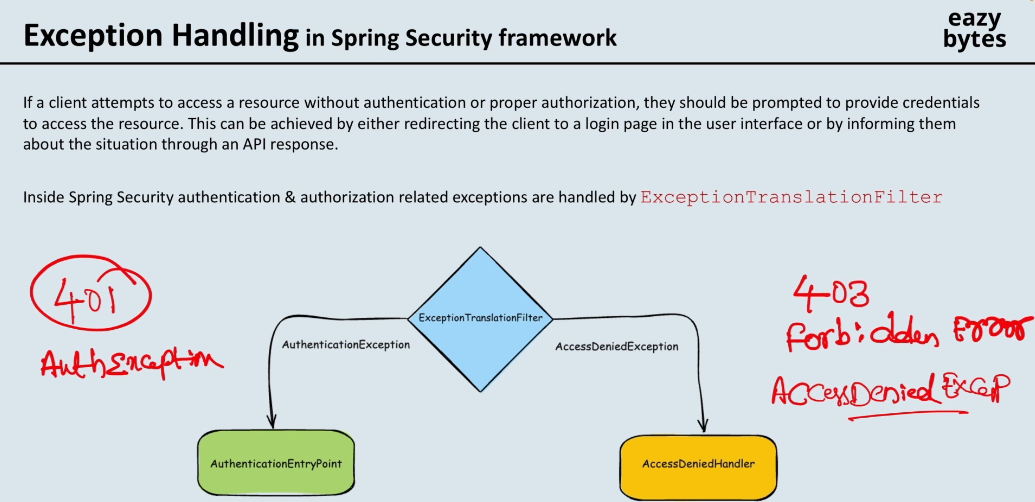
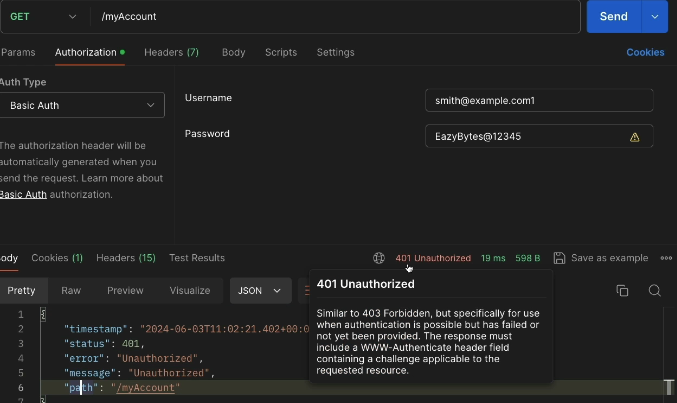
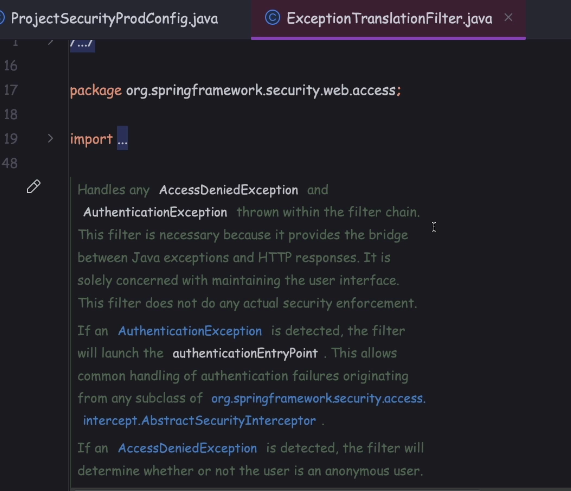
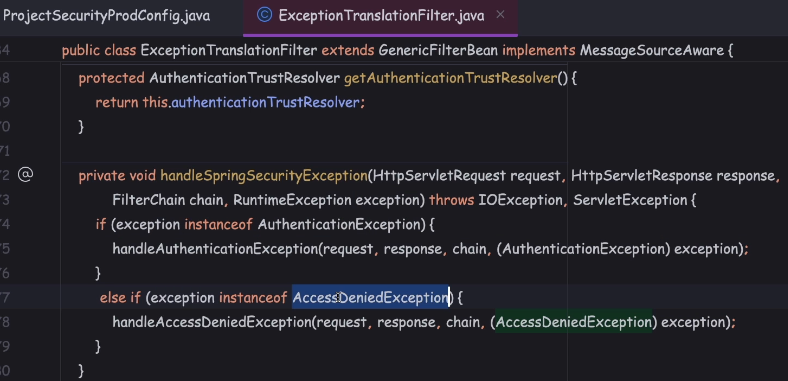
**6.spring security customization ; section 7**

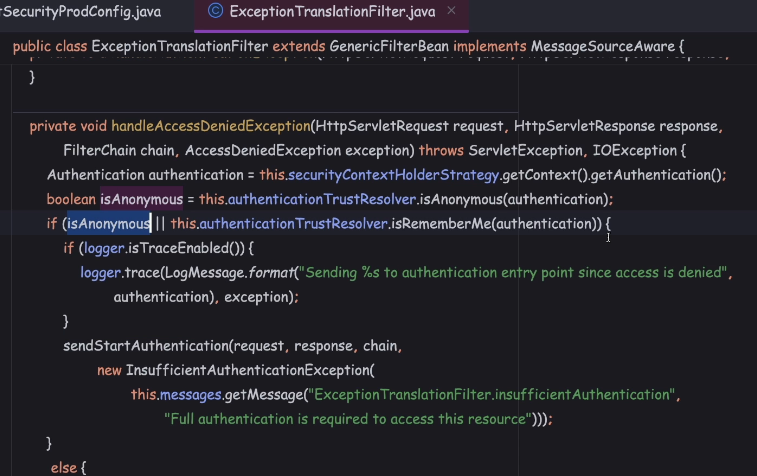
Accepting only HTTPS Traffic using Spring Security

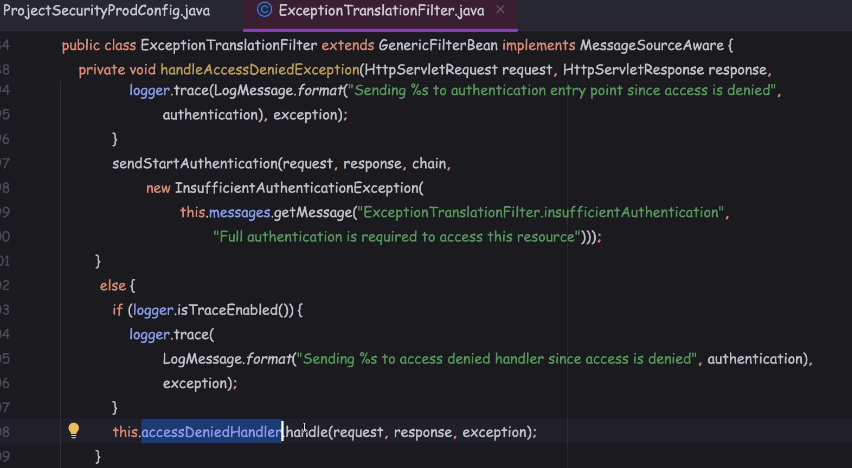
Exception Handling in Spring Security framework

