Day14: Spring Security: Using JWT, Cors, Filters

Cors:(Cross-origin resource sharing)

CORS is not a security issue/attack but the default protection provided by the browsers to stop sharing the data/communication between **different origins**.

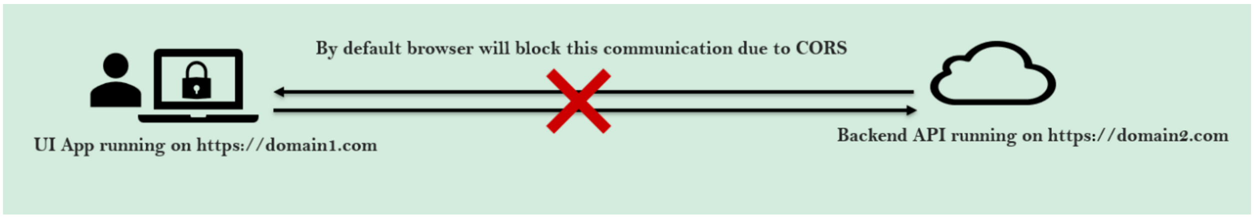
here "**different origins**" means the URL being accessed differs from the location that the Java script is running from, by having :

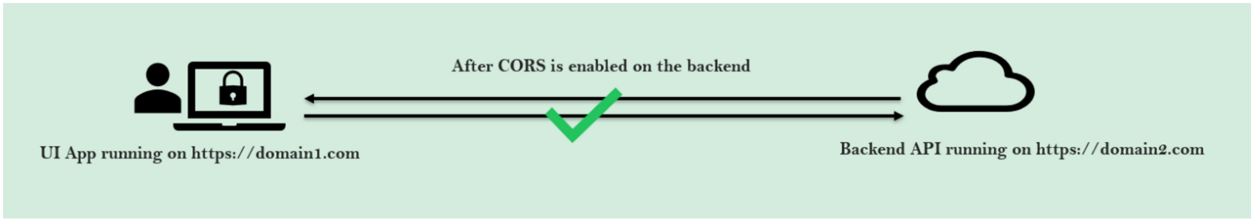
- Different scheme(Http or Https)
- Different domain
- Different port

The combination of these 3 is called an "**origin**".

- If two different **origin** trying to communicate each other. by default will be stopped by the browser. because of the security features provided by the modern browsers.
- Cors is a security layer which protects from the security attacks.

However, there are legitimate scenarios where cross-origin access is desirable or even necessary. For example, if a UI App wishes to make a call to API call on a different domain, it would be blocked by doing so by default due to CORS.


By default browser will block this communication due to CORS
UI App running on https://domain1.com
Backend API running on https://domain2.com


After CORS is enabled on the backend
UI App running on https://domain1.com
Backend API running on https://domain2.com

Possible options to fix the Cors issue:

1. Apply the following annotation on the top of All the controllers which API we want to expose for the UI(Java script AJAX call)

```
@CrossOrigin(origins = "http://localhost:4200")

@CrocssOrigin(origins="*") // will allows any domain
```

If we have 100 controllers then it will be very tedious job to apply @CrossOrign to all the Controllers. so we can centralize these Cross origin globally to all the controller by configuring the SecurityFilterChain bean.

Example:

```
@Bean
  public SecurityFilterChain springSecurityConfiguration(HttpSecurity http) throws Exception {

    http.cors(cors -> {

      cors.configurationSource(new CorsConfigurationSource() {

        @Override
        public CorsConfiguration getCorsConfiguration(HttpServletRequest request) {
          CorsConfiguration cfg = new CorsConfiguration();

          cfg.setAllowedOriginPatterns(Collections.singletonList("*"));
          cfg.setAllowedMethods(Collections.singletonList("*"));
          cfg.setAllowCredentials(true);
          cfg.setAllowedHeaders(Collections.singletonList("*"));
          cfg.setExposedHeaders(Arrays.asList("Authorization"));
          return cfg;
        }
      });

    }).authorizeHttpRequests(auth ->{

    auth
    .requestMatchers(HttpMethod.POST,"/customers").permitAll()
    .requestMatchers(HttpMethod.GET, "/customers","/hello").hasRole("ADMIN")
    .requestMatchers(HttpMethod.GET, "/customers/**").hasAnyRole("ADMIN","USER")
    .requestMatchers("/swagger-ui*/**","/v3/api-docs/**").permitAll()
    .anyRequest().authenticated();

  })
  .csrf(csrf -> csrf.disable())
  .formLogin(Customizer.withDefaults())
  .httpBasic(Customizer.withDefaults());


    return http.build();


  }
```

cfg.setAllowedOriginPatterns(Collections.singletonList("*"));

- We can allow all the origins

cfg.setAllowedMethods(Collections.singletonList("*"));

- We can allows all the http methods like GET, POST, etc.

cfg.setAllowCredentials(true);

- We are fine with passing credentials TO and FRO from this application.

cfg.setMaxAge(3600L);

- Browser can remember this configuration upto1 hour, browser will cache these details up to 1 hour.

How the browser understand the configuration we have done in backed application?

Usually browser makes a preflight request. whenever we try to access any resource or call any API from the different domain, first browser will not make the real request to our application, first it will make a pre-flight request to the backed server, this different domain tries to make a request, are u fine with that. if the backend application is Ok with this origin, then it allows the actual request.

