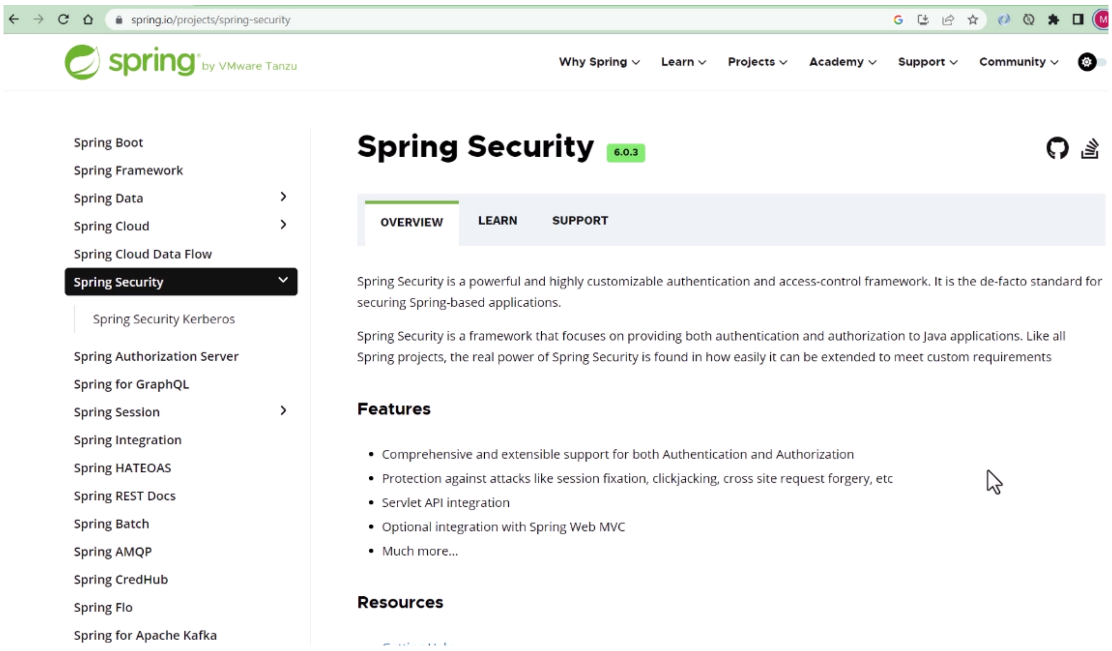
Spring Security

1

Spring Security Project


spring.io/projects/spring-security

Spring Security 6.0.3

Spring Security is a powerful and highly customizable authentication and access-control framework. It is the de-facto standard for securing Spring-based applications.

Spring Security is a framework that focuses on providing both authentication and authorization to Java applications. Like all Spring projects, the real power of Spring Security is found in how easily it can be extended to meet custom requirements

Features

- Comprehensive and extensible support for both Authentication and Authorization
- Protection against attacks like session fixation, clickjacking, cross site request forgery, etc
- Servlet API integration
- Optional integration with Spring Web MVC
- Much more...

Resources

2

Spring Security

- *Spring Security is a powerful and highly customizable authentication and access-control framework. It is the de-facto standard for securing Spring-based applications.*
- *Below is the maven dependency that we can add to implement security using Spring Security project in any of the SpringBoot projects,*

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

If we are using Spring Boot, then we need to include the starter dependency!

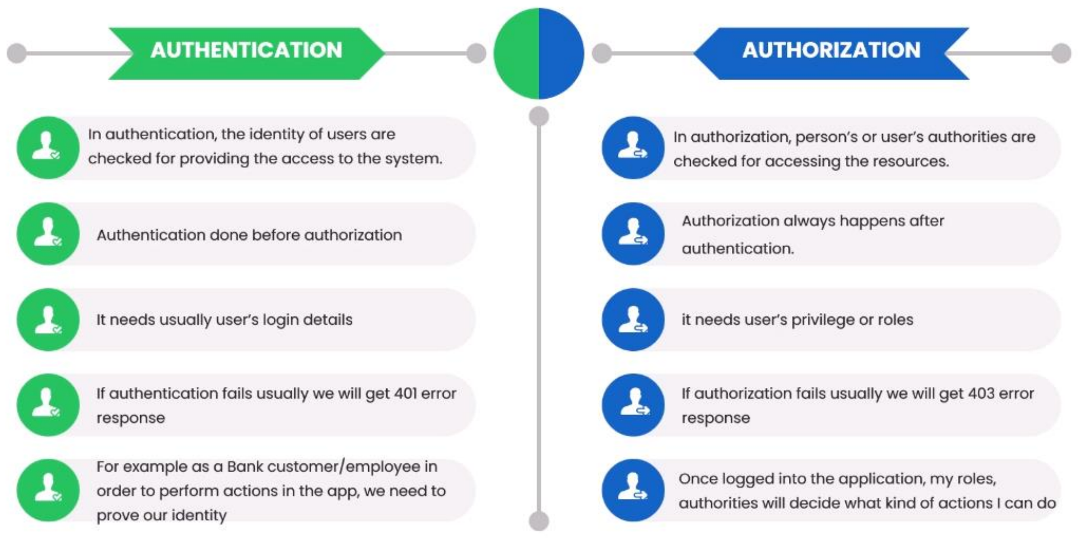
Spring Security is a framework that provides authentication, authorization, and protection against common attacks.

Spring Security helps developers with easier configurations to secure a web application by using standard username/password authentication mechanism.

e.g. the same configurations can be used for MVC web apps and REST webservices

Spring Security provides out of the box features to handle common security attacks like CSRF, CORs. It also has good integration with security standards like JWT, OAUTH2 etc.

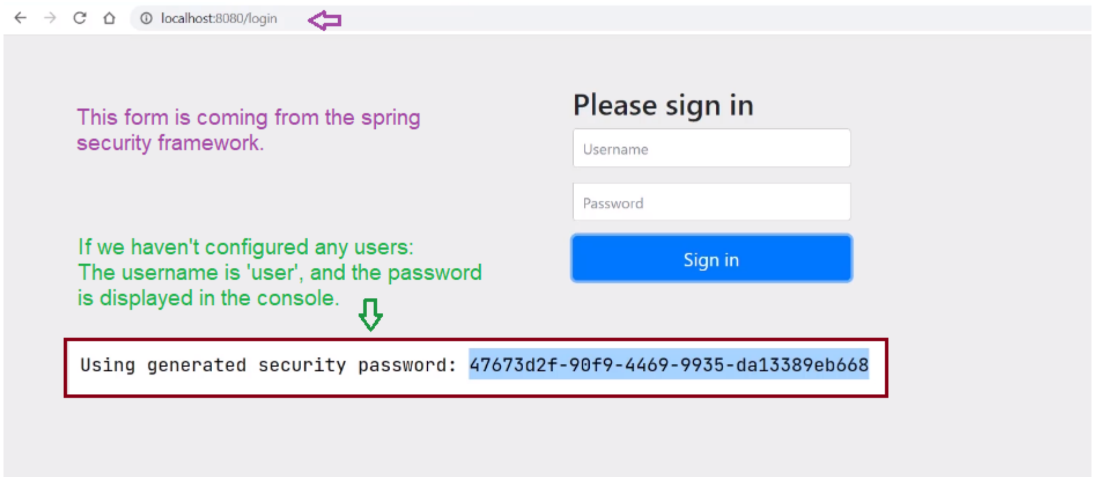
Authentication & Authorization



AUTHENTICATION	AUTHORIZATION
In authentication, the identity of users are checked for providing the access to the system.	In authorization, person's or user's authorities are checked for accessing the resources.
Authentication done before authorization	Authorization always happens after authentication.
It needs usually user's login details	it needs user's privilege or roles
If authentication fails usually we will get 401 error response	If authorization fails usually we will get 403 error response
For example as a Bank customer/employee in order to perform actions in the app, we need to prove our identity	Once logged into the application, my roles, authorities will decide what kind of actions I can do

Spring Security Default Behavior

- If spring-boot-starter-security is added to the application, the login form is automatically added
  - All requests redirect to the login form.
    - Once we log in, Spring will not ask for the credentials for subsequent requests.

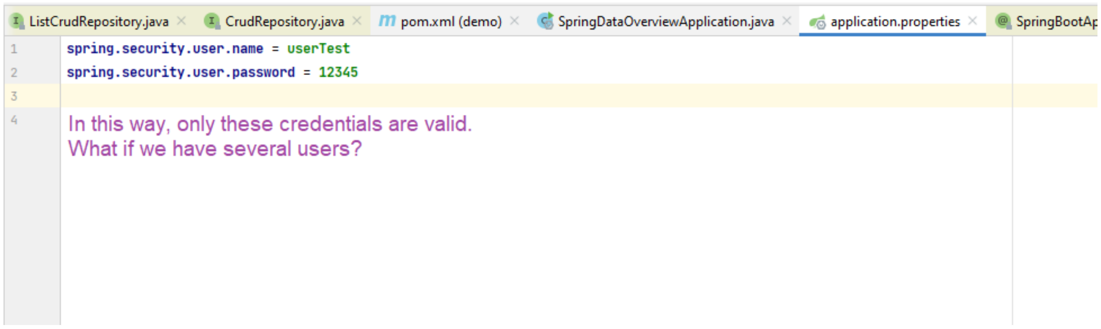

localhost:8080/login
This form is coming from the spring security framework.
Please sign in
Username
Password
Sign in
If we haven't configured any users:
The username is 'user', and the password is displayed in the console.

```
Using generated security password: 47673d2f-90f9-4469-9935-da13389eb668
```

5

Configure Custom Credentials

- Use 'application.properties':



```
spring.security.user.name = userTest
spring.security.user.password = 12345
```

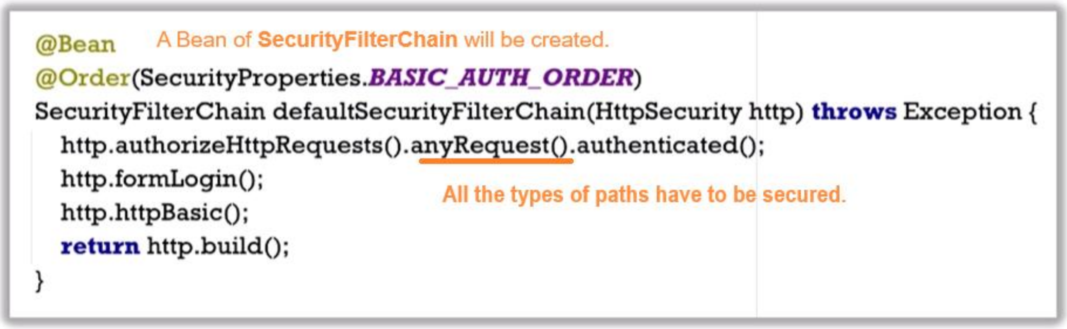
In this way, only these credentials are valid.
What if we have several users?

- This approach is not suitable for production applications.