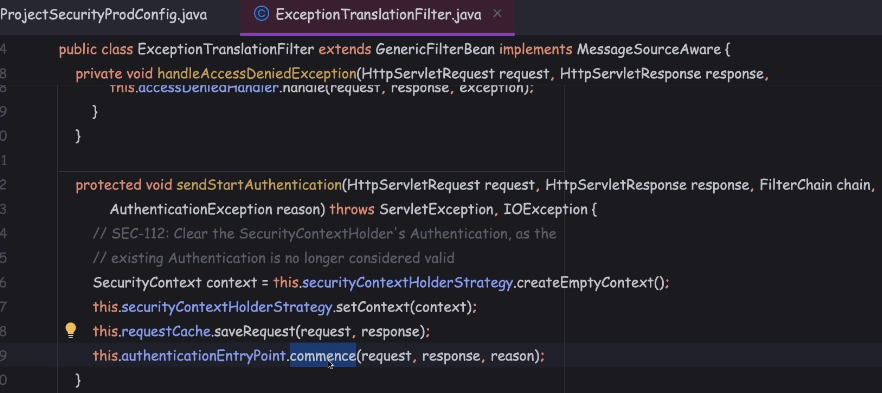
****

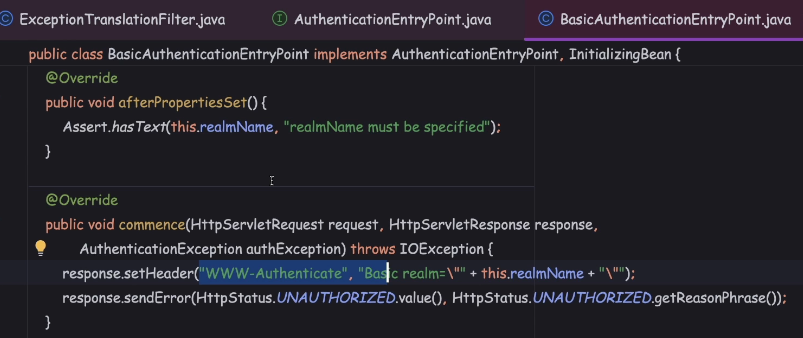
****

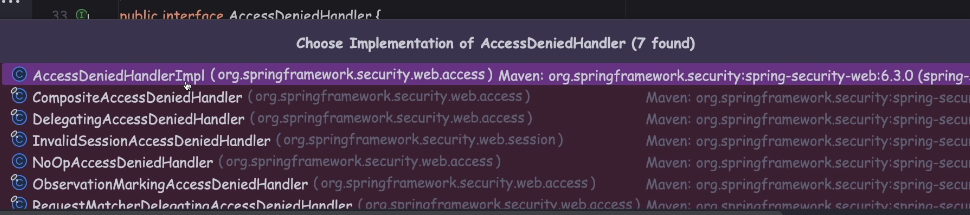
****

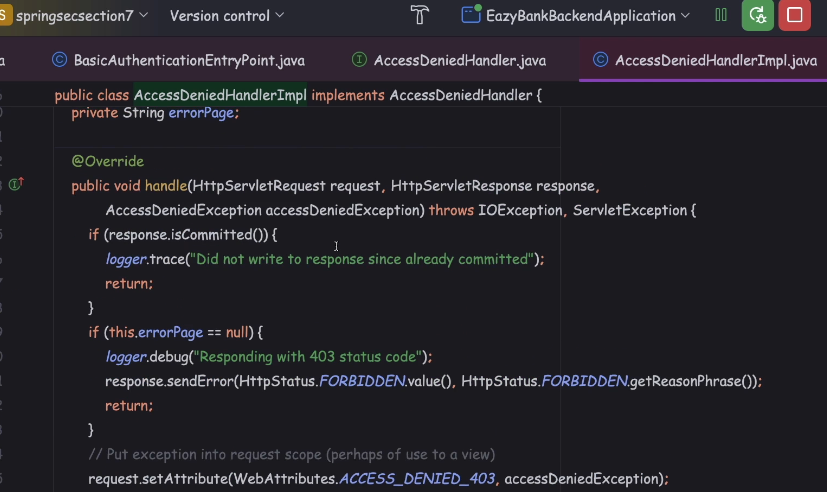
****

****

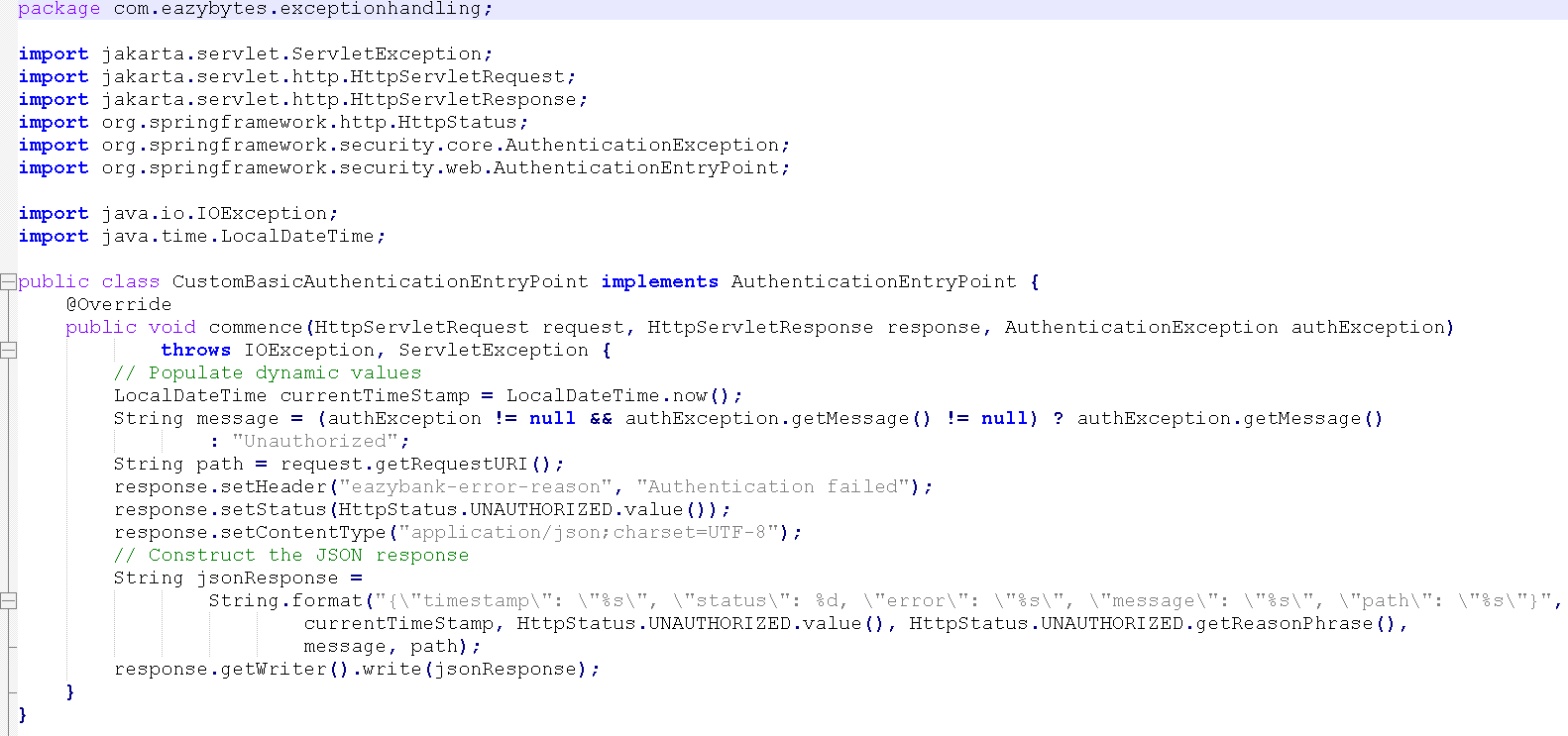
****

****





Defining Custom AuthenticationEntryPoint, Custom AccessDeniedHandler







what is the need of configuring these authenticationEntryPoint() at globally with the help of exceptionHandling?