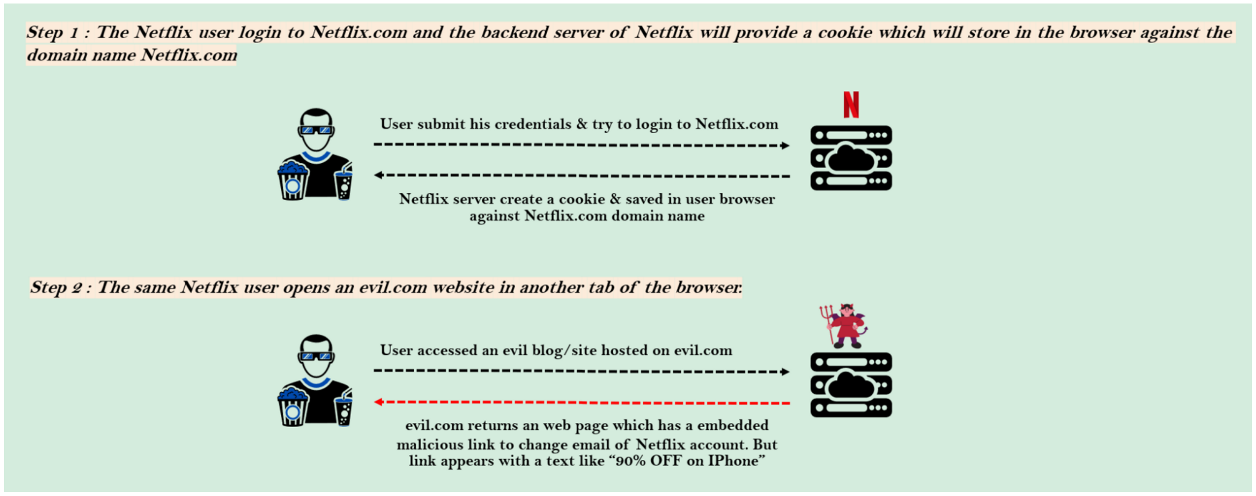
CSRF: (Cross Site request forgery)

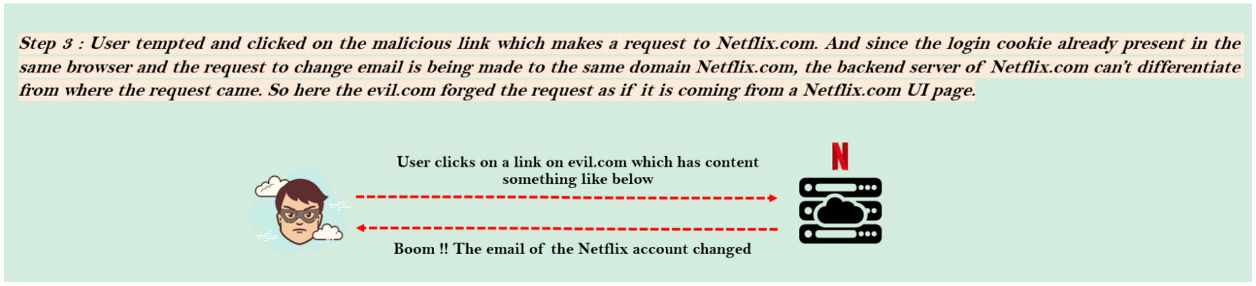
A typical CSRF or XSRF attack aims to perform an operation in a web app on behalf of user without their explicit consent.

In general, it does not directly steal the user's identity, but it exploits the user to carry out an action without their will.

It is one of the most severe vulnerabilities which can be exploited in various ways- from changing a user's info without his knowledge to gaining full access to the user's account.

CSRF attack:


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
User submit his credentials & try to login to Netflix.com
Netflix server create a cookie & saved in user browser against Netflix.com domain name
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"


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
Boom !! The email of the Netflix account changed

```
<form action="https://netflix.com/changeEmail" method="post" id="f">

<input type="hidden" name="email", value="user@evil.com"/>

</form>

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

Along with the above request, browser will also send the cookie details to the netflix.com.
If our netflix.com don't handle the CSRF attack properly they can not differentiate the request is coming from their proper website or from any other website.

By Default Spring Security gives protection against CSRF attacks by not allowing POST and PUT requests to execute and you will get an HTTP 403 error code.

So here, we are disabling CSRF so that we can POST a request. **(Not a recommended approach)**