6

Default Security Configurations in Spring Security

By default, Spring Security framework protects all the paths present inside the web application. This behaviour is due to the code present inside the method defaultSecurityFilterChain(HttpSecurity http) of class SpringBootWebSecurityConfiguration



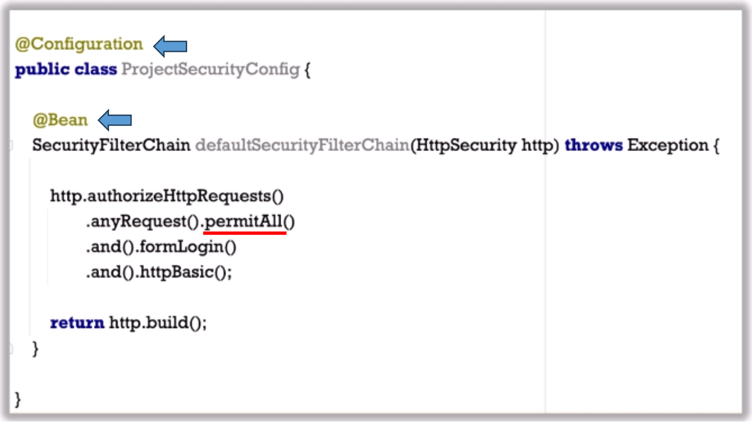
```
@Bean        A Bean of SecurityFilterChain will be created.
@Order(SecurityProperties.BASIC_AUTH_ORDER)
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
   http.authorizeHttpRequests().anyRequest().authenticated();
   http.formLogin();
   http.httpBasic();                   All the types of paths have to be secured.
   return http.build();
}
```

- This is the piece of code inside the Spring Security that will protect all the paths inside our web application!
  - If the user specifies their own **SecurityFilterChain** bean, this will back-off completely and the users should specify all the bits that they want to configure as part of the custom security configuration.

7

Configure **permitAll()** with Spring Security

- We can customize the security configuration.
  - We need to create a bean of the type SecurityFilterChain.
    - E.g. if we want to allow full/public access to all the pages without any security.



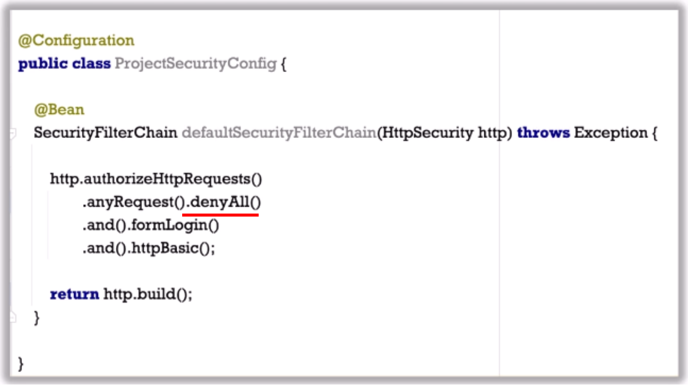
```
@Configuration
public class ProjectSecurityConfig {

  @Bean
  SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {

    http.authorizeHttpRequests()
        .anyRequest().permitAll()
        .and().formLogin()
        .and().httpBasic();

    return http.build();
  }

}
```

- Form Login provides support for username and password being provided through an html form
- HTTP Basic Auth uses an HTTP header in order to provide the username and password when making a request to a server.

8

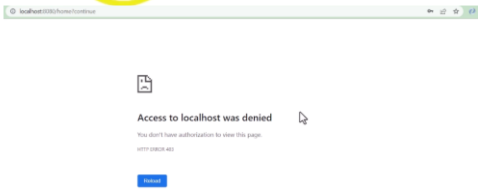
Configure **denyAll()** with Spring Security

- E.g. if we want to **deny** any request that is coming into a web application:
  - In this case the login page will be displayed, but even if we add valid credentials, the access will be denied (*Error code **403***).
    - This is because authentication comes before authorization!



```java
@Configuration
public class ProjectSecurityConfig {

  @Bean
  SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {

    http.authorizeHttpRequests()
          .anyRequest().denyAll()
          .and().formLogin()
          .and().httpBasic();

    return http.build();
  }

}
```

- Usually denyAll() is used to retire a specific API temporarily with out removing the code
- permitAll() is used to allow public access to public APIs, paths, CSS, images, JS files etc.



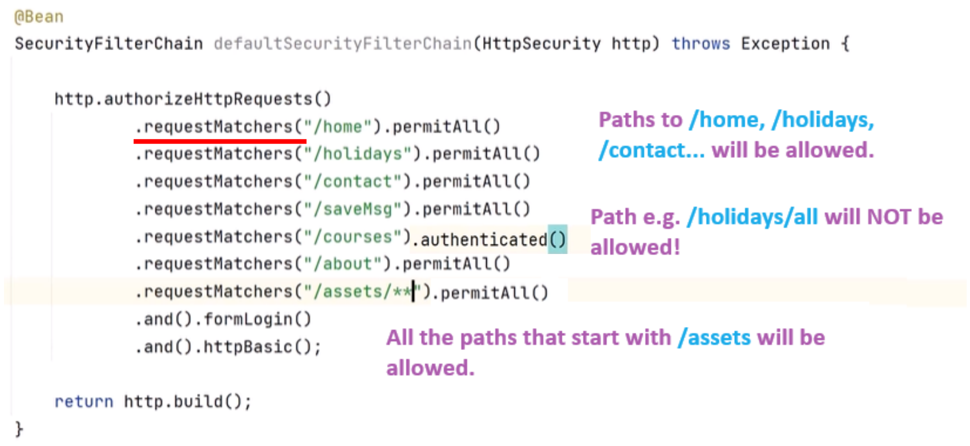
localhost:8080/home/continue

Access to localhost was denied

You don't have authorization to view this page.

HTTP ERROR 403

Configure Spring Security at the API path level



```java
@Bean
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {

    http.authorizeHttpRequests()
            .requestMatchers("/home").permitAll()
            .requestMatchers("/holidays").permitAll()
            .requestMatchers("/contact").permitAll()
            .requestMatchers("/saveMsg").permitAll()
            .requestMatchers("/courses").authenticated()
            .requestMatchers("/about").permitAll()
            .requestMatchers("/assets/**").permitAll()
            .and().formLogin()
            .and().httpBasic();

    return http.build();
}
```

Paths to /home, /holidays, /contact… will be allowed.

Path e.g. /holidays/all will NOT be allowed!

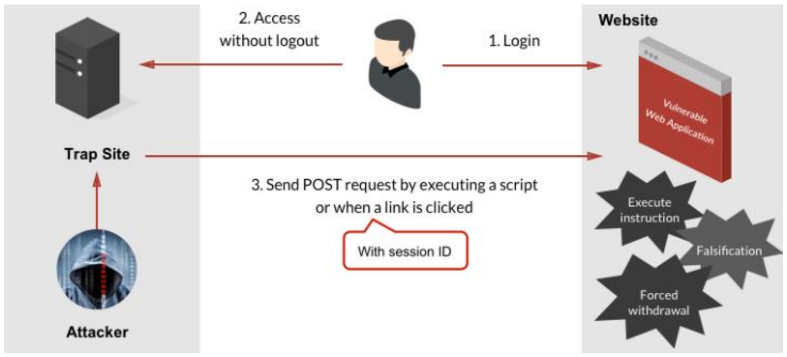
All the paths that start with /assets will be allowed.

If e.g. inside the **assets** folder, we have all our javascript, css, images, etc. files, we have to permit all access to all these resources, otherwise our content will not look / behave correctly.

CSRF Attack

- CSRF (Cross-Site Request Forgery) is a type of security attack where a malicious website or application tricks a user's browser into making unintended requests to a different website where the user is authenticated.
  - This can lead to unauthorized actions being performed on the user's behalf without their consent.

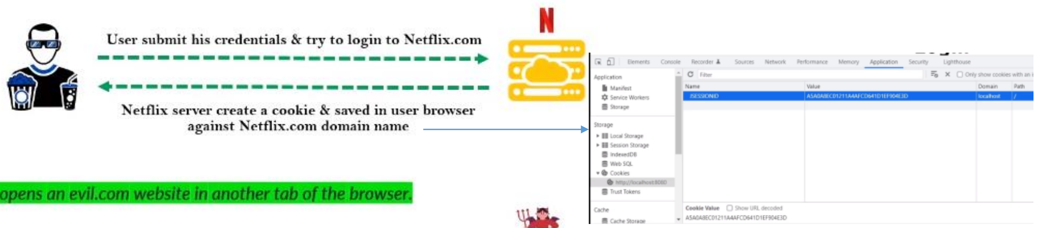
1.User Authentication: The user logs into a legitimate website (e.g., an online banking site) and receives an authentication token (usually in the form of a cookie or session ID) that is stored in their browser.
2.Malicious Website: The user visits a malicious website that contains a hidden form or script designed to exploit the user's authenticated session on the legitimate website.
3.Hidden Form Submission: The malicious website automatically submits a form or sends a request to the legitimate website in the background, using the user's stored authentication token without their knowledge.
4.Unintended Action: Since the request contains the user's authentication token, the legitimate website interprets it as a valid request from the authenticated user and performs the action specified by the attacker (e.g., transferring funds, changing account settings).



11

CSRF - Example

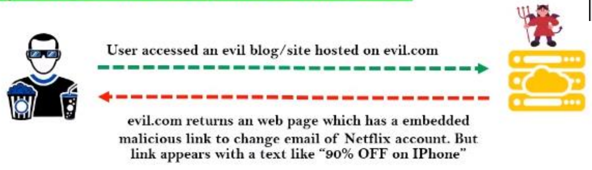
Step 1 : The Netflix user login to Netflix.com and the backend server of Netflix will provide a cookie which will store in the browser against the domain name Netflix.com



Name	Value	Domain	Path
JSESSIONID	A5A0A8EC01211A4AFCD641D1EF904E3D	localhost	/

Cookie Value
A5A0A8EC01211A4AFCD641D1EF904E3D

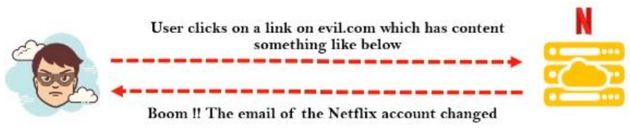
Step 2 : The same Netflix user opens an evil.com website in another tab of the browser.

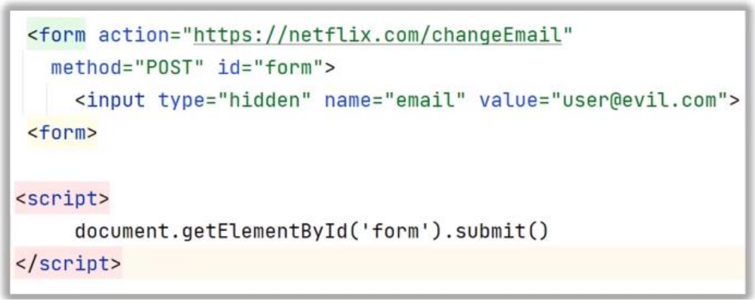

User accessed an evil blog/site hosted on evil.com
evil.com returns an web page which has a embedded malicious link to change email of Netflix account. But link appears with a text like "90% OFF on IPhone"

12

CSRF – Example...

Step 3 : User tempted and clicked on the malicious link which makes a request to Netflix.com. And since the login cookie already present in the same browser and the request to change email is being made to the same domain Netflix.com, the backend server of Netflix.com can't differentiate from where the request came. So here the evil.com forged the request as if it is coming from a Netflix.com UI page.