If you try to configure the authenticationEntryPoint() only for the httpBasic(),

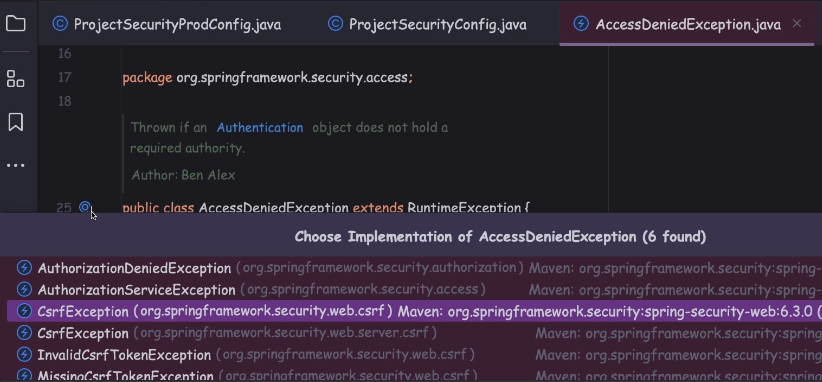
then these authenticationEntryPoint() is going to be executed or considered

only during the httpBasic() flow, like during the login flow.

But apart from login flow, Spring Security may also throw 401 or AuthenticationException in many other places throughout the framework.

To handle all such possible scenarios, you can go with these global configuration.

Below are subclasses of access denied exceptions



Session Timeout & invalid session configurations

Default session timeout is 30 mins

server.servlet.session.timeout=${SESSION\_TIMEOUT:20m}

That's why Spring Boot never allow you to configure a timeout which is less than two minutes.

If I try to tamper this value by changing this eight to the six, then obviously this is an invalid session.

In this scenario, also though my session is not timed out, but I'm trying to access the API with an invalid session.

Even in these kind of scenarios also, the end user will be redirected

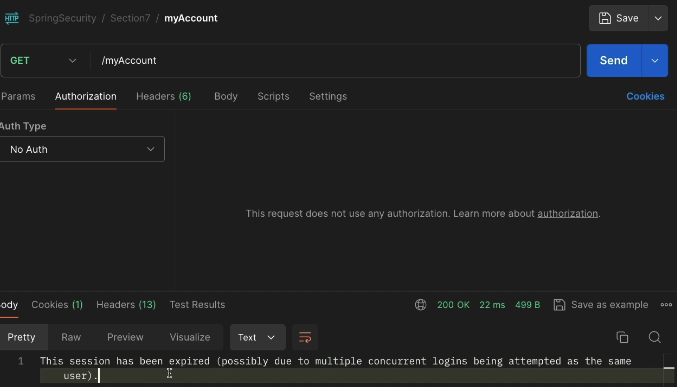
to the invalid session page instead of login page.

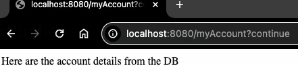
Regardless whether your session is getting invalidated due to some invalid tampered value or due to some timeout scenarios, in all these kind of scenarios,

the end user will be redirected to this page instead of the login page.

Concurrent Session Control configurations

http.sessionManagement(smc -> mc.invalidSessionUrl("/invalidSession").maximumSessions(1))





So the behavior right now we have is if the end user creates a new second session, the first session that the end user created is getting invalidate.

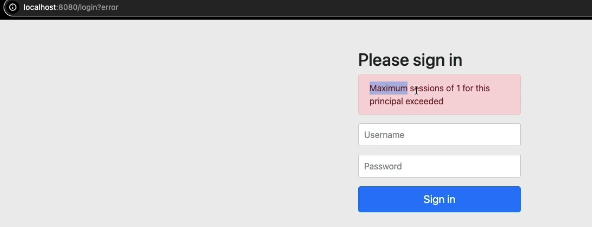
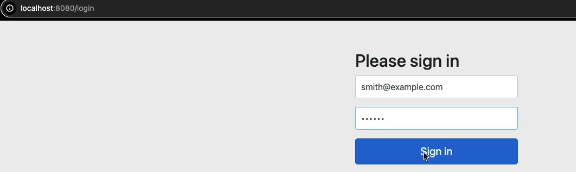
We can also change this behavior such a way that the end user can't create the second session, which means during the login of the second session itself,

the end user will be stopped and the first session is always going to be retain.

We can achieve this behavior by making a small configuration inside the Spring Security configurations. The configuration that we have to do here is we have

to invoke a method which is maximumSessionsPreventsLogin().

http.sessionManagement(smc -> smc.invalidSessionUrl("/invalidSession").maximumSessions(3).maxSessionsPreventsLogin(true))

access in browser after getting session in postman : localhost:8080//myAccount

if you want to have a fancy expired URL page, then feel free to leverage the expiredUrl() method. But please note that whenever you are configuring the expiredUrl() with the path, make sure that expiredUrl()path is also configured under the permitAll().

 Session Fixation Attack protection with Spring Security

To avoid Session Hijacking attack scenarios there are multiple approaches.

The very first approach the organizations they're going to use is, to use the history HTTPS protocol.

If they use the history HTTPS protocol for the communication, then hacker cannot steal the session ID during the network traffic or while your request is traveling throughout the internet.

And the next approach that typically organizations are going to follow is they're going to limit the session ID timeout for short time, like 10 minutes or 15 minutes.

At the same time, few organizations during the login operation,

they have a checkbox for the end user to check if they're using a public computer.

In such scenarios, the organizations are going to be very cautious in creating the cookies inside the browser.

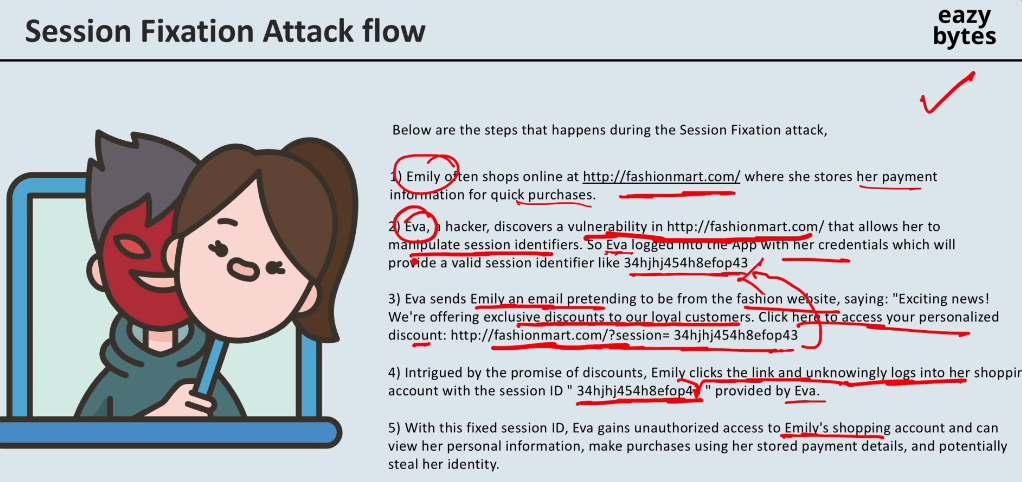
Instead of creating the cookies inside the user browser, they may switch to some other option.

So if you see here in the scenario of session hijacking, there is no role played by the Spring Security directly.

So the organization, they have to take multiple measurements like using the history HTTPS protocol, asking the end user if they're using the public computer

and having the session timeout with a very short time.

By using all these approaches, we can avoid the Session Hijacking scenario.



like how match fixation is going to happen inside the cricket matches and football matches. Very similarly here also the fixation is happening around the session ID.

Initially the hacker is going to generate a session ID and the hacker is going to provide the session ID to some other normal user and if the normal user use the same session ID to enter into the web application, then all the details of the normal user are going to be associated to the session ID which is originally shared by the hacker.

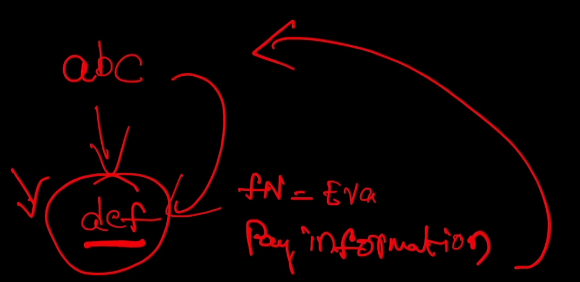
And with that hacker can happily use the session ID to know the normal user details.