```
.csrf(csrf -> csrf.disable())
```

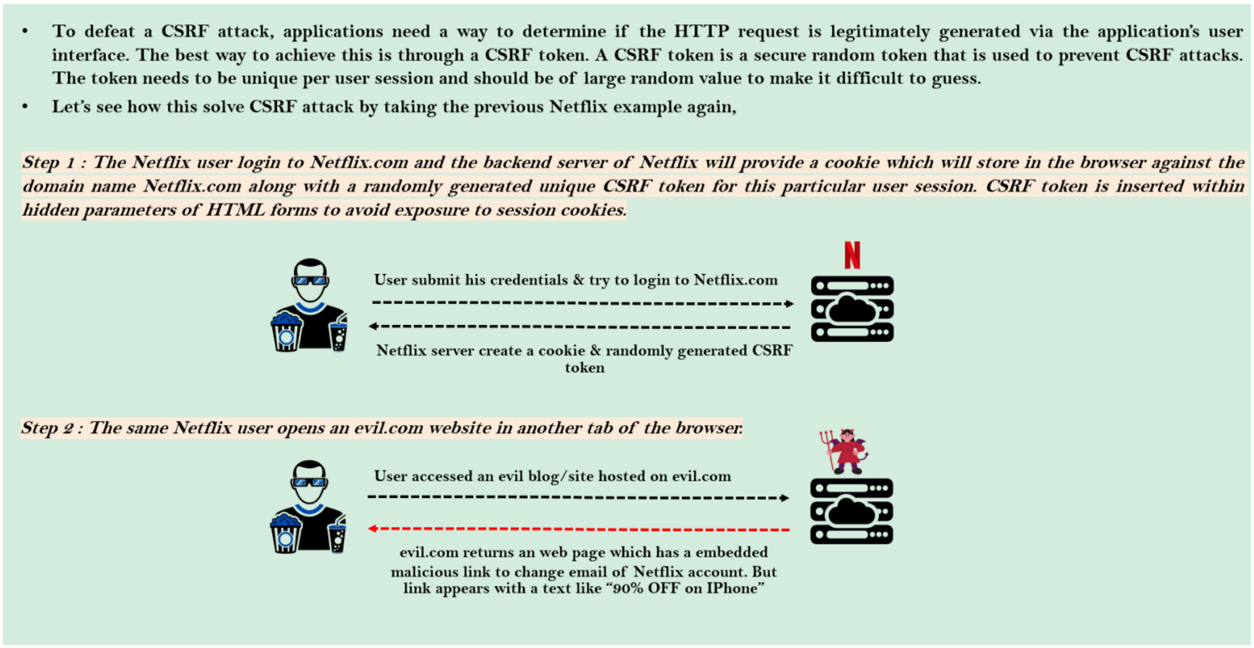
Solution to handle the CSRF attack:

To defeat a CSRF attack, application need a way to determine if the http request is legitimatly generated via the application's user interface.

The best way to achieve this is through a CSRF token.

A CSRF token is a secure random token that is used to prevent CSRF attacks.

The token need to be unique per user session and should be in a large random value to make it difficult to guess.



- **To defeat a CSRF attack, applications need a way to determine if the HTTP request is legitimately generated via the application's user interface. The best way to achieve this is through a CSRF token. A 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.**
- **Let's see how this solve CSRF attack by taking the previous Netflix example again,**

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.

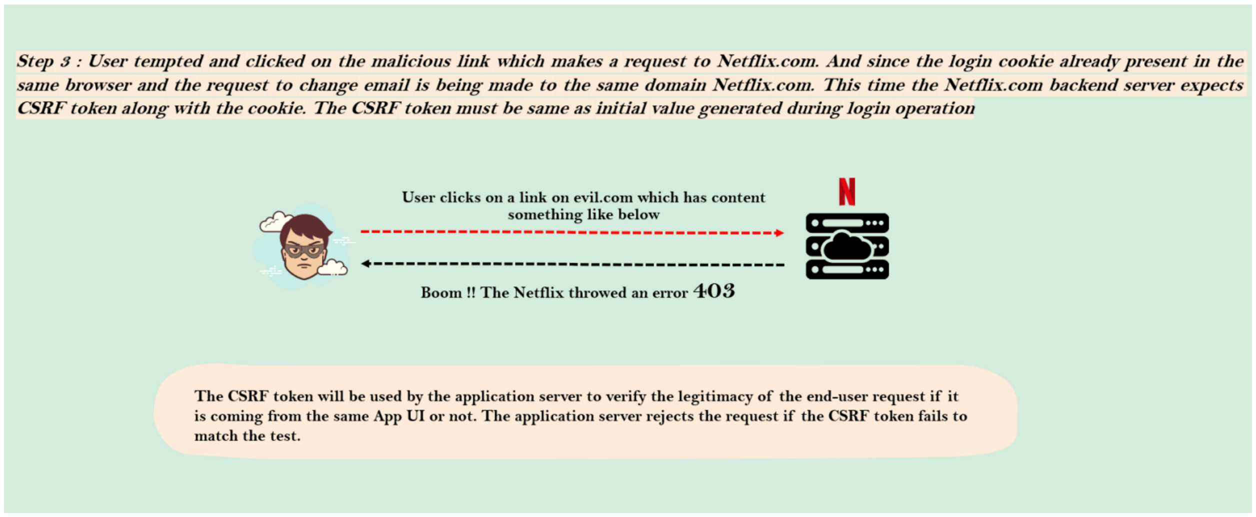
User submit his credentials & try to login to Netflix.com

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"


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.

Step1: Netflix user login and server will send a CSRF token along with the cookie for this particular user session.

Now our browser will gets the 2 cookies:

1. authentication related cookie
2. CSRF related cookie.

Step2: the same Netflix user opens an evil.com website in another tab of the browser.

Now evil.com makes the request to the netflix.com, here also both cookie will go to the Netflix server. but this time netflix.com will deny the request.
because:

There is trick we need to follow inside our Client application or UI application, so whenever we receive the CSRF token initially from the backend application, we are going to get the CSRF token also inside a cookie but instead of using the default style of sending the cookie to the backed app with the help of browser, we need to write a small piece of code inside our UI application or client application to manually read that cookie and send that cookie inside the header or inside the body payload, where ever we have the agreement with the backend server.

And since we are reading the cookie from the same domain which is netflix.com we should be able to read the cookie from our Java-script code or from our html code. where hacker can not do it from the evil.com. because even he will try to execute some Java-script code inside the evil.com he will running that website from the evil.com, that domain will not have the access of that CSRF token

So the hacker can not send the CSRF token to the header or body payload, he will always relying on the automatic style of sending the cookie to the backend.

This time our backend application will be very smart it will not going to process the request as long as CSRF token is coming from the header or body payload based on the agreement.

Ignoring the CSRF protection for public APIs:

For some kind of POST and PUT request if they don't harm our application we need not manage the CSRF. because there is no sensitive information or their is no vulnerability attack which a hacker can do with those public APIs.

Create two POST endpoint inside our previous CustomerController class:

```
@PostMapping("/contact")
public String postDemo1() {
  return "Not harmfull POST operation";
}

@PutMapping("/notice")
public String putDemo1() {
  return "Not harmfull PUT operation";
}
```

So we can tell Spring Security to some public API to not apply CSRF protection by default.

```
.csrf(csrf -> csrf.ignoringRequestMatchers("/notice","/contact"))
```

Now let’s implement proper CSRF token solution inside our application:

To tell the Spring Security framework to generate the CSRF token and send that token to my client application.