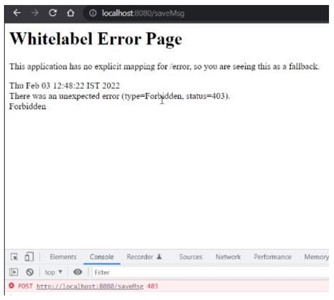

User clicks on a link on evil.com which has content something like below
N
Boom !! The email of the Netflix account changed



```
<form action="https://netflix.com/changeEmail"
   method="POST" id="form">
      <input type="hidden" name="email" value="user@evil.com">
<form>

<script>
      document.getElementById('form').submit()
</script>
```

How Spring Security blocks any requests that update data

- If we don't handle CSRF in the application, in the way Spring Security is expecting, then by default **all POST/PUT requests will be blocked**.



localhost:8080/saveMsg

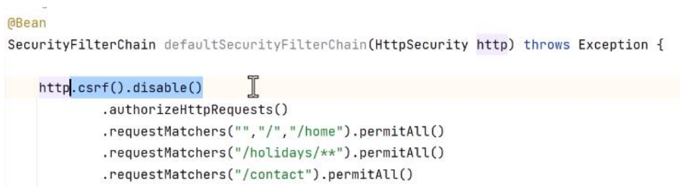
Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Thu Feb 03 12:48:22 IST 2022
There was an unexpected error (type=Forbidden, status=403).
Forbidden

```
POST http://localhost:8080/saveMsg 403
```

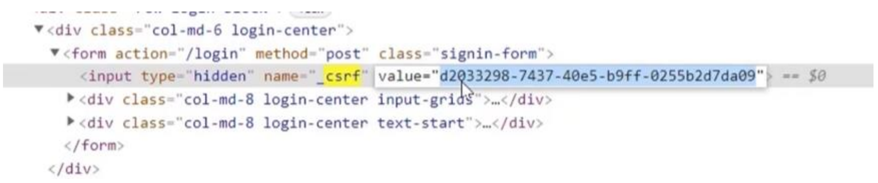
- How to disable if there is no security threat?



```
@Bean
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {

    http.csrf().disable()
            .authorizeHttpRequests()
            .requestMatchers("","/","/home").permitAll()
            .requestMatchers("/holidays/**").permitAll()
            .requestMatchers("/contact").permitAll()
```

Solution to CSRF

- Can we remove the use of cookies?
  - No, because otherwise, we will need to ask the user for the credentials before any action.
- To defeat a CSRF attack, applications need a way to determine if the HTTP request is legitimately generated via the application's user interface. This can be achieved through a CSRF token.
  - **CSRF token**: is a secure random token that is used to prevent CSRF attacks. The token needs to be unique per user session and should be of large random value to make it difficult to guess.
    - This token will never be stored as a cookie in the browser… it will be inserted as a hidden parameter inside all the forms.

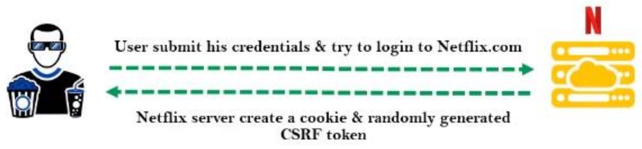


```
▼<div class="col-md-6 login-center">
  ▼<form action="/login" method="post" class="signin-form">
     <input type="hidden" name="_csrf" value="d2033298-7437-40e5-b9ff-0255b2d7da09"> == $0
    ▶<div class="col-md-8 login-center input-grids">…</div>
    ▶<div class="col-md-8 login-center text-start">…</div>
   </form>
 </div>
```

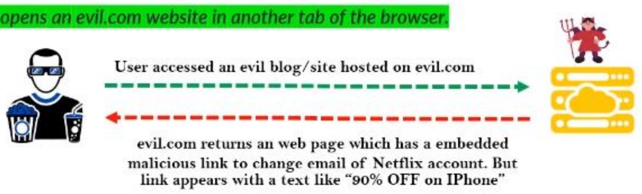
15

Solution to CSRF - Example

Step 1 : The Netflix user login to Netflix.com and the backend server of Netflix will provide a cookie which will store in the browser against the domain name Netflix.com along with a randomly generated unique CSRF token for this particular user session. CSRF token is inserted within hidden parameters of HTML forms to avoid exposure to session cookies.


N
User submit his credentials & try to login to Netflix.com
Netflix server create a cookie & randomly generated CSRF token

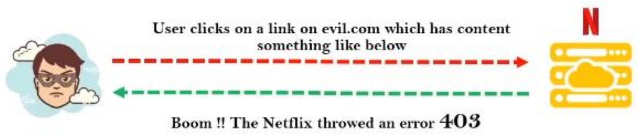
Step 2 : The same Netflix user opens an evil.com website in another tab of the browser.


User accessed an evil blog/site hosted on evil.com
evil.com returns an web page which has a embedded malicious link to change email of Netflix account. But link appears with a text like "90% OFF on IPhone"

16

Solution to CSRF – Example…

Step 3 : User tempted and clicked on the malicious link which makes a request to Netflix.com. And since the login cookie already present in the same browser and the request to change email is being made to the same domain Netflix.com. This time the Netflix.com backend server expects CSRF token along with the cookie. The CSRF token must be same as initial value generated during login operation


User clicks on a link on evil.com which has content something like below
Boom !! The Netflix throwed an error 403

The CSRF token will be used by the application server to verify the legitimacy of the end-user request if it is coming from the same App UI or not. The application server rejects the request if the CSRF token fails to match the test.

The CSRF token is saved at the session inside the **server**. The UI application should send the CSRF token as a hidden field for every request!

17

Summary

- ✓ **By default, Spring Security enables CSRF fix for all the HTTP methods which results in data change like POST, DELETE etc. But not for GET.**

- ✓ **Using Spring Security configurations we can disable the CSRF protection for complete application or for only few paths based on our requirements like below.**

  - ***http.csrf((csrf) -> csrf.disable())***
  - ***http.csrf((csrf) -> csrf.ignoringRequestMatchers("/saveMsg"))***

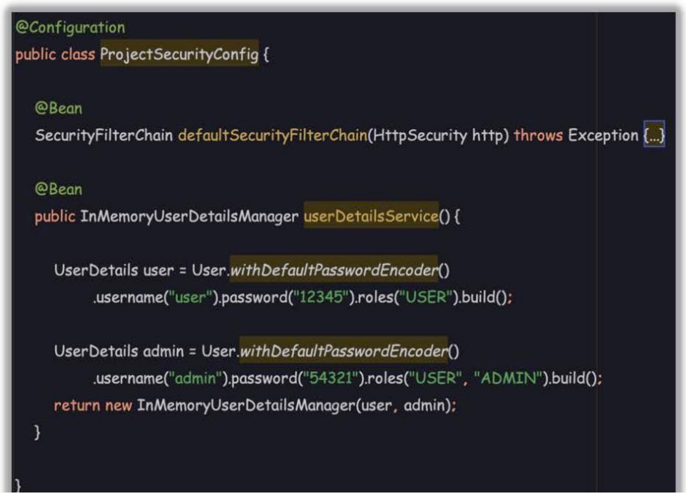
- ✓ **Thymeleaf has great integration & support with Spring Security to generate a CSRF token. We just need to add the below code in login html form code and Thymeleaf will automatically appends the CSRF token for the remaining pages/forms inside the web application,**

```
<input type="hidden" th:name="${_csrf.parameterName}" th:value="${_csrf.token}" />
```

18

How to configure multiple users?

- *Spring Security provide support for username/password based authentication based on the users stored in application memory.*
- *Like mentioned below, we can configure any number of users & their roles, passwords using in-memory authentication,*



```
@Configuration
public class ProjectSecurityConfig {

  @Bean
  SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {...}

  @Bean
  public InMemoryUserDetailsManager userDetailsService() {

    UserDetails user = User.withDefaultPasswordEncoder()
          .username("user").password("12345").roles("USER").build();

    UserDetails admin = User.withDefaultPasswordEncoder()
          .username("admin").password("54321").roles("USER", "ADMIN").build();
    return new InMemoryUserDetailsManager(user, admin);
  }

}
```

in-memory authentication is idle for POC Web Apps or any internal Web App that get used only in non-prod environments. NEVER EVER use in-memory authentication for PROD web applications.

This is the in-memory authentication available in Spring Security.

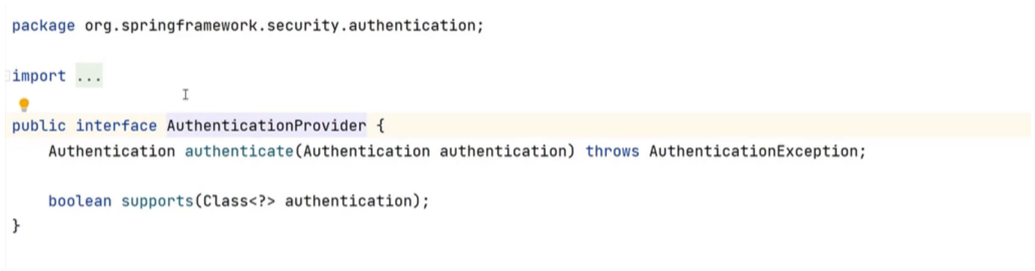
But, in real applications, we usually don't want to hardcode the credentials in Java code.

We should store the credentials in a database or other storage system.

19

Spring Security AuthenticationProvider

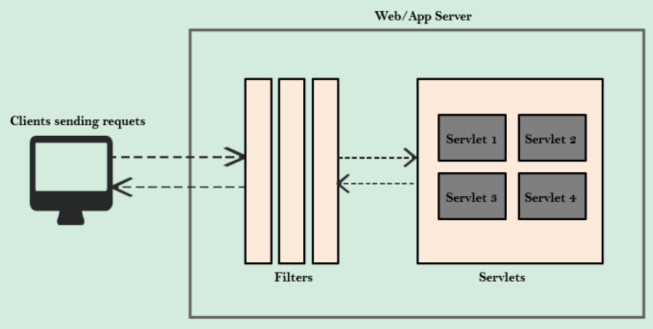
- Spring Security allows us to write our custom logic to authenticate a user based on our requirements by implementing the AuthenticationProvider interface.
  - This allows additional flexibility compared to the standard scenario using a simple UserDetailsService
- An AuthenticationProvider, processes an Authentication request, and a fully authenticated object with full credentials is returned.



```
package org.springframework.security.authentication;

import ...

public interface AuthenticationProvider {
    Authentication authenticate(Authentication authentication) throws AuthenticationException;

    boolean supports(Class<?> authentication);
}
```

20

Servlets and Filters



★ **Typical Scenario inside a web application**

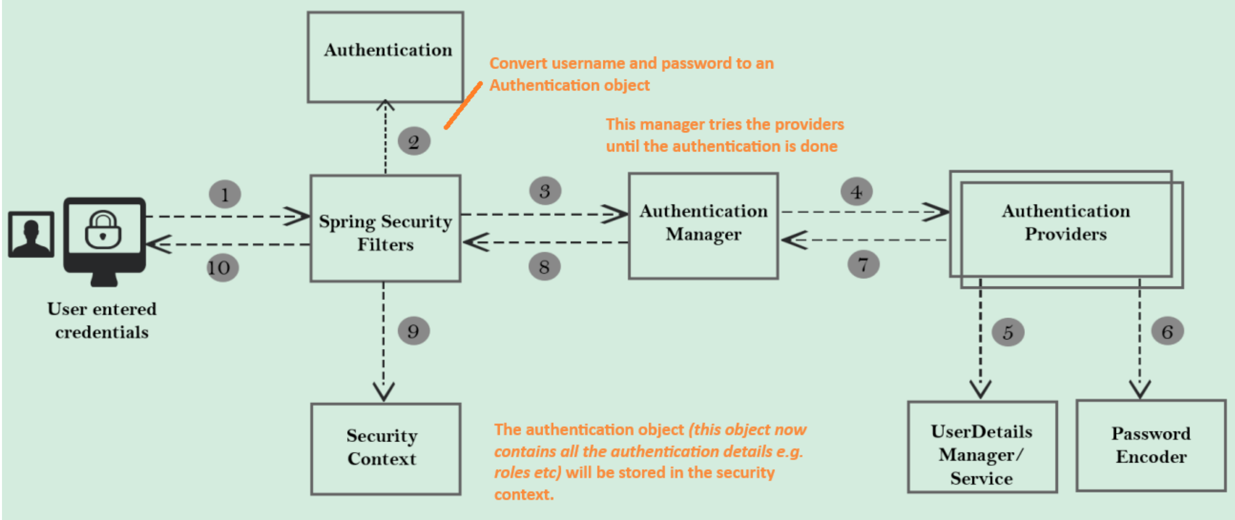
In Java web apps, Servlet Container (Web Server) takes care of translating the HTTP messages for Java code to understand. One of the mostly used servlet container is Apache Tomcat. Servlet Container converts the HTTP messages into ServletRequest and hand over to Servlet method as a parameter. Similarly, ServletResponse returns as an output to Servlet Container from Servlet. So everything we write inside Java web apps are driven by Servlets

★ **Role of Filters**

Filters inside Java web applications can be used to intercept each request/response and do some pre work before our business logic. So using the same filters, Spring Security enforce security based on our configurations inside a web application.