This is happening because there is a vulnerability inside these Fashion Mart application. Ideally they should have some checks to avoid the Session Fixation Attack.

By default, if an application is using Spring Security as a dependency, Spring Security is going to take care of handling the Session Fixation Attack.



**changeSessionId** **strategy**



Let's assume the hacker initially performed the login operation and the hacker gave the session ID abc to normal person.

When the normal person trying to log in into the application with the **changeSessionId** **strategy**, the normal person is going to get the newSessionId

like def, but this session it is going to have all the details from the previous session.

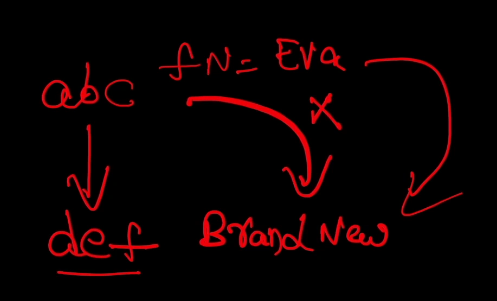
For example, if the previous session has first name as Eva, the same details are going to be available inside this session as well with a different session ID which is def.

With this what is going to happen, the hacker will never know what is the new session ID that got generated on the normal person computer and with that he should not be able to see the extra details that are sailed inside this session.

Maybe payment information is stored after the login is completed.

So this extra information will never reach to the hacker because the hacker does not know the new session ID that got created.

**So this is the default approach that is being used by the Spring Security and Spring Boot framework as of now.**



The next strategy available is new session.

In this newSession what is going to happen is, for example, first the hacker logged in

and the hacker got a session ID which is abc. The same is sent to the normal person.

And the normal person this time, even though he's trying to log in with the session ID abc but behind the scenes a new brand newSession is going to be created with the session ID like def.

So inside these brand newSession, the previous session details

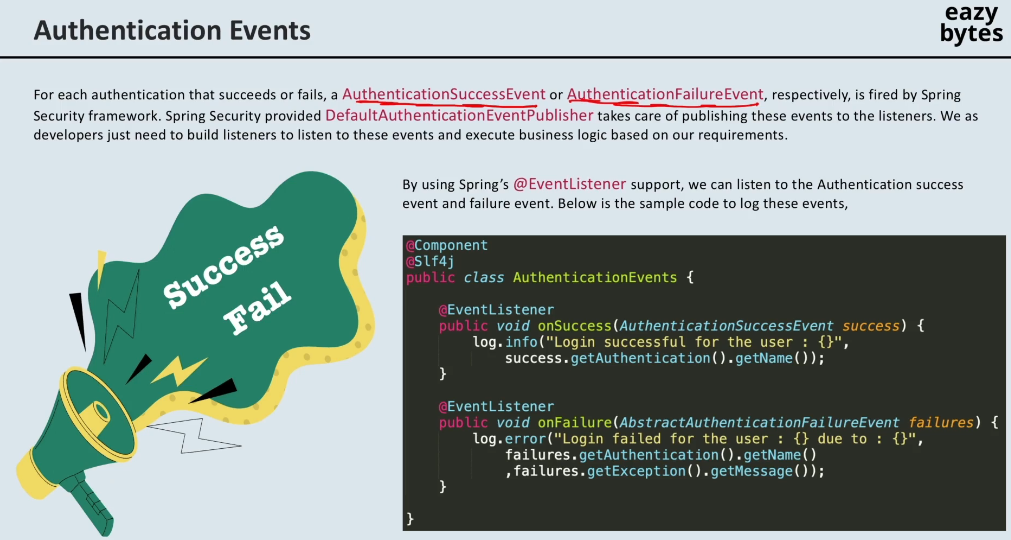
that are not going to be copied. For example, inside this session, if there is a variable which is first name as Eva, it is not going to be copied to the brand newSession

that is getting created here with the session ID def.

But please note that Spring Security related attributes, they're going to be copied into this brand new scenario.

So that is the only caveat that you need to be aware whenever you are using the newSession strategy.

 Listening Authentication Events



For example, on successful authentication, you may want to send an email to an end user saying that you have successfully authenticated, just informing the end user about the login operation.

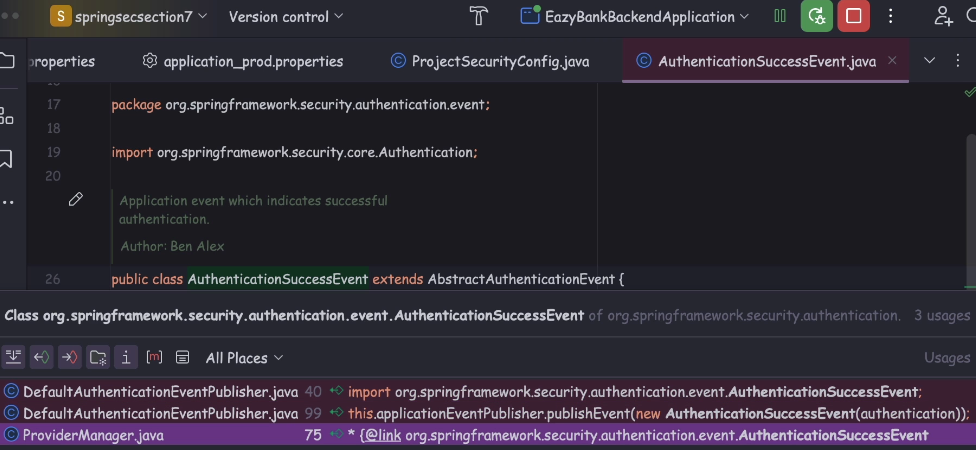
This might be required for the super critical applications, and similarly, in the scenarios of authentication failure also, you may want to send an email

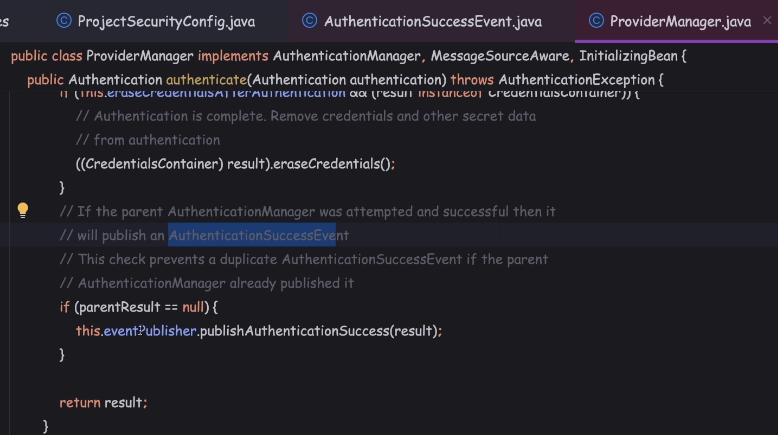
or you may want to make an entry inside the database that so and so failure attempts happened.

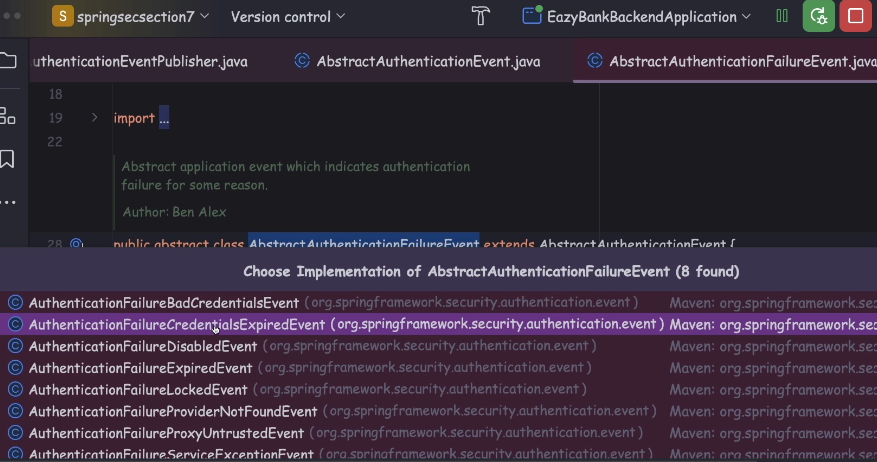
So, these kind of requirements are very common inside real applications.