```
.csrf(csrf -> csrf.ignoringRequestMatchers("/notice","/contact")
                  .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse()))
```

The CookieCsrfTokenRepository class is responsible to persist the CSRF token in a cookie named **“XSRF-TOKEN”** and reads from the client application request header **“X-XSRF-TOKEN”**;

With the above configuration, now the framework is capable of generating a CSRF token capable of storing it inside the memory of the application and it is also capable of accepting the header and cookie name values from the UI/Client application. And it will validate the same.

But now our backend application, first it has to send the cookie and header value to the UI/Client application after the initial login. Otherwise, how our client application will know what is the CSRF Token value?

That's why our Spring Security framework also should send the token value that generated as part of the response to the UI/client application.

So in order to send the CSRF token for each and every response we are going to send to the UI application, we need to create a filter class.

CsrfCookieFilter.java

```
package com.masai.config;

import java.io.IOException;

import org.springframework.security.web.csrf.CsrfToken;
import org.springframework.web.filter.OncePerRequestFilter;

import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

public class CsrfCookieFilter extends OncePerRequestFilter{

  @Override
  protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
      throws ServletException, IOException {


    CsrfToken csrfToken= (CsrfToken)request.getAttribute(CsrfToken.class.getName());

    if(csrfToken.getHeaderName() != null) {
      response.setHeader(csrfToken.getHeaderName(), csrfToken.getToken());
    }

    filterChain.doFilter(request, response);

  }

}
```

This way, eventually, whenever we are sending the response to the UI application, our csrfToken value will be present inside the header.

And here we may have a question like, "How about the cookie?" We are just sending the header, but not the cookie.

That's where the beauty of Spring Security come into picture. As a developer, when we populate the CsrfToken value as part of response header, Spring Security framework is going to take care

of generating the CSRF cookie and sending the same to the browser or UI/Client application as part of the response.

So now we have created an filter but we need to configure this filter inside the Security framework.

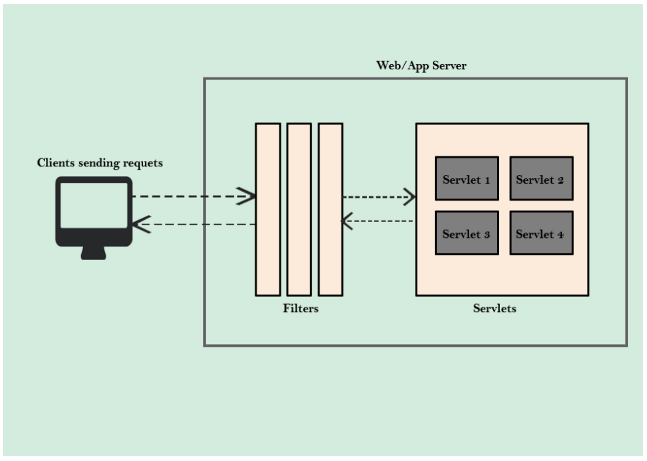
```
.addFilterAfter(new CsrfCookieFilter(), BasicAuthenticationFilter.class)
```

Now, after the initial login, as a response we will get 2 extra information:

1. XSRF-TOKEN as an extra cookie
2. X-XSRF-TOKEN as a response header

To call any protected POST or PUT request we need to send the same XSRF-TOKEN as a request cookie and X-XSRF-TOKEN as a request header.

Filters In Spring Security:



Filters inside java web applications can be used to intercept each request/response and do some pre-processing or post-processing for each client request. using the same filters, Spring Security enforce security based on our configurations inside a Spring boot application.

We can always check the registered filters inside Spring Security by using following configurations:

- Apply **@EnableWebSecurity(debug = true)** annotation on the top of the main class

- Enable the following logging details inside the **application.properties** file.

  ```
  logging.level.org.springframework.security.web.FilterChainProxy=DEBUG
  ```

Some of the internal filters that are executed by spring security in the authentication flow

```
Security filter chain: [
  DisableEncodeUrlFilter
  WebAsyncManagerIntegrationFilter
  SecurityContextHolderFilter
  HeaderWriterFilter
  LogoutFilter
  UsernamePasswordAuthenticationFilter
  DefaultLoginPageGeneratingFilter
  DefaultLogoutPageGeneratingFilter
  BasicAuthenticationFilter
  RequestCacheAwareFilter
  SecurityContextHolderAwareRequestFilter
  AnonymousAuthenticationFilter
  ExceptionTranslationFilter
  AuthorizationFilter
]
```

- A filter is a component which receives the request, process its logic and handover to the next filter in the chain. Spring Security is based on a chain of Servlet filters.
- Each filters has a specific responsibility and depending on the configuration, filters can be added or removed also.
- We can add our custom filters as well based on the requirement.

Implementing the Custom filters:

- We can create our own filters by implementing the **Filter** interface from **jakarta.servlet** package. post that we need to override **doFilter(-,-,-)** method to have our own custom logic, this method having 3 parameters:
  1. **ServletReqest:** It represents the Http request, we use this object to retrieve details about the request from the client(like header information, etc.)
  2. **ServletResponse:** It represents the Http response, we use this object to modify the response before sending it back to the client or further along the filter chain.
  3. **FilterChain:** It represents the collection of filters with a defined order in which they act. we use this FilterChain object to forward the request to the next filter in the chain.

Note: We can add a new filter to the Spring Security chain either **Before**, **After** or **at the position** of a known one.

Following methods are used to configure a custom filter in the Spring Security flow:

```
addFilterBefore(filter, class): //add a filter before the position of the specified filter class.

addFilterAfter(filter, class): //add a filter after the position of the specified filter class.

addFilterAt(filter, class): //add a filter at the location of the specified filter class.
```

Adding a custom filter using addFilterBefore:

Example: Let’s create a custom RequestValidationFilter and configure it before the BasicAuthenticationFilter of the Spring Security flow:

Requirement: request header should contain key as **Allow** and value should not be **test**

RequestValidationFilter.java:

```
package com.masai.config;

import java.io.IOException;

import org.springframework.security.authentication.BadCredentialsException;

import jakarta.servlet.Filter;
import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.ServletRequest;
import jakarta.servlet.ServletResponse;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

public class RequestValidationFilter implements Filter {

  @Override
  public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
      throws IOException, ServletException {