21

Spring Security Internal Flow


Authentication
Convert username and password to an Authentication object
This manager tries the providers until the authentication is done
1
2
3
4
5
6
7
8
9
10
Spring Security Filters
Authentication Manager
Authentication Providers
User entered credentials
Security Context
The authentication object (this object now contains all the authentication details e.g. roles etc) will be stored in the security context.
UserDetails Manager/ Service
Password Encoder

22

Spring Security Internal Flow

★ **Spring Security Filters**
A series of Spring Security filters intercept each request & work together to identify if Authentication is required or not. If authentication is required, accordingly navigate the user to login page or use the existing details stored during initial authentication.

★ **Authentication**
Filters like UsernamePasswordAuthenticationFilter will extract username/password from HTTP request & prepare Authentication type object. Because Authentication is the core standard of storing authenticated user details insdie Spring Security framework.

★ **AuthenticationManager**
Once received request from filter, it delegates the validating of the user details to the authentication providers available. Since there can be multiple providers inside an app, it is the responsibility of the AuthenticationManager to manage all the authentication providers available.

★ **AuthenticationProvider**
AuthenticationProviders has all the core logic of validating user details for authentication.

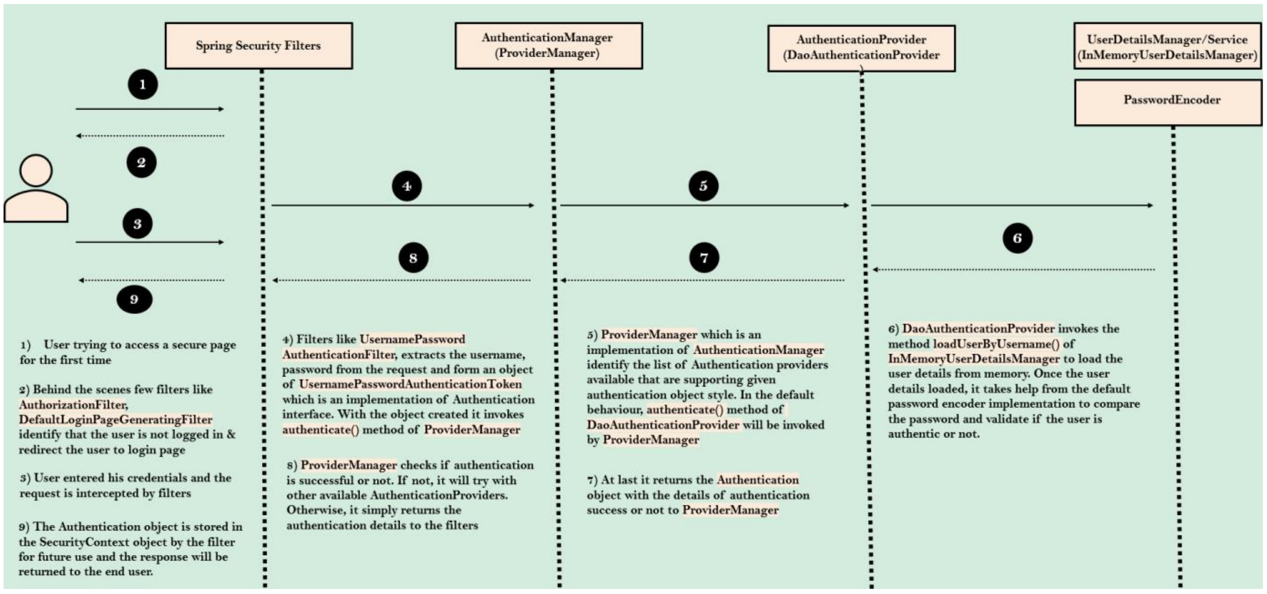
★ **UserDetailsManager/UserDetailsService**
UserDetailsManager/UserDetailsService helps in retrieving, creating, updating, deleting the User Details from the DB/storage systems.

★ **PasswordEncoder**
Service interface that helps in encoding & hashing passwords. Otherwise we may have to live with plain text passwords ☹

★ **SecurityContext**
Once the request has been authenticated, the Authentication will usually be stored in a thread-local SecurityContext managed by the SecurityContextHolder. This helps during the upcoming requests from the same user.

23

Spring Security Internal Flow – Default Behavior


Spring Security Filters
AuthenticationManager (ProviderManager)
AuthenticationProvider (DaoAuthenticationProvider
UserDetailsManager/Service (InMemoryUserDetailsManager)
PasswordEncoder
1
2
3
4
5
6
7
8
9

1) User trying to access a secure page for the first time

2) Behind the scenes few filters like AuthorizationFilter, DefaultLoginPageGeneratingFilter identify that the user is not logged in & redirect the user to login page

3) User entered his credentials and the request is intercepted by filters

9) The Authentication object is stored in the SecurityContext object by the filter for future use and the response will be returned to the end user.

4) Filters like UsernamePassword AuthenticationFilter, extracts the username, password from the request and form an object of UsernamePasswordAuthenticationToken which is an implementation of Authentication interface. With the object created it invokes authenticate() method of ProviderManager

8) ProviderManager checks if authentication is successful or not. If not, it will try with other available AuthenticationProviders. Otherwise, it simply returns the authentication details to the filters

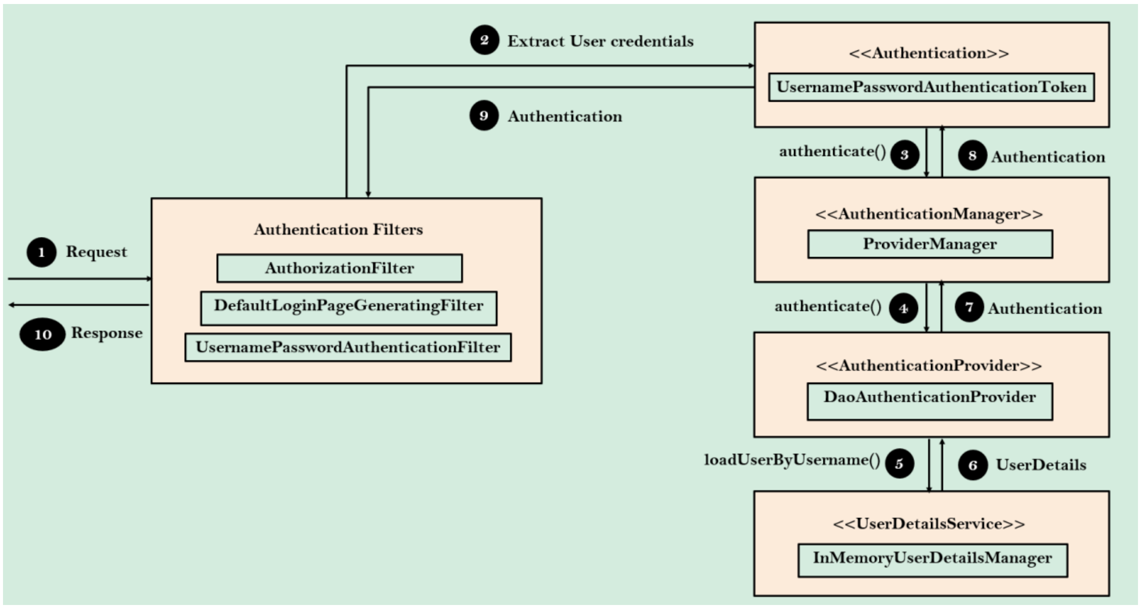
5) ProviderManager which is an implementation of AuthenticationManager identify the list of Authentication providers available that are supporting given authentication object style. In the default behaviour, authenticate() method of DaoAuthenticationProvider will be invoked by ProviderManager

7) At last it returns the Authentication object with the details of authentication success or not to ProviderManager

6) DaoAuthenticationProvider invokes the method loadUserByUsername() of InMemoryUserDetailsManager to load the user details from memory. Once the user details loaded, it takes help from the default password encoder implementation to compare the password and validate if the user is authentic or not.

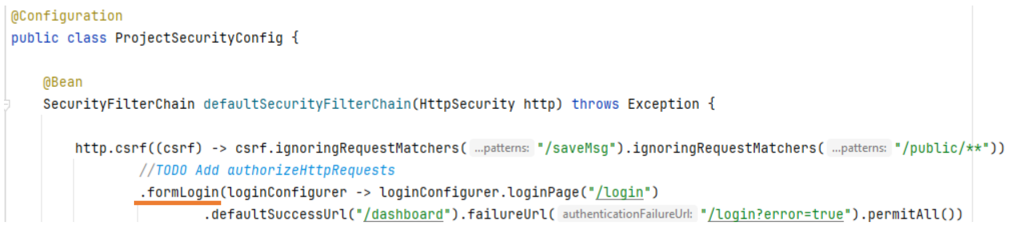
24

Spring Security Internal Flow – Default Behavior


1 Request
10 Response
Authentication Filters
AuthorizationFilter
DefaultLoginPageGeneratingFilter
UsernamePasswordAuthenticationFilter
2 Extract User credentials
9 Authentication
<<Authentication>>
UsernamePasswordAuthenticationToken
authenticate() 3
8 Authentication
<<AuthenticationManager>>
ProviderManager
authenticate() 4
7 Authentication
<<AuthenticationProvider>>
DaoAuthenticationProvider
loadUserByUsername() 5
6 UserDetails
<<UserDetailsService>>
InMemoryUserDetailsManager