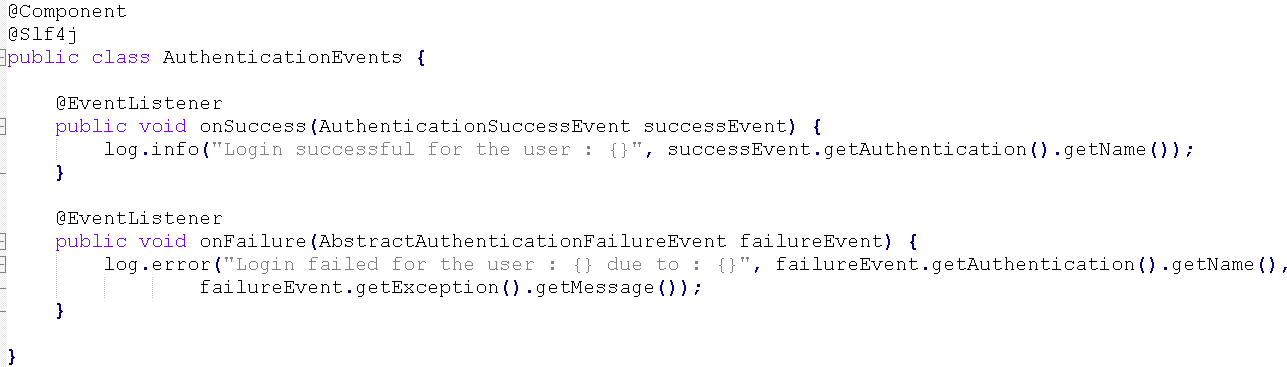
To handle these kind of requirements, Spring Security is going to publish events

whenever an authentication is going to be successful and whenever the authentication is a failure.