    HttpServletRequest req = (HttpServletRequest) request;
    HttpServletResponse res = (HttpServletResponse) response;

    String header = req.getHeader("Allow");

    if (header == null || header.equals("test")) {
      res.setStatus(HttpServletResponse.SC_BAD_REQUEST);
    throw new BadCredentialsException("header should contain key as Allow and value should not be test");
    }

    chain.doFilter(request, response);

  }

}
```

Now we need to apply this custom filter inside our Spring Security configuration flow:

AppConfig.java:

```
package com.masai.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpMethod;
import org.springframework.security.config.Customizer;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.security.web.authentication.www.BasicAuthenticationFilter;

@Configuration
public class AppConfig {


  @Bean
  public SecurityFilterChain springSecurityConfiguration(HttpSecurity http) throws Exception {

    http.authorizeHttpRequests(auth ->{

      auth
      .requestMatchers(HttpMethod.POST,"/customers").permitAll()
    .requestMatchers(HttpMethod.GET, "/customers","/hello").hasRole("ADMIN")
    .requestMatchers(HttpMethod.GET, "/customers/**").hasAnyRole("ADMIN","USER")
```

```
      .anyRequest().authenticated();

    }).addFilterBefore(new RequestValidationFilter(), BasicAuthenticationFilter.class)
    .csrf(csrf -> csrf.disable())
    .formLogin(Customizer.withDefaults())
    .httpBasic(Customizer.withDefaults());

    return http.build();

  }

  @Bean
  public PasswordEncoder passwordEncoder() {

    return new BCryptPasswordEncoder();

  }
}
```

Here from the UI application we should catch this 400 error and show it to the end user inside our UI application.

Note: We should not write any real or big business logic which involves database inside the Filters, because it will be called for all the request and will cause the performance problem.

Adding a custom filter using addFilterAfter:

If we want to execute some of the business logic just after our authentication gets completed.

- here we add a filter just after the authentication to write a logger about successful authentication and authorities details of logged in user.

AuthoritiesLogginAfterFilter.java

```
package com.masai.config;

import java.io.IOException;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.context.SecurityContextHolder;

import jakarta.servlet.Filter;
import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.ServletRequest;
import jakarta.servlet.ServletResponse;

public class AuthoritiesLogginAfterFilter implements Filter{


  private final Logger log = LoggerFactory.getLogger(AuthoritiesLogginAfterFilter.class);

  @Override
  public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
      throws IOException, ServletException {




    Authentication auth= SecurityContextHolder.getContext().getAuthentication();


    if(auth != null) {

      log.info("User "+auth.getName()+" is successfully authenticated and has Authorties "+ auth.getAuthorities().toString());
    }

    chain.doFilter(request, response);
  }

}
```

Configuring the above filter inside the Spring Security configuration flow:

```
@Bean
  public SecurityFilterChain springSecurityConfiguration(HttpSecurity http) throws Exception {

    http.authorizeHttpRequests(auth ->{

      auth
      .requestMatchers(HttpMethod.POST,"/customers").permitAll()
    .requestMatchers(HttpMethod.GET, "/customers","/hello").hasRole("ADMIN")
    .requestMatchers(HttpMethod.GET, "/customers/**").hasAnyRole("ADMIN","USER")
      .anyRequest().authenticated();

    }).addFilterBefore(new RequestValidationFilter(), BasicAuthenticationFilter.class)
    .addFilterAfter(new AuthoritiesLogginAfterFilter(), BasicAuthenticationFilter.class)
    .csrf(csrf -> csrf.disable())
    .formLogin(Customizer.withDefaults())
    .httpBasic(Customizer.withDefaults());


    return http.build();

  }
```

Adding a custom filter using addFilterAt:

It adds a filter at the location of the specified filter class, but the order of the execution can't be guaranteed. It will not replace the filters already presented at the same order.

Since we will not have control on the order of the filters and it is random in nature, we should avoid providing the filters at same order.

LoggingFilterAt.java:

```
package com.masai.config;

import java.io.IOException;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import jakarta.servlet.Filter;
import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.ServletRequest;
import jakarta.servlet.ServletResponse;

public class LoggingFilterAt implements Filter {

  private final Logger log = LoggerFactory.getLogger(LoggingFilterAt.class);

  @Override
  public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
      throws IOException, ServletException {

    log.info("Authentication validation is in progress");

    chain.doFilter(request, response);

  }
}
```

Configuring the above filter inside the Spring Security configuration flow:

```
@Bean
  public SecurityFilterChain springSecurityConfiguration(HttpSecurity http) throws Exception {

    http.authorizeHttpRequests(auth ->{

      auth
      .requestMatchers(HttpMethod.POST,"/customers").permitAll()
    .requestMatchers(HttpMethod.GET, "/customers","/hello").hasRole("ADMIN")
    .requestMatchers(HttpMethod.GET, "/customers/**").hasAnyRole("ADMIN","USER")
      .anyRequest().authenticated();

    }).addFilterBefore(new RequestValidationFilter(), BasicAuthenticationFilter.class)
    .addFilterAfter(new AuthoritiesLogginAfterFilter(), BasicAuthenticationFilter.class)
    .addFilterAt(new LoggingFilterAt(), BasicAuthenticationFilter.class)
    .csrf(csrf -> csrf.disable())
    .formLogin(Customizer.withDefaults())
    .httpBasic(Customizer.withDefaults());


    return http.build();