25

UsernamePasswordAuthenticationFilter



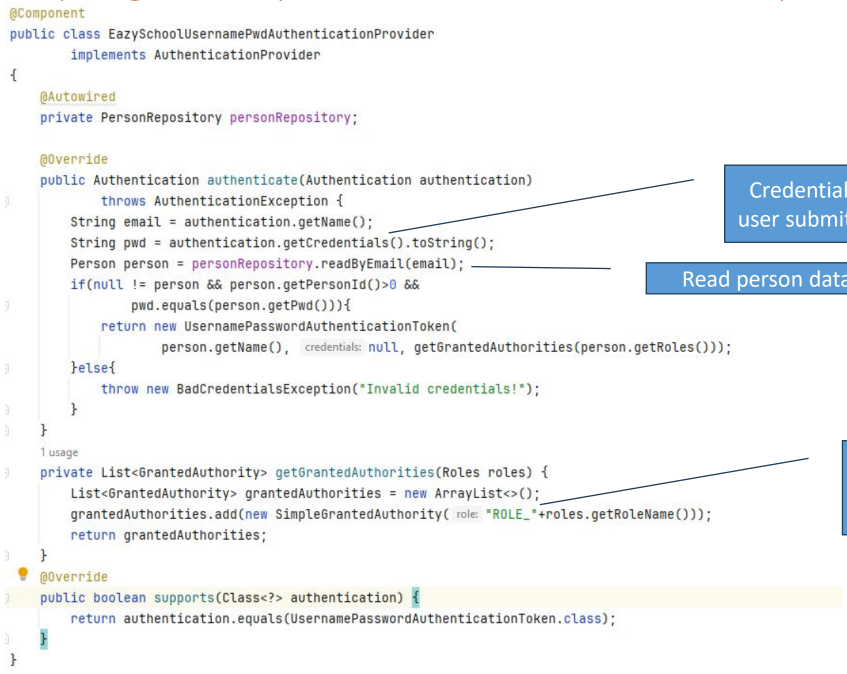
```
@Configuration
public class ProjectSecurityConfig {

    @Bean
    SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {

        http.csrf((csrf) -> csrf.ignoringRequestMatchers( ...patterns: "/saveMsg").ignoringRequestMatchers( ...patterns: "/public/**"))
                //TODO Add authorizeHttpRequests
                .formLogin(loginConfigurer -> loginConfigurer.loginPage("/login")
                        .defaultSuccessUrl("/dashboard").failureUrl( authenticationFailureUrl: "/login?error=true").permitAll())
```

When you configure form-based login using the .formLogin() method in your Spring Security configuration, it effectively instructs Spring Security to use the UsernamePasswordAuthenticationFilter for handling login requests.

The .formLogin() method sets up the UsernamePasswordAuthenticationFilter as the filter responsible for processing login requests. When a user submits a login form, Spring Security's filter chain intercepts the request, and the UsernamePasswordAuthenticationFilter processes it. This filter extracts the username and password from the request parameters, creates an Authentication object, and then delegates the authentication process to the configured AuthenticationProvider.

26

Spring Security AuthenticationProvider - Implementation



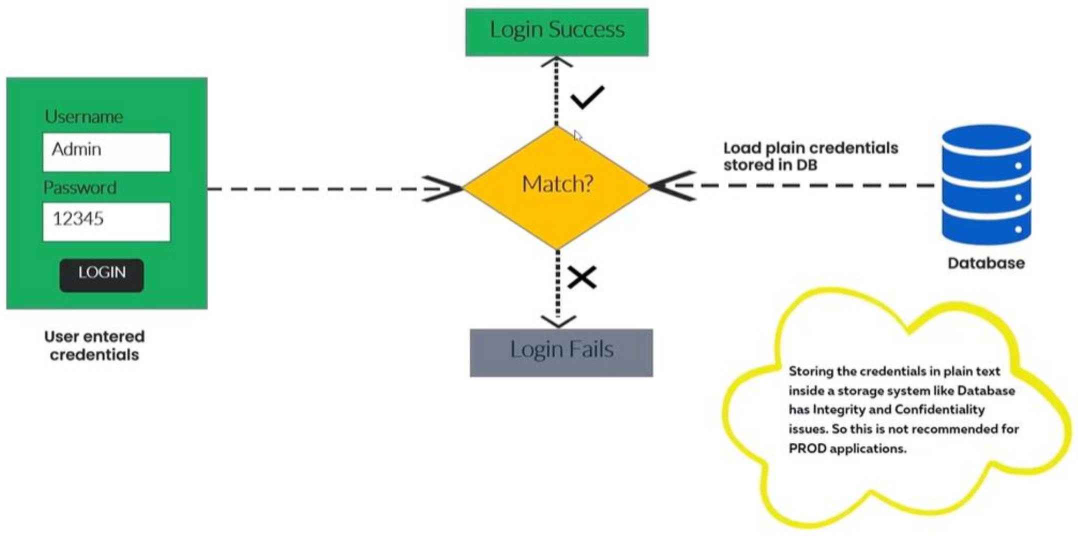
```
@Component
public class EazySchoolUsernamePwdAuthenticationProvider
        implements AuthenticationProvider
{
    @Autowired
    private PersonRepository personRepository;

    @Override
    public Authentication authenticate(Authentication authentication)
            throws AuthenticationException {
        String email = authentication.getName();
        String pwd = authentication.getCredentials().toString();
        Person person = personRepository.readByEmail(email);
        if(null != person && person.getPersonId()>0 &&
                pwd.equals(person.getPwd())){
            return new UsernamePasswordAuthenticationToken(
                    person.getName(),  credentials: null, getGrantedAuthorities(person.getRoles()));
        }else{
            throw new BadCredentialsException("Invalid credentials!");
        }
    }
    1 usage
    private List<GrantedAuthority> getGrantedAuthorities(Roles roles) {
        List<GrantedAuthority> grantedAuthorities = new ArrayList<>();
        grantedAuthorities.add(new SimpleGrantedAuthority( role: "ROLE_"+roles.getRoleName()));
        return grantedAuthorities;
    }
    @Override
    public boolean supports(Class<?> authentication) {
        return authentication.equals(UsernamePasswordAuthenticationToken.class);
    }
}
```

Credentials (email, pwd) that the user submitted from the login form

Read person data form the database

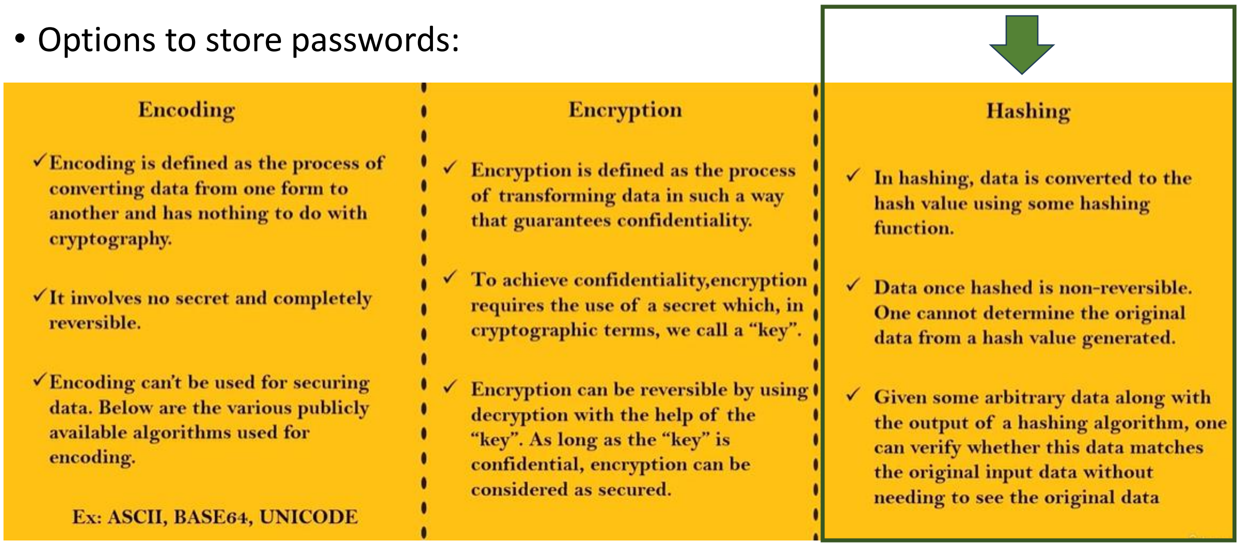
Spring Security will always maintain the roles with the prefix ROLE_

Plain text passwords – NOT secure


Username
Admin
Password
12345
LOGIN
User entered credentials
Login Success
Match?
Login Fails
Load plain credentials stored in DB
Database
Storing the credentials in plain text inside a storage system like Database has Integrity and Confidentiality issues. So this is not recommended for PROD applications.

Password Management

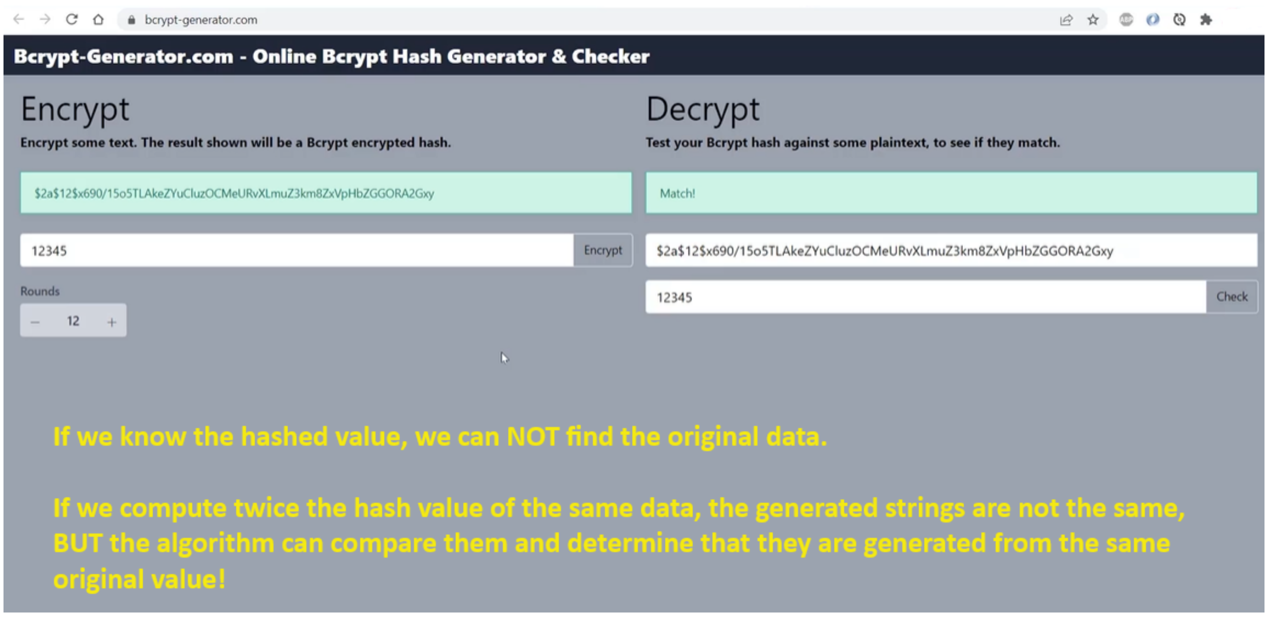
- Options to store passwords:



Encoding	Encryption	Hashing
✓Encoding is defined as the process of converting data from one form to another and has nothing to do with cryptography.	✓ Encryption is defined as the process of transforming data in such a way that guarantees confidentiality.	✓ In hashing, data is converted to the hash value using some hashing function.
✓It involves no secret and completely reversible.	✓ To achieve confidentiality,encryption requires the use of a secret which, in cryptographic terms, we call a "key".	✓ Data once hashed is non-reversible. One cannot determine the original data from a hash value generated.
✓Encoding can't be used for securing data. Below are the various publicly available algorithms used for encoding.	✓ Encryption can be reversible by using decryption with the help of the "key". As long as the "key" is confidential, encryption can be considered as secured.	✓ Given some arbitrary data along with the output of a hashing algorithm, one can verify whether this data matches the original input data without needing to see the original data
Ex: ASCII, BASE64, UNICODE		

29

Hashing



Bcrypt-Generator.com - Online Bcrypt Hash Generator & Checker

Encrypt

Encrypt some text. The result shown will be a Bcrypt encrypted hash.

