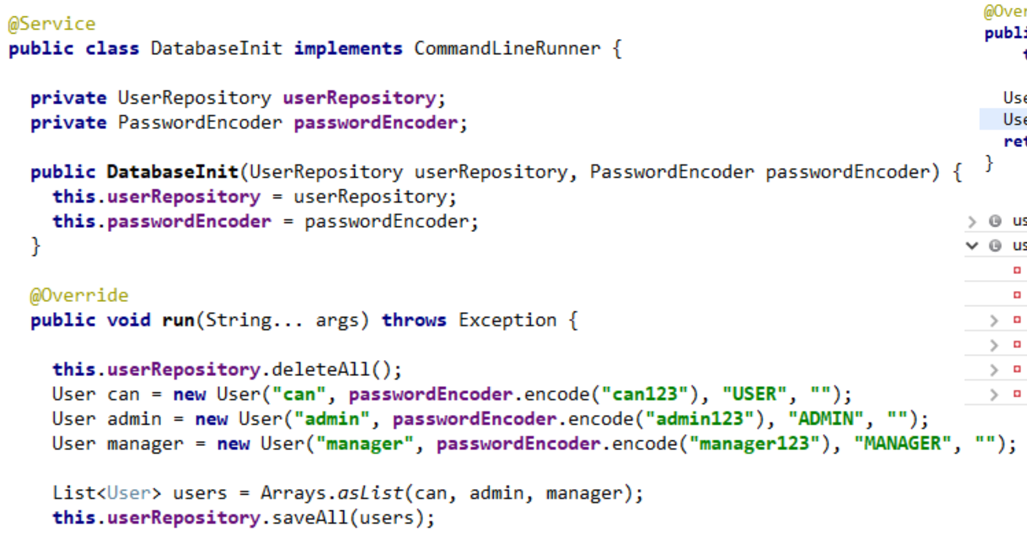
```java
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
  auth
      .inMemoryAuthentication()
      .withUser("admin")
      .password(passwordEncoder().encode("admin123"))
      .authorities("ACCESS_TEST1", "ACCESS_TEST2", "ROLE_ADMIN")
      .and()
      .withUser("can")
      .password(passwordEncoder().encode("can123"))
      .roles("USER")
      .and()
      .withUser("manager")
      .password(passwordEncoder().encode("manager123"))
      .authorities("ACCESS_TEST1", "ROLE_MANAGER");
```

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

  @Autowired
  private UserPrincipalDetailsService userPrincipalDetailsService;

  @Bean
  public DaoAuthenticationProvider authProvider() {
    DaoAuthenticationProvider authProvider = new DaoAuthenticationProvider();
    authProvider.setPasswordEncoder(passwordEncoder());
    authProvider.setUserDetailsService(userPrincipalDetailsService);
    return authProvider;
  }

  @Bean
  public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
  }

  @Override
  protected void configure(AuthenticationManagerBuilder auth) {
    auth.authenticationProvider(authProvider());
  }

  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers("/").permitAll() // no auth needed to access
        .antMatchers("/profile/**").authenticated()
        .antMatchers("/admin/**").hasRole("ADMIN")
        .antMatchers("/management/**").hasAnyRole("ADMIN", "MANAGER")
        .antMatchers("/api/test1").hasAnyAuthority("ACCESS_TEST1", "ROLE_MANAGER", "ROLE_ADMIN")
        .antMatchers("/api/test2").hasAnyAuthority("ACCESS_TEST2", "ROLE_ADMIN")
        .and()
        .httpBasic();
  }
```

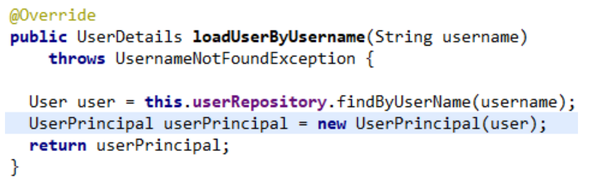
```java
@Service
public class UserPrincipalDetailsService
```

Right now when I started my app and accessed /api/test1 and entered credentials, it did not work. The reason was that in our DatabaseInit class, we saved the password as a plain text "manager123". Setting a breakpoint in UserPrincipal showed that when we get the user back from our userRepository, the password was "manager123".

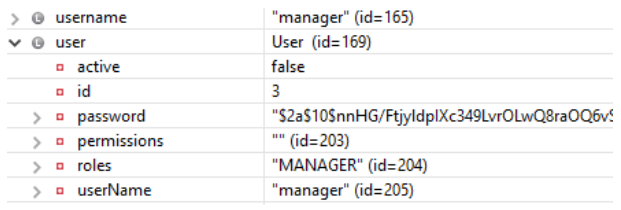
- But we are using a password encoder for our DaoAuthenticationProvider. This means the encoded password and the plain text one does not match. So we have to encode our password before we save it to the database.