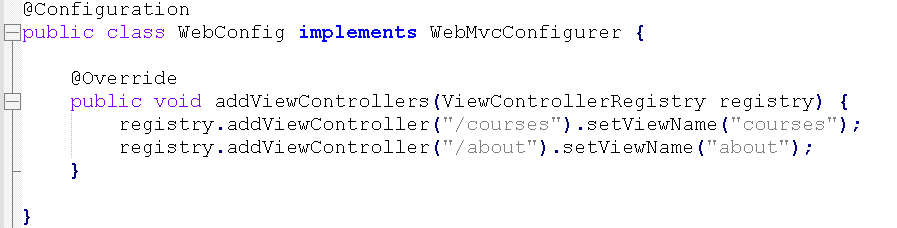


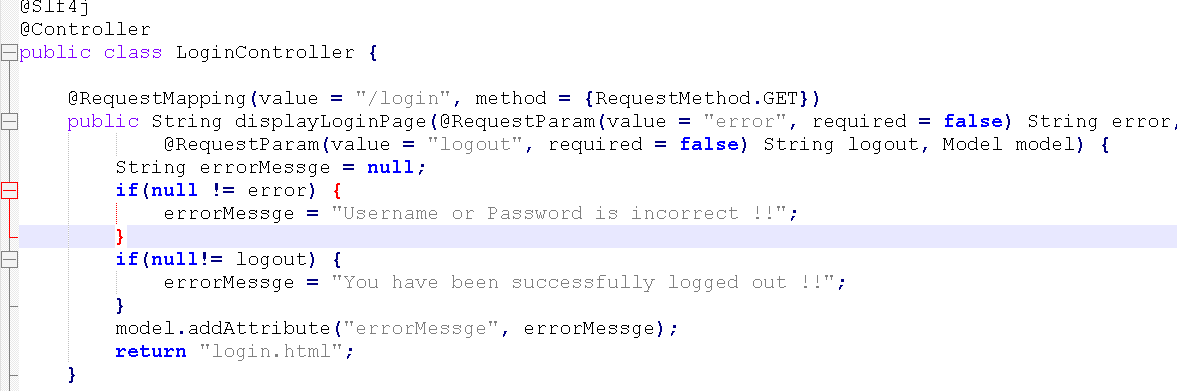


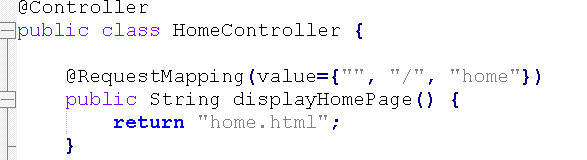


Form Login Configurations for MVC or monolithic apps : section7 - easyschool

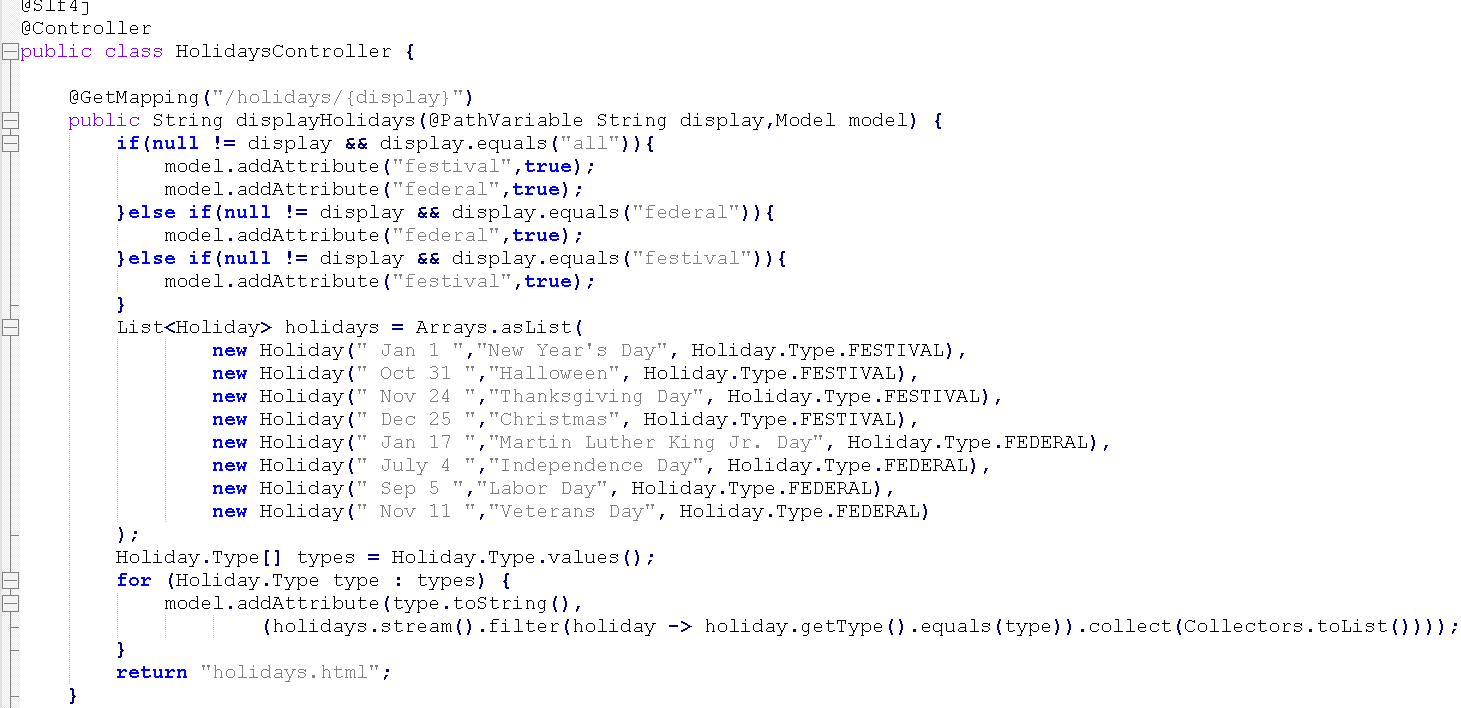


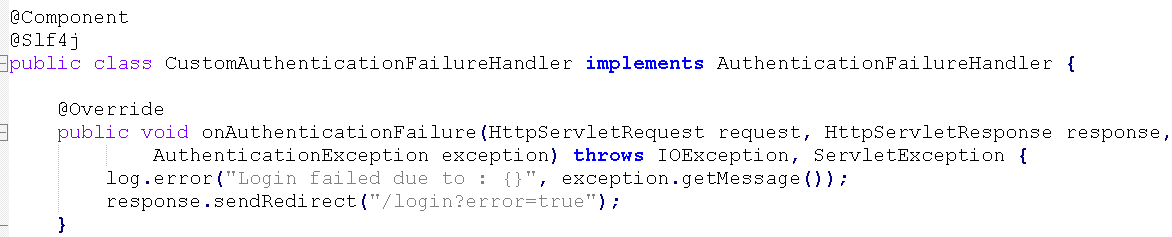


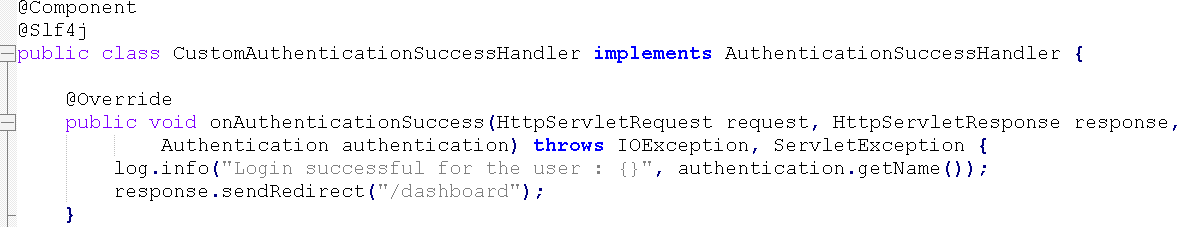








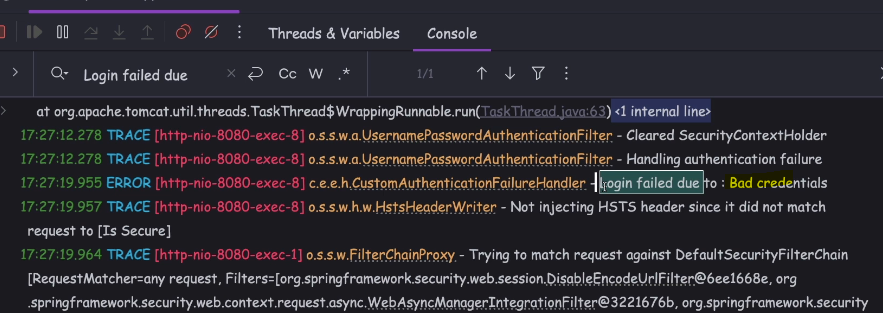


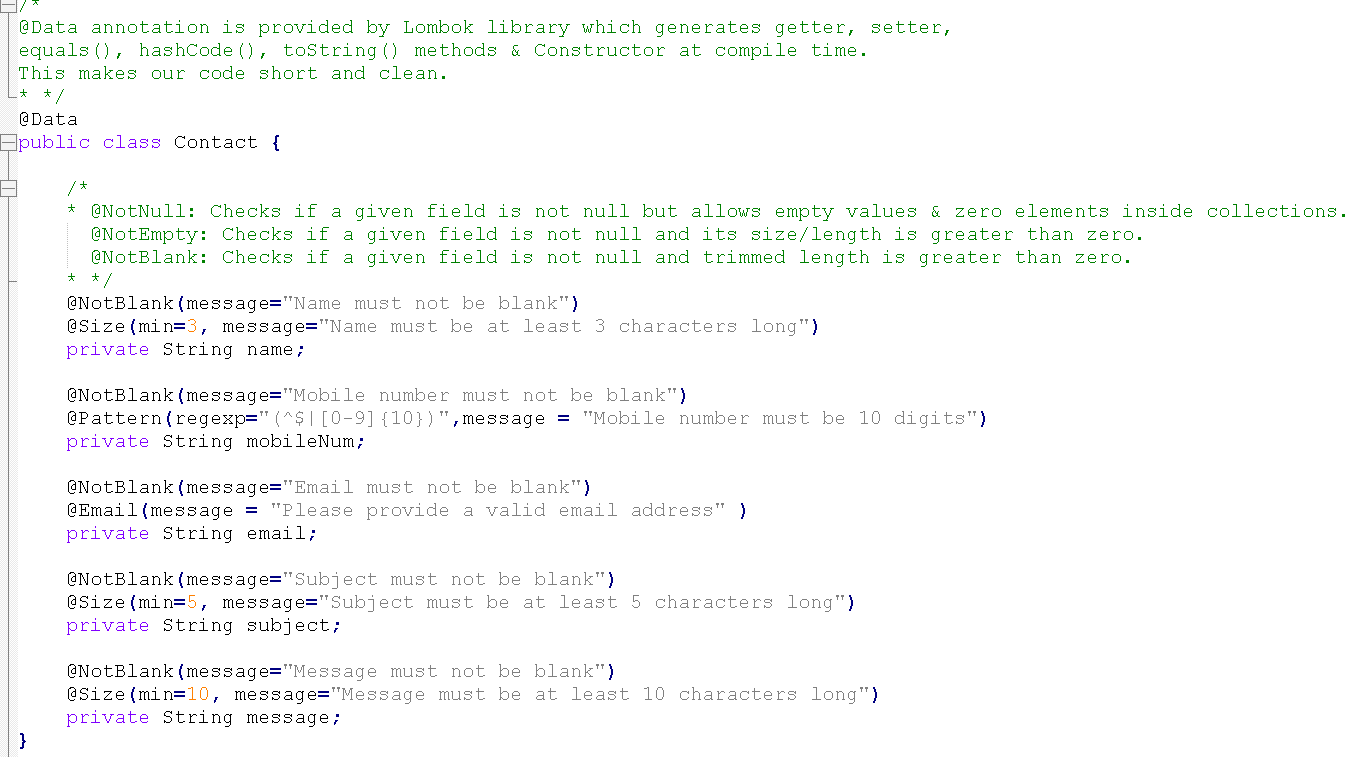