$2a$12$x690/15o5TLAkeZYuCluzOCMeURvXLmuZ3km8ZxVpHbZGGORA2Gxy

12345 Encrypt

Rounds

– 12 +

Decrypt

Test your Bcrypt hash against some plaintext, to see if they match.

Match!

$2a$12$x690/15o5TLAkeZYuCluzOCMeURvXLmuZ3km8ZxVpHbZGGORA2Gxy

12345 Check

If we know the hashed value, we can NOT find the original data.

If we compute twice the hash value of the same data, the generated strings are not the same, BUT the algorithm can compare them and determine that they are generated from the same original value!

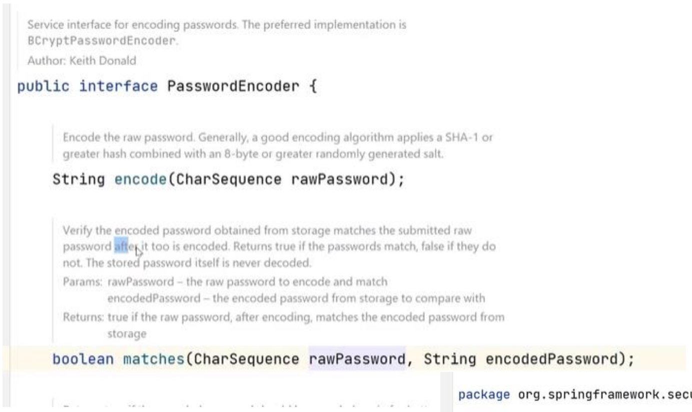
30

Spring Security - Hashing

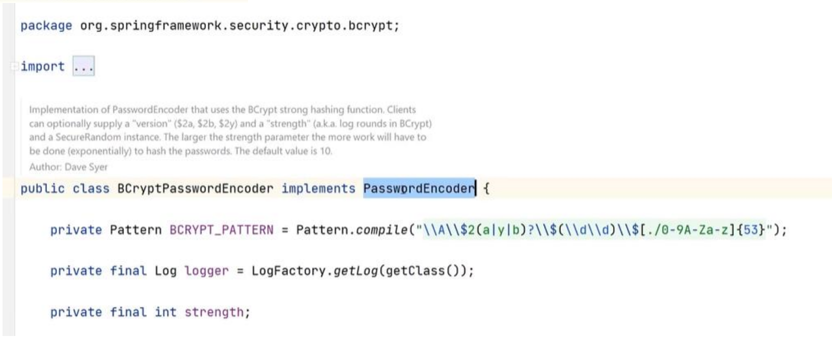
- Spring Security provides support for password hashing through its PasswordEncoder interface and various implementations.
- PasswordEncoder Interface:
  - The PasswordEncoder interface defines a contract for encoding (hashing) and matching (verifying) passwords.
  - It includes two main methods: encode() for hashing a password and matches() for verifying if a raw password matches a hashed password.
- Spring Security provides several implementations of the PasswordEncoder interface, each using a different hashing algorithm:
  - **BCryptPasswordEncoder**: Uses the bcrypt password hashing algorithm, which is considered secure and resistant to brute-force attacks.
  - **StandardPasswordEncoder**: Uses a custom algorithm based on SHA-256 hashing with a salt.
  - **NoOpPasswordEncoder**: No operation password encoder (for testing purposes only; should not be used in production).
  - **MessageDigestPasswordEncoder**: Uses a Message Digest algorithm (e.g., MD5, SHA-1, SHA-256) for password hashing.
  - **DelegatingPasswordEncoder**: A delegating password encoder that supports multiple hash algorithms and can upgrade passwords to a stronger hash over time.
  - …

31

PasswordEncoder and Implementation

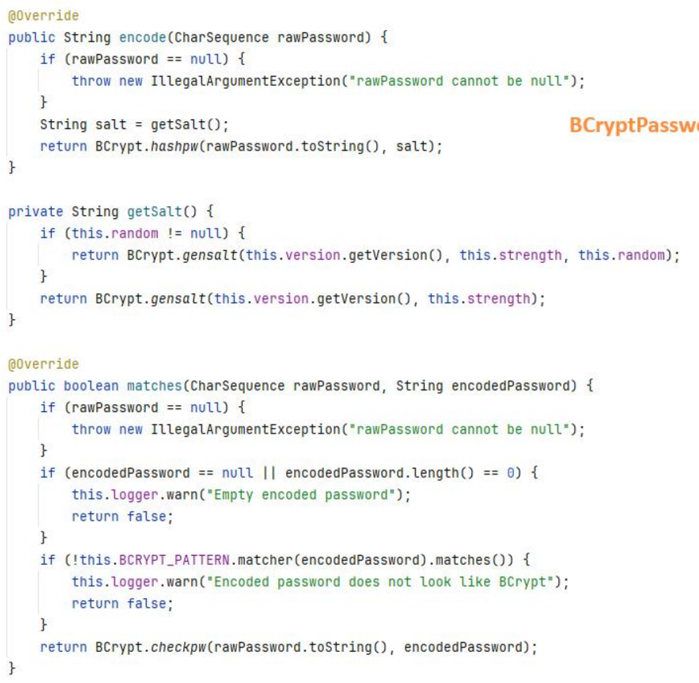


```
Service interface for encoding passwords. The preferred implementation is
BCryptPasswordEncoder.
Author: Keith Donald
public interface PasswordEncoder {

    Encode the raw password. Generally, a good encoding algorithm applies a SHA-1 or
    greater hash combined with an 8-byte or greater randomly generated salt.
    String encode(CharSequence rawPassword);

    Verify the encoded password obtained from storage matches the submitted raw
    password after it too is encoded. Returns true if the passwords match, false if they do
    not. The stored password itself is never decoded.
    Params: rawPassword – the raw password to encode and match
            encodedPassword – the encoded password from storage to compare with
    Returns: true if the raw password, after encoding, matches the encoded password from
            storage
    boolean matches(CharSequence rawPassword, String encodedPassword);
```



```
package org.springframework.security.crypto.bcrypt;

import ...

Implementation of PasswordEncoder that uses the BCrypt strong hashing function. Clients
can optionally supply a "version" ($2a, $2b, $2y) and a "strength" (a.k.a. log rounds in BCrypt)
and a SecureRandom instance. The larger the strength parameter the more work will have to
be done (exponentially) to hash the passwords. The default value is 10.
Author: Dave Syer
public class BCryptPasswordEncoder implements PasswordEncoder {

    private Pattern BCRYPT_PATTERN = Pattern.compile("\\A\\$2(a|y|b)?\\$(\\d\\d)\\$[./0-9A-Za-z]{53}");

    private final Log logger = LogFactory.getLog(getClass());

    private final int strength;
```

32

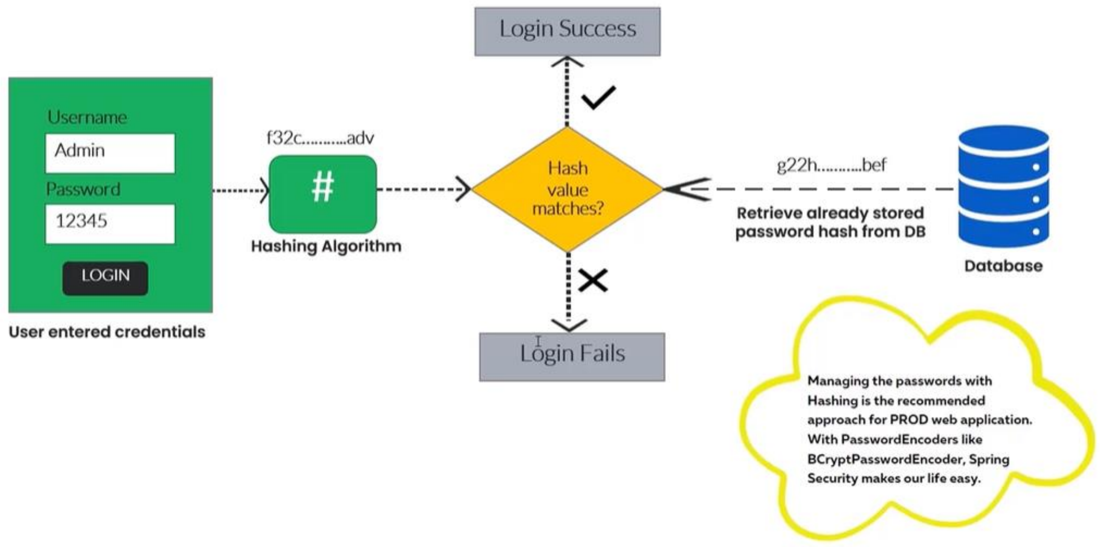


```
@Override
public String encode(CharSequence rawPassword) {
    if (rawPassword == null) {
        throw new IllegalArgumentException("rawPassword cannot be null");
    }
    String salt = getSalt();
    return BCrypt.hashpw(rawPassword.toString(), salt);
}

private String getSalt() {
    if (this.random != null) {
        return BCrypt.gensalt(this.version.getVersion(), this.strength, this.random);
    }
    return BCrypt.gensalt(this.version.getVersion(), this.strength);
}

@Override
public boolean matches(CharSequence rawPassword, String encodedPassword) {
    if (rawPassword == null) {
        throw new IllegalArgumentException("rawPassword cannot be null");
    }
    if (encodedPassword == null || encodedPassword.length() == 0) {
        this.logger.warn("Empty encoded password");
        return false;
    }
    if (!this.BCRYPT_PATTERN.matcher(encodedPassword).matches()) {
        this.logger.warn("Encoded password does not look like BCrypt");
        return false;
    }
    return BCrypt.checkpw(rawPassword.toString(), encodedPassword);
}
```

BCryptPasswordEncoder class

33

How Passwords are Validated with Hashing


Username
Admin
Password
12345
LOGIN
User entered credentials
f32c..........adv
#
Hashing Algorithm
Login Success
Hash value matches?
Login Fails
g22h..........bef
Retrieve already stored password hash from DB
Database
Managing the passwords with Hashing is the recommended approach for PROD web application. With PasswordEncoders like BCryptPasswordEncoder, Spring Security makes our life easy.