```java
@Service
public class DatabaseInit implements CommandLineRunner {

  private UserRepository userRepository;
  private PasswordEncoder passwordEncoder;

  public DatabaseInit(UserRepository userRepository, PasswordEncoder passwordEncoder) {
    this.userRepository = userRepository;
    this.passwordEncoder = passwordEncoder;
  }

  @Override
  public void run(String... args) throws Exception {

    this.userRepository.deleteAll();
    User can = new User("can", passwordEncoder.encode("can123"), "USER", "");
    User admin = new User("admin", passwordEncoder.encode("admin123"), "ADMIN", "");
    User manager = new User("manager", passwordEncoder.encode("manager123"), "MANAGER", "");

    List<User> users = Arrays.asList(can, admin, manager);
    this.userRepository.saveAll(users);
```

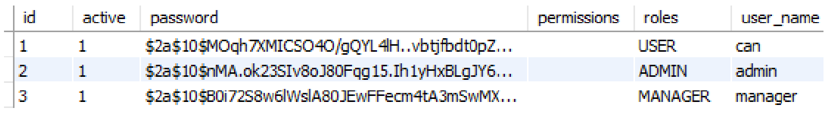


```java
@Override
public UserDetails loadUserByUsername(String username)
    throws UsernameNotFoundException {

  User user = this.userRepository.findByUserName(username);
  UserPrincipal userPrincipal = new UserPrincipal(user);
  return userPrincipal;
}
```



username	"manager" (id=165)
user	User (id=169)
active	false
id	3
password	"$2a$10$nnHG/FtjyIdpIXc349LvrOLwQ8raOQ6v$
permissions	"" (id=203)
roles	"MANAGER" (id=204)
userName	"manager" (id=205)

Now everything works as we want, correct permissions etc.

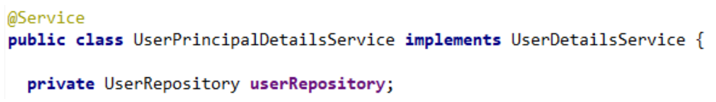


id	active	password	permissions	roles	user_name
1	1	$2a$10$MOqh7XMICSO4O/gQYL4lH..vbtjfbdt0pZ...		USER	can
2	1	$2a$10$nMA.ok23SIv8oJ80Fqg15.Ih1yHxBLgJY6...		ADMIN	admin
3	1	$2a$10$B0i72S8w6lWslA80JEwFFecm4tA3mSwMX...		MANAGER	manager

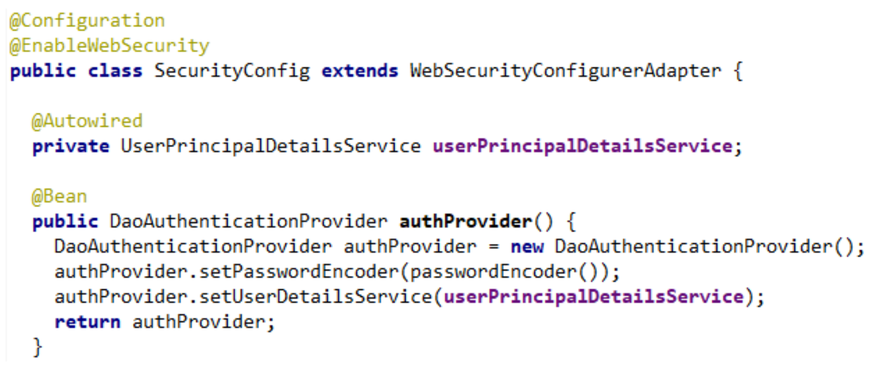
Important Things to Note

Remember that the store mechanism **is decoupled** from the actual authentication method.

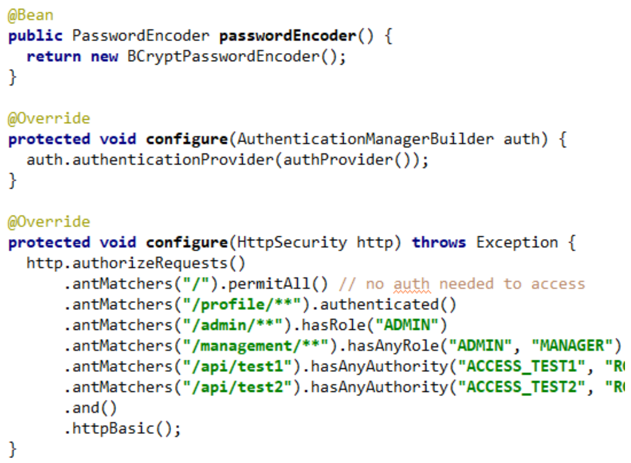
- For the storage we can use file system, cloud, database, in memory etc. and the only thing we would need to change is the **UserRepository implementation** of how and from where we get the data.
- The `SecurityConfig` would stay the same even if we changed the storage, because our `DaoAuthenticationProvider` cares only about a `UserPrincipalDetailsService` which again only cares about a `UserRepository` implementation.



```java
@Service
public class UserPrincipalDetailsService implements UserDetailsService {

  private UserRepository userRepository;
```



```java
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

  @Autowired
  private UserPrincipalDetailsService userPrincipalDetailsService;

  @Bean
  public DaoAuthenticationProvider authProvider() {
    DaoAuthenticationProvider authProvider = new DaoAuthenticationProvider();
    authProvider.setPasswordEncoder(passwordEncoder());
    authProvider.setUserDetailsService(userPrincipalDetailsService);
    return authProvider;
  }
```



```java
@Bean
public PasswordEncoder passwordEncoder() {
  return new BCryptPasswordEncoder();
}

@Override
protected void configure(AuthenticationManagerBuilder auth) {
  auth.authenticationProvider(authProvider());
}

@Override
protected void configure(HttpSecurity http) throws Exception {
  http.authorizeRequests()
      .antMatchers("/").permitAll() // no auth needed to access
      .antMatchers("/profile/**").authenticated()
      .antMatchers("/admin/**").hasRole("ADMIN")
      .antMatchers("/management/**").hasAnyRole("ADMIN", "MANAGER")
      .antMatchers("/api/test1").hasAnyAuthority("ACCESS_TEST1", "R
      .antMatchers("/api/test2").hasAnyAuthority("ACCESS_TEST2", "R
      .and()
      .httpBasic();
}
```