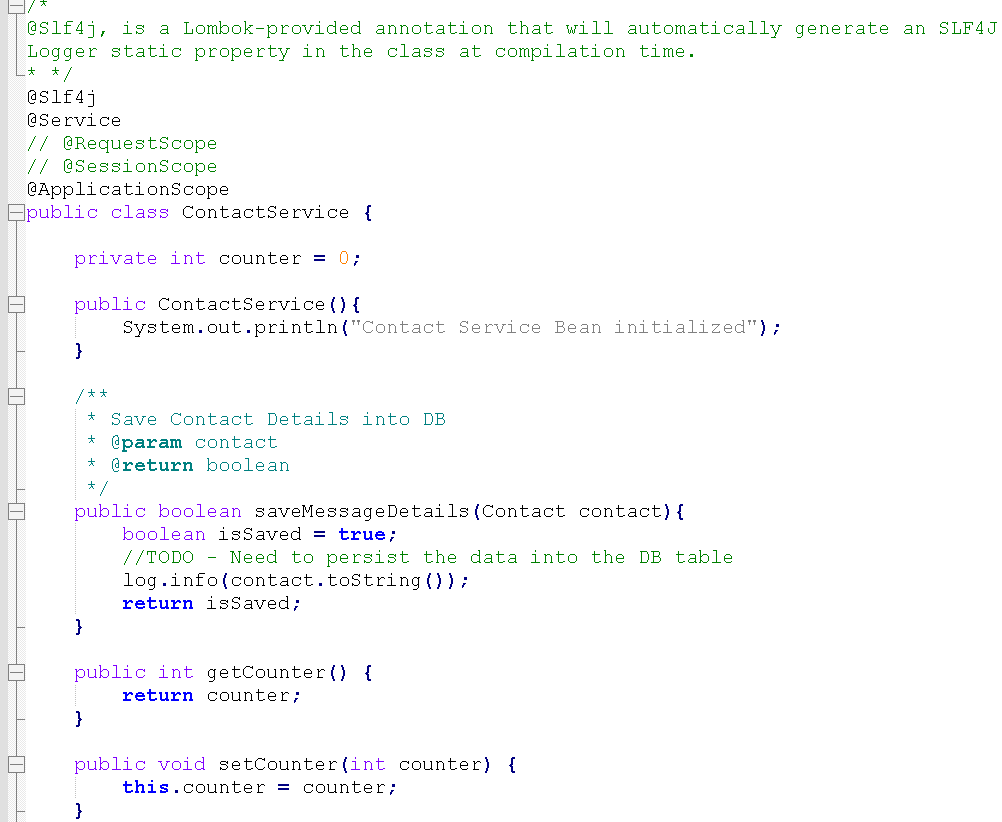
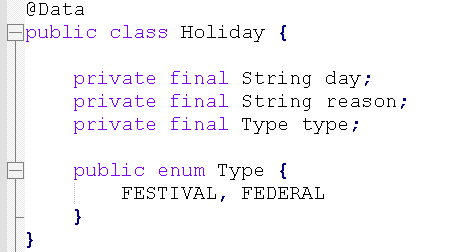
Spring Security is giving priority to these handlers. Though we have provided the same configuration with the defaultSuccessUrl() and the failureUrl().

So in real projects, you can either use this approach at the second approach that I have highlighted here.

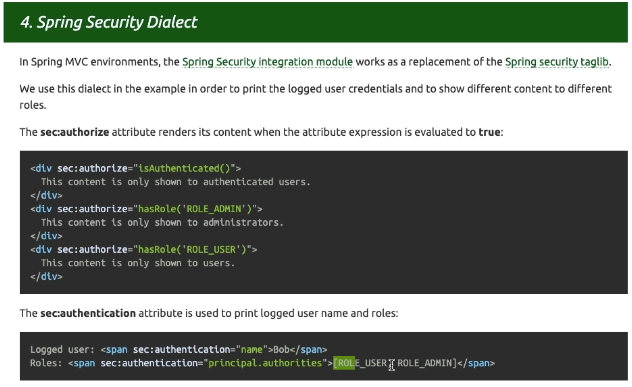
If you want more control in your business logic, then always use a second approach.