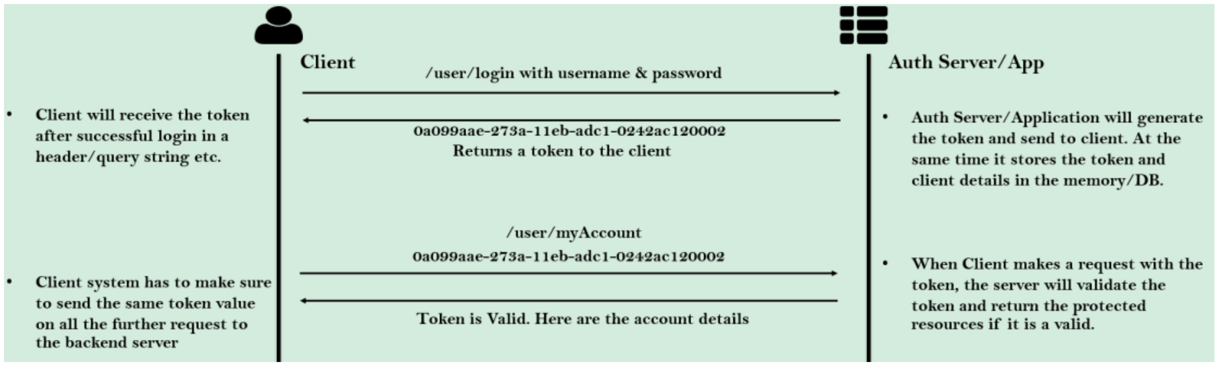
34

JWT Tokens

35

Role of Tokens

- A Token can be a plain string of format universally unique identifier (UUID) or it can be of type JSON (JWT) usually that get generated when the user authenticated for the first time.
  - JSESSIONID is the default token generated by Spring Security. It makes it possible not to share the credentials to the backend every time. The JSESSIONID is very simple. Disadvantages:
    - It doesn't have any user data (it is a randomly generated value).
    - It is saved as a cookie.
- In every request to a restricted resource, the client sends the access token in the query string or Authorization header. The server then validates the token and, if it is valid, returns the secure resource to the client.


Client
Auth Server/App
/user/login with username & password
0a099aae-273a-11eb-adc1-0242ac120002
Returns a token to the client
/user/myAccount
0a099aae-273a-11eb-adc1-0242ac120002
Token is Valid. Here are the account details

- Client will receive the token after successful login in a header/query string etc.
- Client system has to make sure to send the same token value on all the further request to the backend server
- Auth Server/Application will generate the token and send to client. At the same time it stores the token and client details in the memory/DB.
- When Client makes a request with the token, the server will validate the token and return the protected resources if it is a valid.

36

Advantages of Tokens

Token helps us not to share the credentials for every request. It is a security risk to send credentials over the network frequently.

Tokens can be invalidated during any suspicious activities without invalidating the user credentials.

Tokens can be created with a short life span.

Tokens can be used to store the user related information like roles/authorities etc.

Reusability - We can have many separate servers, running on multiple platforms and domains, reusing the same token for authenticating the user.

Stateless, easier to scale. The token contains all the information to identify the user, eliminating the need for the session state. If we use a load balancer, we can pass the user to any server, instead of being bound to the same server we logged in on.

37

JWT Tokens

JWT means JSON Web Token. It is a token implementation which will be in the JSON format and designed to use for the web requests.

JWT is the most common and favorite token type that many systems use these days due to its special features and advantages.

JWT tokens can be used both in the scenarios of Authorization/Authentication along with Information exchange which means you can share certain user related data in the token itself which will reduce the burden of maintaining such details in the sessions on the server side.

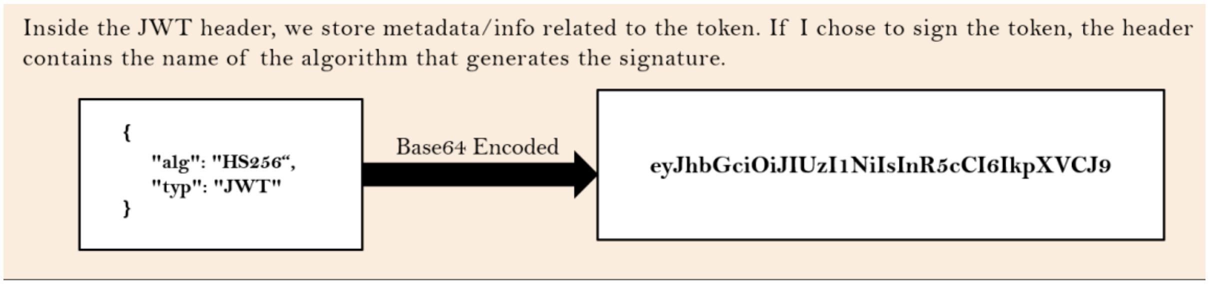
A JWT token has 3 parts each separated by a period(.) Below is a sample JWT token,

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

1. Header
2. Payload
3. Signature (Optional)

38

JWT Tokens

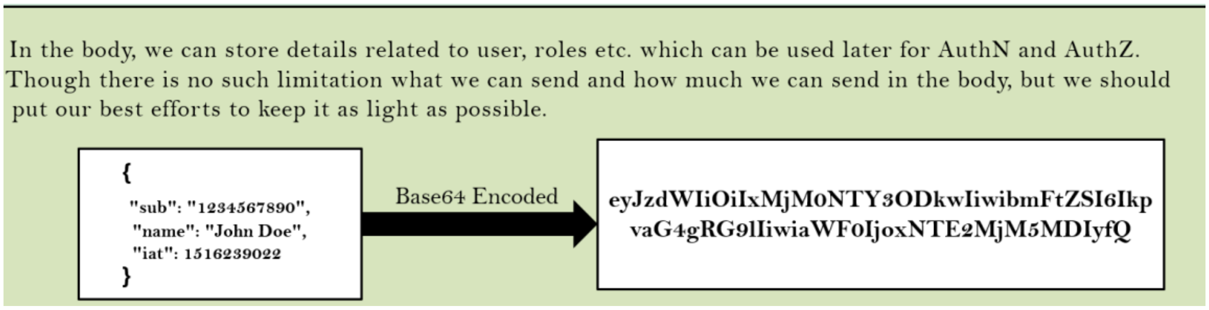


Inside the JWT header, we store metadata/info related to the token. If I chose to sign the token, the header contains the name of the algorithm that generates the signature.

```
{
    "alg": "HS256“,
    "typ": "JWT"
}
```

Base64 Encoded

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9



In the body, we can store details related to user, roles etc. which can be used later for AuthN and AuthZ. Though there is no such limitation what we can send and how much we can send in the body, but we should put our best efforts to keep it as light as possible.

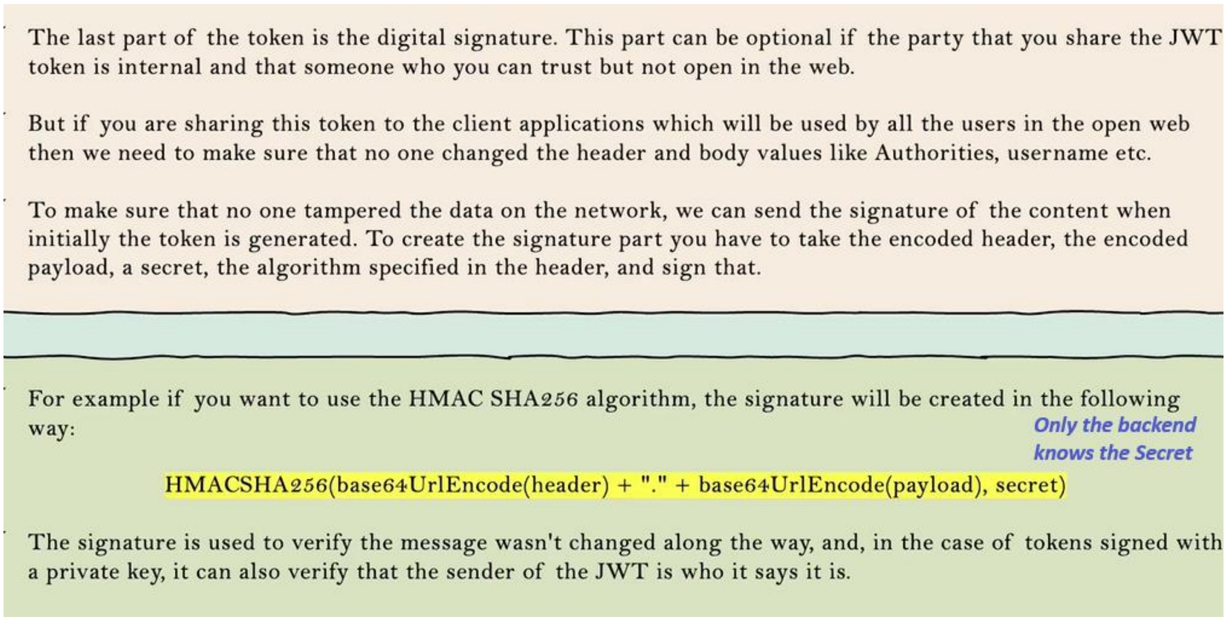
```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

Base64 Encoded

eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikp
vaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ

39

JWT Tokens - Signature



The last part of the token is the digital signature. This part can be optional if the party that you share the JWT token is internal and that someone who you can trust but not open in the web.

But if you are sharing this token to the client applications which will be used by all the users in the open web then we need to make sure that no one changed the header and body values like Authorities, username etc.

To make sure that no one tampered the data on the network, we can send the signature of the content when initially the token is generated. To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.

For example if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way:

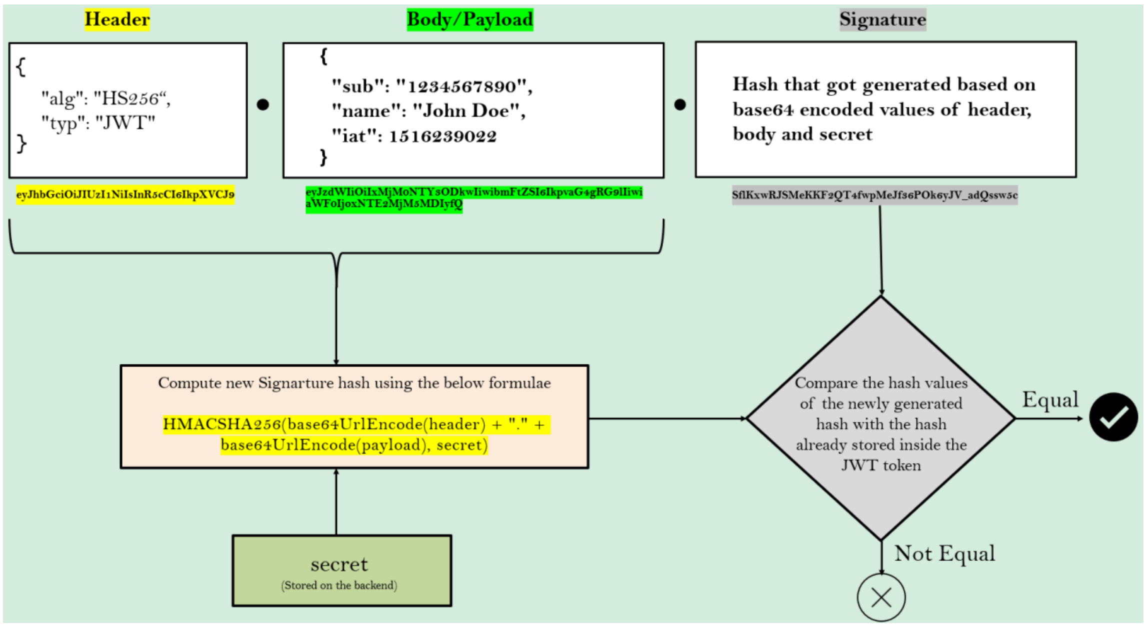
Only the backend knows the Secret

HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)

The signature is used to verify the message wasn't changed along the way, and, in the case of tokens signed with a private key, it can also verify that the sender of the JWT is who it says it is.

40

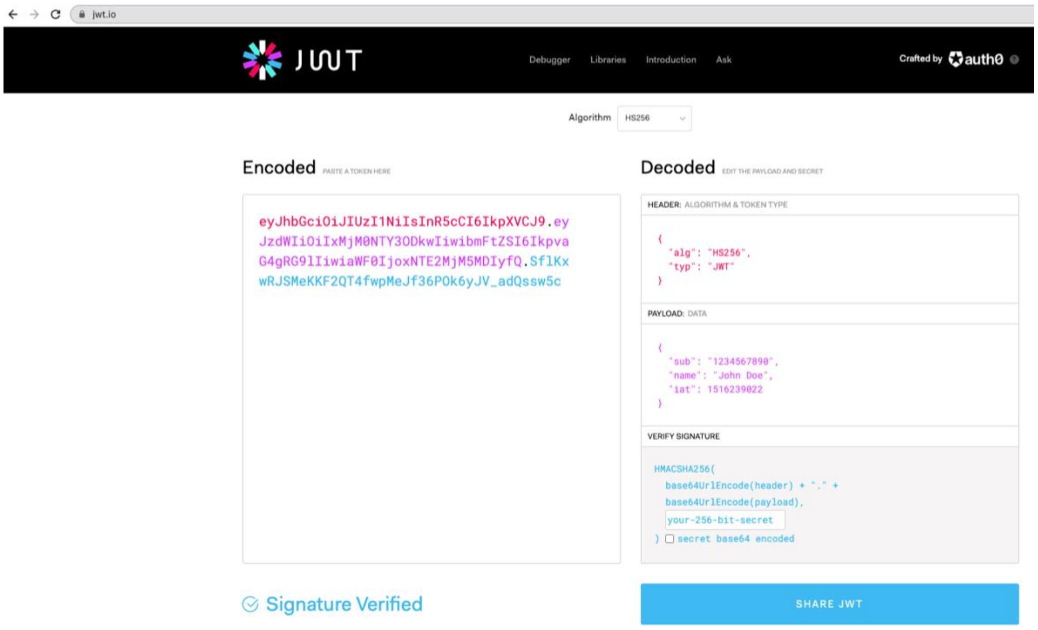
Validation of JWT Tokens


Header
Body/Payload
Signature
{ "alg": "HS256", "typ": "JWT" }
{ "sub": "1234567890", "name": "John Doe", "iat": 1516239022 }
Hash that got generated based on base64 encoded values of header, body and secret
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ
SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
Compute new Signarture hash using the below formulae
HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)
secret
(Stored on the backend)
Compare the hash values of the newly generated hash with the hash already stored inside the JWT token
Equal
Not Equal

41

JWT Tokens

- If you want to try out JWT, you can use jwt.io debugger to decode, verify and generate JWTs.