  }
```

Apart from implementing the Filter interface, there are some another advance concepts are available when we try to implement a custom filter in Spring Security.

1. **Using GenericFilterBean class.**
2. **Using OncePerRequestFilter class**

GenericFilterBean:

It is the simple implementation of **Filter** interface by the Spring Security framework. It is an abstract class

The main advantage with this class is, it is going to provide all the details of our config parameters, init parameters, and the ServletContext parameters which we have configured inside the Servlet configuration file i.e. web.xml, so inside our custom filter if we want all the details of our init, config, context parameters then we can use this abstract class.

OncePerRequestFilter:

It is the child class of GenericFilterBean class.

It is also an abstract class

Whenever we create our custom filter, by default the Spring Security framework can not guarantee that it will execute only one time per request.

There might be scenario, where our servlet container can invoke the same filter multiple times for various reasons. for example, One servlet may forward the request to another servlet and the same filter is configured with both the filters.

But if we want to make sure that our custom filter needs to be executed only once per request at any cost, then we should define our custom filter by extending this abstract class.

In this filter, inside doFilter(-,-,-) method, it is already mentioned that if our custom filter is already executed or not, if it is already executed then it simply call the chain.doFilter(-,-,-) by skipping our custom filter.

In this case we need to write our filtering logic by overriding an abstract method called:

```
abstract void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
      throws ServletException, IOException
```

Inside this **OncePerRequestFilter** class there is a another method called **boolean shouldNotFilter()** where we can define the logic to our custom filter should not executed for some API path.

Note: the BasicAuthenticationFilter, internally extends this OncePerRequestFilter class.

To use the JWT token, we always use this OncePerRequestFilter class only.

Example:

CustomOncePerRequestFilter.java

```
package com.masai.config;

import java.io.IOException;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.web.filter.OncePerRequestFilter;

import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

public class CustomOncePerRequestFilter extends OncePerRequestFilter {

  private final Logger log = LoggerFactory.getLogger(OurCustomerOnceFilter.class);

  @Override
  protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
      throws ServletException, IOException {

    Authentication auth = SecurityContextHolder.getContext().getAuthentication();

    if (auth != null) {

      log.info("User " + auth.getName() + " is successfully authenticated and has Authorties "
          + auth.getAuthorities().toString());
    }

    filterChain.doFilter(request, response);

  }

  // only apply this filter for signIn endpoint, for remaining endpoint it will not filter
  @Override
  protected boolean shouldNotFilter(HttpServletRequest request) throws ServletException {

    return !request.getServletPath().equals("/signIn");

  }

}
```

Just replace the above filter with AuthoritiesLogginAfterFilter inside the AppConfig class of previous example.

This filter will be invoked only for the /signIn URI, for remaining URI it will not be invoked and this filter will make sure that it will be executed only once per request, it will not execute multiple time internally.

Token Based Authentication using JSON Web Token (JWT):

As of now we have used to cookies while calling our API using client, i.e. JSessionId. It is randomly generated value, we called them as tokens. using these tokens only our Spring Security framework do lots of magic. these cookie will be automatically appended by the browser for subsequent request. based on these cookie value, our Spring Security is going to identify whether the user is logged in or not. tokens play a very important role in authN and authZ.

These tokens are vary simple in nature and they are not going to help when we develop a complex application like microservices, where different-different application wants to communicate each other. that's we need to use other options in terms of tokens generating the tokens and sharing between the multiple application, because the default token generated by the Spring Security framework it works fine with a smaller application but if we want to build some Enterprise Application, this JSessionId will not be sufficient because for 2 reasons:

1. this token does not holds any user data.
2. this JsessionId is being shared as a cookie inside our browser and these cookies will be tied to the user session, and for some reason if end user not closing his browser and if the end-user session is still valid then there is a good chance that someone can misuse this kind of token which is stored inside the browser.

that way we should explore more advance option to use the token to use AuthT and AuthZ.

Roles of Tokens in Authentication and Authorization:

A token can be a plain string of format universally unique identifier(UUID) or it can be a type of JSON web token (JWT) usually that get generated when the user authenticated for the first time during login.

On every request to a restricted resource, the client send the access token in the query string or Authorization header. the server then validates the token, and if it is valid, it returns the secure resource to the client.

Advantages of Using Tokens:

Tokens helps us to not shares the credentials for every request. it is a security risk to send the credentials over the network frequently., we just need to share our credentials only at the first time, then for the subsequent request for any API, we need not share our credentials again and again.

- 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 and authorities, etc.

Reusability: We can have many separate servers, running on multiple platforms and domains, reusing the same token for authenticating the token.
like from Google, we can navigate to google drive, map, or photos, with the same token.
it is also known as SSO(single sign on).

Stateless: easier to scale, The tokens contains all the information to identify the user, eliminating the need for the session state, if we use a load balancer(inside the cluster environment, where our same application runs over multiple instances), we can pass the user to any server, instead of being bound to the same server we logged in on.

JWT tokens:

It is a token implementation which will be in JSON format(it maintain the data in JSON format) and designed to use for the web-requests.

It is the most common and favorite token type that many system uses these days due to its special feature and advantages.

JWT tokens can be used both in the scenarios of Authentication/Authorization along with information exchange. which means we 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.

It will be very helpful in the microservices environment where multiple application instances running on a different-different servers and our application be stateless.

Each JWT contains encoded JSON objects. The token is mainly composed of a **header, payload, and signature**. These three parts are separated by period(.).

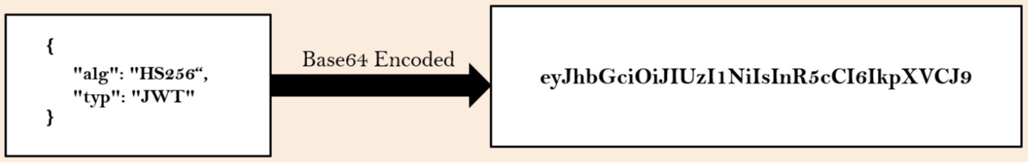
A JWT token has 3 parts each separated by a period(.) Below is a sample JWT token,

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

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

Header:

✓ 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.

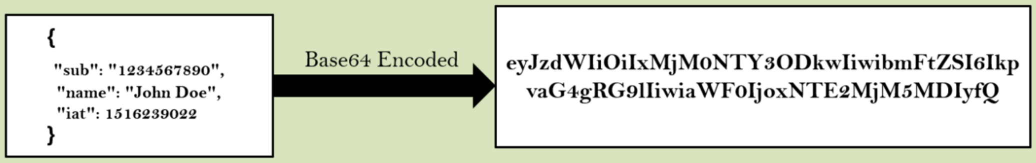