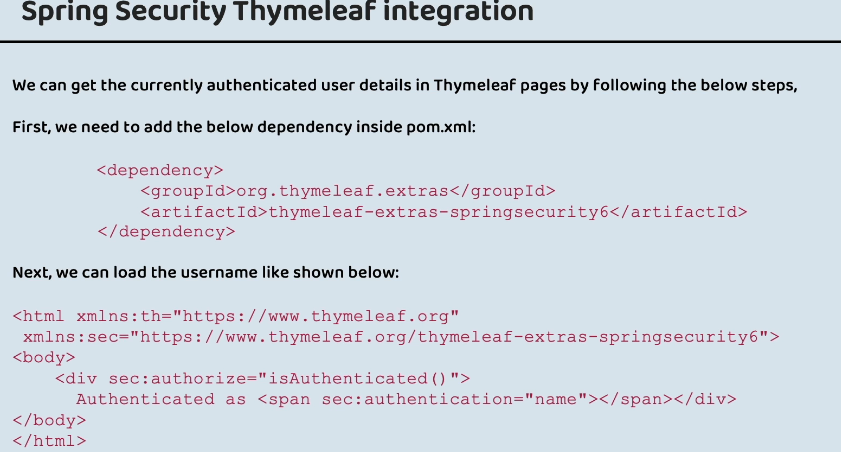




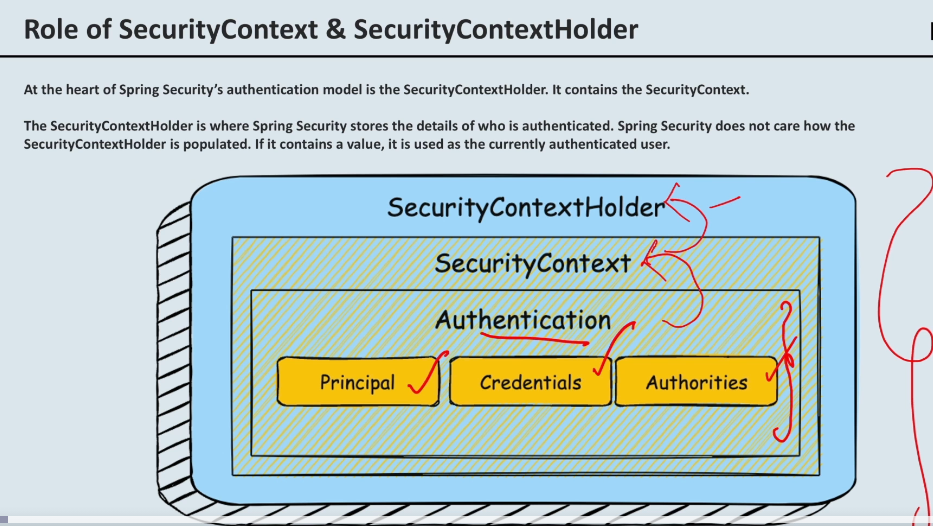
 

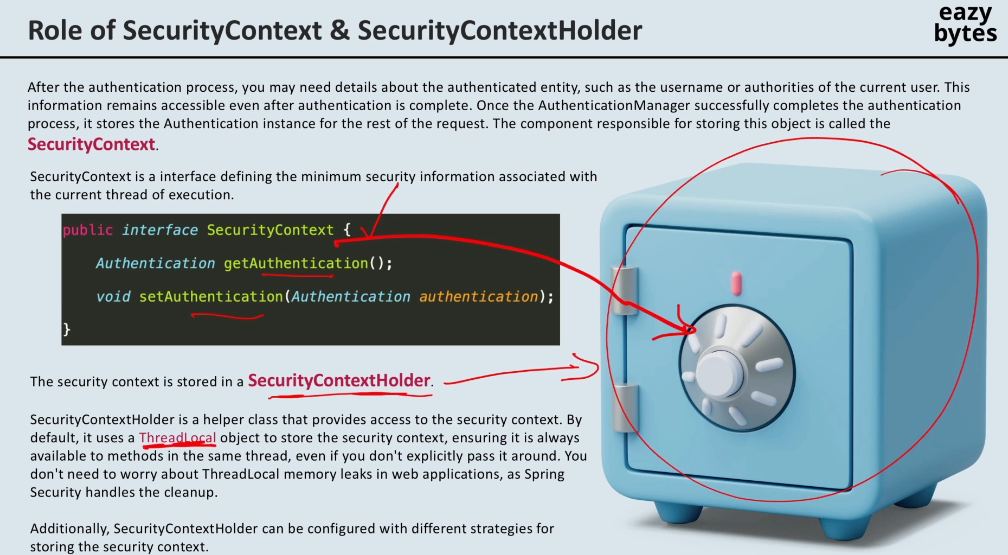
Spring Security Thymeleaf integration

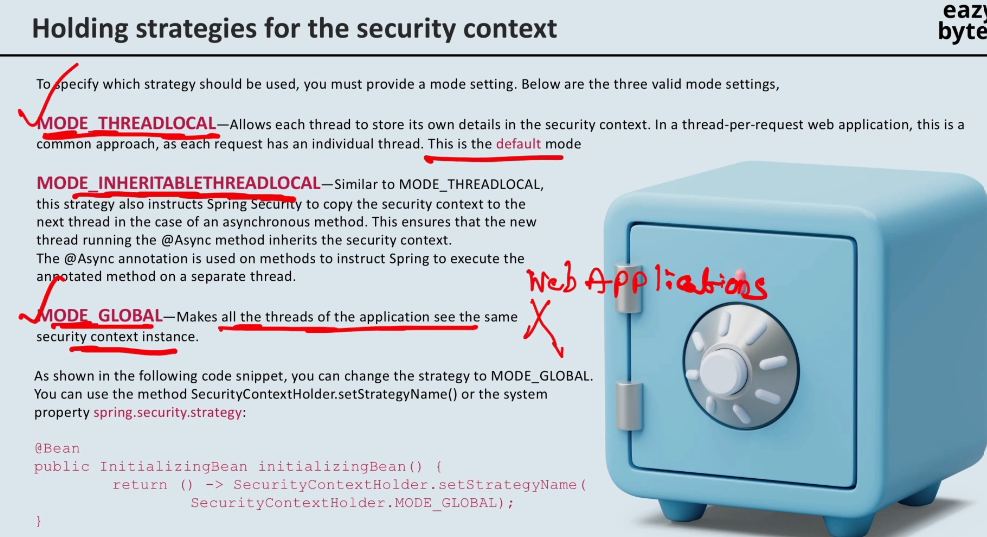


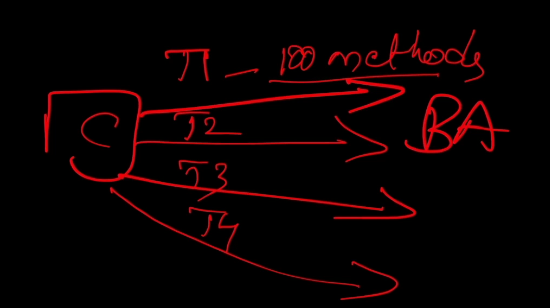


Role of SecurityContext & SecurityContextHolder









how the ThreadLocal is going to work.

So whenever the client is sending a request to the backend application,

for each request, there is going to be a new thread is going to be created.

So think like these are all threads like T1, T2, T3, and T4.

Each thread may invoke hundreds of different methods during its execution.

Think like this T1 is invoking hundred methods. We can't manually pass the SecurityContext object or Authentication object that is stored

inside the SecurityContextHolder manually to all these hundred different methods.

Each of the thread, they're going to have their own ThreadLocal.

So any objects that are stored inside the ThreadLocal, they're going to be access to all the methods that are going to be invoked as part of the thread journey.

So this is how the ThreadLocal is going to work.



For example, there is a mode with the name MODE\_**INHERITABLETHREADLOCAL**.

Let me explain you under which scenarios this INHERITABLETHREADLOCAL is going to be used.

Think like there is a client application, and there is a backend application.

So this client application can be your browser. If a user is trying to send a request

to the backend application, a thread is going to be created, which is T1,

and inside the backend logic, somewhere you might have written some asynchronous logic.

We have annotations like @Async inside the Spring framework.

Whenever you are using this annotation on top of a method, the logic inside that method is going to be executed using a separate asynchronous thread.

In such scenario, what is going to happen?

A new thread is going to be created by your backend method which is annotated with the @Async annotation.

So these threads, we can call them as asynchronous thread.

Let's name this asynchronous thread as T2.

Technically, if you see all these threads, they are being processed inside the same request.

In these kind of scenarios, whenever you are trying to create the asynchronous threads inside your business logic, you can use the **inheritable** mode.

Inside this mode, what is going to happen is whenever an asynchronous thread is going to be created by the Spring framework,

all the details available inside this T1, they're going to be copied into the T2 thread,

which is an asynchronous thread.

This way, the SecurityContext details, they're going to be available to the T2 thread as well.

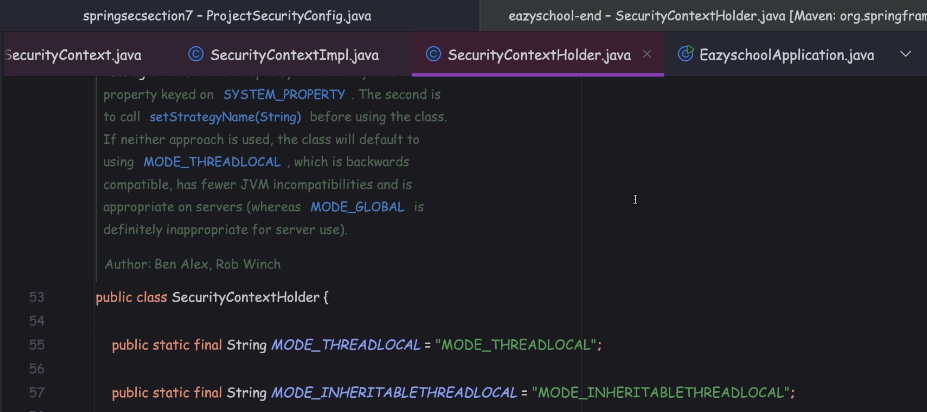
If you're not using this INHERITABLE mode, and if you're using the default THREADLOCAL mode, then in such scenarios, this T2 thread is not going to have

any information around the SecurityContext.

we need to use is MODE\_GLOBAL.

Think like you are building a desktop application.

So usually desktop applications that are going to be used by a same user inside the same system. So if you have a scenario where you are building the desktop application, which can be used by only one user at a time, then in such scenarios you can use this MODE\_GLOBAL strategy.



Load login user details in Spring Security



Top of Form

What is the primary purpose of the **AuthenticationEntryPoint** interface in Spring Security?

**To handle authentication failures by redirecting users to a specific login page.**

Top of Form

Which of the following is the correct configuration to configure custom authentication entry point for handling basic authentication?

**http.httpBasic(hbc -> hbc.authenticationEntryPoint(new CustomBasicAuthenticationEntryPoint()));**

Top of Form

Which of the following is the correct configuration to configure custom access denied handler ?

**http.exceptionHandling(ehc -> ehc.accessDeniedHandler(new CustomAccessDeniedHandler()));**

Top of Form

Which of the following is the correct configuration to limit the maximum number of concurrent sessions per user ?

**http.sessionManagement(smc -> smc.maximumSessions(1).maxSessionsPreventsLogin(true));**

Top of Form

Which option for Session Fixation Protection in Spring Security creates a new session and copies all existing session attributes to the new session?

**migrateSession**

Bottom of Form

Top of Form

Which of the following code snippet is correct to listen and process authentication success events that are published by Spring Security ?@Component

* 1. @Slf4j
  2. public class AuthenticationEvents {
  4. @EventListener
  5. public void onSuccess(AuthenticationSuccessEvent successEvent) {
  6. // Logic to handle success authentication event
  7. }
  9. }

Top of Form

Which of the following code snippet can be used to customize a login, success and failure pages in Spring Security ?

**http.formLogin(formLoginConfig -> formLoginConfig.loginPage("/login")  .defaultSuccessUrl("/dashboard").failureUrl("/login?error=true").permitAll())**

Top of Form

Which of the following code snippet can be used to customize a logout page in Spring Security ?

http.logout(logoutConfigurer -> logoutConfigurer

* 1. .logoutSuccessUrl("/login?logout=true").permitAll()
  2. .invalidateHttpSession(true)
  3. .clearAuthentication(true)
  4. .deleteCookies("JSESSIONID"));

Top of Form

Which of the following code snippet is valid to load the login user details from the SecurityContextHolder ?

**Authentication authentication = SecurityContextHolder.getContext().getAuthentication();**