```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYLOAD: DATA

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

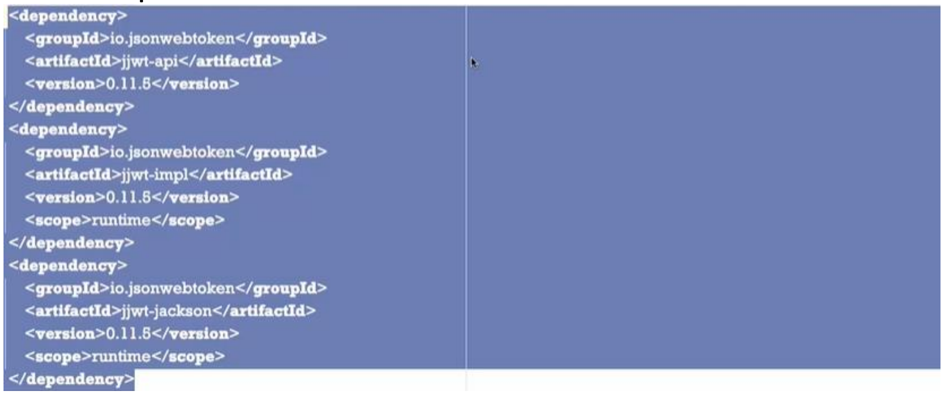
VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) ☐ secret base64 encoded
```

42

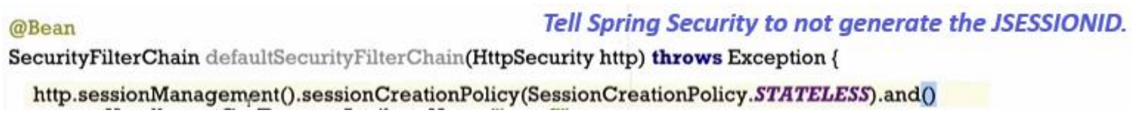
Use JWT tokens

1. Add dependencies:



```
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-api</artifactId>
  <version>0.11.5</version>
</dependency>
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-impl</artifactId>
  <version>0.11.5</version>
  <scope>runtime</scope>
</dependency>
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-jackson</artifactId>
  <version>0.11.5</version>
  <scope>runtime</scope>
</dependency>
```

2. Change session creation policy:

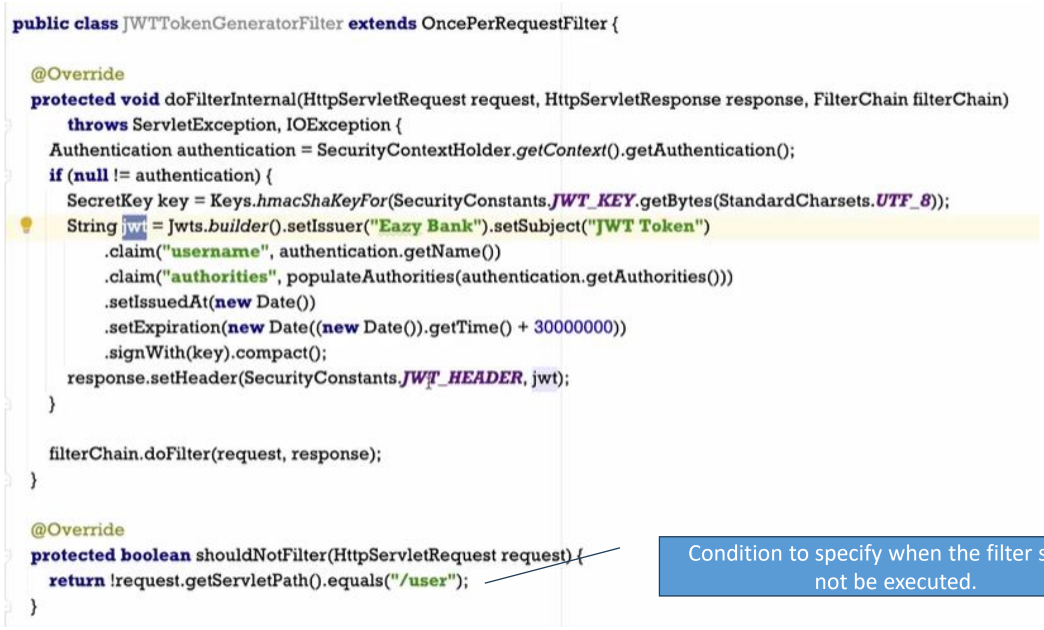


Tell Spring Security to not generate the JSESSIONID.

```
@Bean
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
  http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS).and()
```

43

Create a new Filter to generate the JWT Token

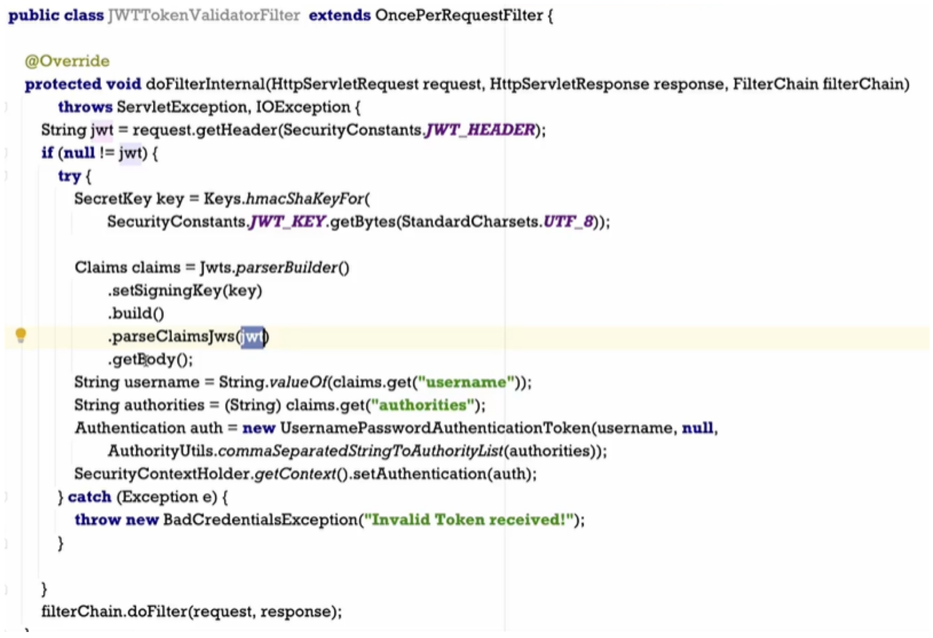


```
public class JWTTokenGeneratorFilter extends OncePerRequestFilter {

  @Override
  protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
      throws ServletException, IOException {
    Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
    if (null != authentication) {
      SecretKey key = Keys.hmacShaKeyFor(SecurityConstants.JWT_KEY.getBytes(StandardCharsets.UTF_8));
      String jwt = Jwts.builder().setIssuer("Eazy Bank").setSubject("JWT Token")
          .claim("username", authentication.getName())
          .claim("authorities", populateAuthorities(authentication.getAuthorities()))
          .setIssuedAt(new Date())
          .setExpiration(new Date((new Date()).getTime() + 30000000))
          .signWith(key).compact();
      response.setHeader(SecurityConstants.JWT_HEADER, jwt);
    }

    filterChain.doFilter(request, response);
  }

  @Override
  protected boolean shouldNotFilter(HttpServletRequest request) {
    return !request.getServletPath().equals("/user");
  }
```

Condition to specify when the filter should not be executed.

44

Create a new Filter to validate JWT tokens



```
public class JWTTokenValidatorFilter  extends OncePerRequestFilter {

  @Override
  protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
        throws ServletException, IOException {
    String jwt = request.getHeader(SecurityConstants.JWT_HEADER);
    if (null != jwt) {
      try {
        SecretKey key = Keys.hmacShaKeyFor(
            SecurityConstants.JWT_KEY.getBytes(StandardCharsets.UTF_8));

        Claims claims = Jwts.parserBuilder()
            .setSigningKey(key)
            .build()
            .parseClaimsJws(jwt)
            .getBody();
        String username = String.valueOf(claims.get("username"));
        String authorities = (String) claims.get("authorities");
        Authentication auth = new UsernamePasswordAuthenticationToken(username, null,
            AuthorityUtils.commaSeparatedStringToAuthorityList(authorities));
        SecurityContextHolder.getContext().setAuthentication(auth);
      } catch (Exception e) {
        throw new BadCredentialsException("Invalid Token received!");
      }

    }
    filterChain.doFilter(request, response);
```

45