```
{
    "alg": "HS256“,
    "typ": "JWT"
}
```

Note: all the info of the JWT token will not be send in the plain text format, instead we should Base64 encode that value.

Payload:

✓ 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

Signature:

To make sure no one tempered the data on the network, we can add the signature of the content when initially token is generated. To create the signature part you have to take the encoded header, the encoded payload, a secret key(This secret key/value will be known only to the backend application where it is issuing a JWT token.), the algorithm specified in the header, and sign that.

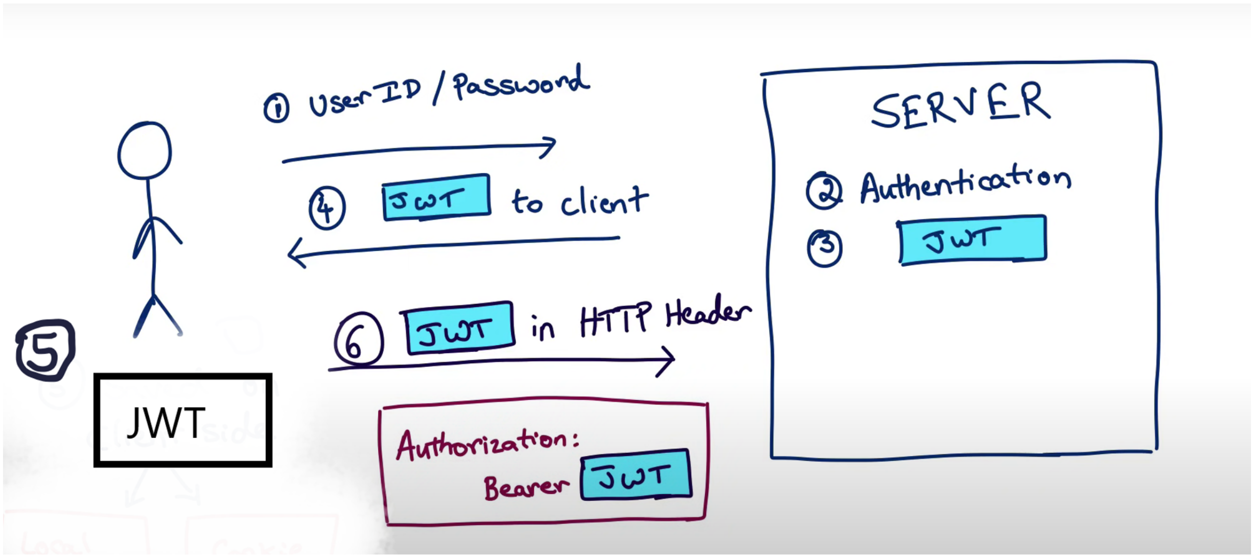
- ✓ For example if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way:

  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.

We can play with the JWT token with the website: jwt.io debugger to decode, verify and generate Jwts.

Making use of JWT token inside our application:



- User sign-in using username and password.
- Backend server verifies the credentials and issues a JWT token signed using a private key.
- Client uses the JWT to access protected resources by passing the JWT in HTTP Authorization header.
- Backend server then verifies the authenticity of the token using the secret key and grant permission to the protected API.

In order to use the JWT token inside our application, we need to follow the following steps:

Step1: Inside the pom.xml we need to add some JWT related dependencies which will going to help in generating and validating the JWT tokens.

```
<dependency>
      <groupId>io.jsonwebtoken</groupId>
      <artifactId>jjwt-api</artifactId>
      <version>0.11.1</version>
    </dependency>
    <dependency>
      <groupId>io.jsonwebtoken</groupId>
      <artifactId>jjwt-impl</artifactId>
      <version>0.11.1</version>
      <scope>runtime</scope>
    </dependency>
    <dependency>
      <groupId>io.jsonwebtoken</groupId>
      <artifactId>jjwt-jackson</artifactId>
      <version>0.11.1</version>
      <scope>runtime</scope>
    </dependency>
```

Step2: Perform some changes inside our Spring Security configuration:

As of now inside our application, by default Spring Security framework is generating a JSessionId and it uses that JSsesionId for all the subsequent request that our user is going to make post successful authentication.

So we need to disable that default behavior, post that only we can generate our own JWT token.

```
http.sessionManagement(sessionManagement -> sessionManagement.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
```

With the above configuration, we are telling the Spring Security framework to not generate any JSessionId, we are going to take care of our own session management or token management inside our application.

Step3: Expose the Authorization header to the client with the response:

When we generate the JWT token inside our application after successful authentication, we are going to send that token to the client application inside the response as an header with name **Authorization**.

So inside the **Cors configuration,** we need to define those details telling to the Spring Security framework, please allow to send this header information as part of the response that we are going to send from the backend application to the client application.

Then only the client browser is going to accept that new header which we are going to send in as a part of the response.

```
  http.cors(cors -> {

      cors.configurationSource(new CorsConfigurationSource() {

        @Override
        public CorsConfiguration getCorsConfiguration(HttpServletRequest request) {
          CorsConfiguration cfg = new CorsConfiguration();

          cfg.setAllowedOriginPatterns(Collections.singletonList("*"));
          cfg.setAllowedMethods(Collections.singletonList("*"));
          cfg.setAllowCredentials(true);
          cfg.setAllowedHeaders(Collections.singletonList("*"));
          cfg.setExposedHeaders(Arrays.asList("Authorization"));
          return cfg;
        }
      });

    })
```

Using this setExposedHeader method we can expose the headers that we are sending inside the response to the client application. Otherwise our browser is not going to accept these headers, because there are 2 different origins are trying to communicate. So, that's why all these details we need to define as part of CorsConfiguration.

Step4: Create a JWT token generator filter.

As part of that filter, we can write the logic to generate the JWT token as soon as the authentication is successful during the login operation.

```
SecurityConstants.java
-----------------------

package com.masai.config;

public interface SecurityConstants {

  public static final String JWT_KEY ="secretsfhsfjhdkjngdfjkgfgjdlkfjsdkfjsd";
  public static final String JWT_HEADER = "Authorization";

}
```

```
JwtTokenGeneratorFilter.java
-----------------------------

package com.masai.config;

import java.io.IOException;
import java.util.Collection;
import java.util.Date;
import java.util.HashSet;
import java.util.Set;

import javax.crypto.SecretKey;

import org.springframework.security.core.Authentication;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.web.filter.OncePerRequestFilter;

import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.security.Keys;
import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;


public class JwtTokenGeneratorFilter extends OncePerRequestFilter {

  @Override
  protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
      throws ServletException, IOException {

    System.out.println("inside doFilter....");

    Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
        if (null != authentication) {

          System.out.println("auth.getAuthorities "+authentication.getAuthorities());


            SecretKey key = Keys.hmacShaKeyFor(SecurityConstants.JWT_KEY.getBytes());



            String jwt = Jwts.builder()
                .setIssuer("Ram")
                .setSubject("JWT Token")
                    .claim("username", authentication.getName())
                    .claim("authorities", populateAuthorities(authentication.getAuthorities()))
                    .setIssuedAt(new Date())
                    .setExpiration(new Date(new Date().getTime()+ 30000000)) // expiration time of 8 hours
                    .signWith(key).compact();

            response.setHeader(SecurityConstants.JWT_HEADER, jwt);

        }

        filterChain.doFilter(request, response);
  }

    private String populateAuthorities(Collection<? extends GrantedAuthority> collection) {

      Set<String> authoritiesSet = new HashSet<>();
```

```
        for (GrantedAuthority authority : collection) {
            authoritiesSet.add(authority.getAuthority());
        }
        return String.join(",", authoritiesSet);


    }

//this make sure that this filter will execute only for first time when client call the api /signIn at first time
  @Override
  protected boolean shouldNotFilter(HttpServletRequest request) throws ServletException {

        return !request.getServletPath().equals("/signIn");
  }


}
```

Step5: Configure the above Filter inside the SecurityFilterChain:

```
.addFilterAfter(new JwtTokenGeneratorFilter(), BasicAuthenticationFilter.class)
```

- Inside the above filter, since we are injecting this filter after the BasicAuthenticationFilter, which means the authentication of our end-user will be successful by the time this filter will get invoked.
- We will get the current Authenticated user details from the **SecurityContext** object. here we are maintaining the secret key and Jwt_header inside a separate interface called **SecurityContext**.
- This secret key value will be known to our backend application only, this has to be super secure. but in ideal production environment, we should ask our Devops team to inject this value at runtime, during the deployment of our application as an environmental variable using CI/CD tools like the Jenkins or GitHub. or we can also configure this as an environmental variable inside our production server, and same can be accessed from our code.
- Using **claim()** method we can set any kind of user information to our JWT token.
- The purpose of the **shouldNotFilter()** method is, if we provide a condition to this method, based upon the condition, it will not execute the filter. So here our requirement is that this **JwtTokenGeneratingFilter** should get executed only during the **signIn** process, because we don't want the token to be generated again and again for all the subsequent requests. we want this to be done only during the signIn operation.
- So the login operation will be called by the client using **/signIn** api.

Step6: create an endpoint by name: /signIn inside the controller

```
@GetMapping("/signIn")
  public ResponseEntity<String> getLoggedInCustomerDetailsHandler(Authentication auth){

    System.out.println(auth); // this Authentication object having Principle object details

     Customer customer= customerService.getCustomerDetailsByEmail(auth.getName());

     return new ResponseEntity<>(customer.getName()+"Logged In Successfully", HttpStatus.ACCEPTED);


  }
```

Step7: Creating another Filter to validate the JWT token sent by the client:

Now we need to define another filter which will be responsible to validate the JWT token that we receive from the client application for all the subsequent requests that they are going to make post successful authentication.

This filter will validate the JWT token whenever our client application is sending for all the REST API's other than signIn operation.

```
JwtTokenValidatorFilter.java:
-----------------------------


package com.masai.config;

import java.io.IOException;
import javax.crypto.SecretKey;
import org.springframework.security.authentication.BadCredentialsException;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.authority.AuthorityUtils;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.web.filter.OncePerRequestFilter;

import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.security.Keys;
import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

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(jwt != null) {

      try {

        //extracting the word Bearer
        jwt = jwt.substring(7);


        SecretKey key= Keys.hmacShaKeyFor(SecurityConstants.JWT_KEY.getBytes());

        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(au

        SecurityContextHolder.getContext().setAuthentication(auth);

      } catch (Exception e) {
        throw new BadCredentialsException("Invalid Token received..");
      }

    }

    filterChain.doFilter(request, response);

  }



  //this time this validation filter has to be executed for all the apis except the /signIn api

  @Override
  protected boolean shouldNotFilter(HttpServletRequest request) throws ServletException {

    return request.getServletPath().equals("/signIn");
  }

}
```

Step8: Configure the above JwtTokenValidatorFilter to the Spring Security FilterChain:

This filter we need to execute before the authentication attempt by the Spring Security framework. So we need to configure this filter with addFilterBefore() method.

Example:

```
.addFilterBefore(new JwtTokenValidatorFilter(),BasicAuthenticationFilter.class)
```