Spring Security
Lab 3

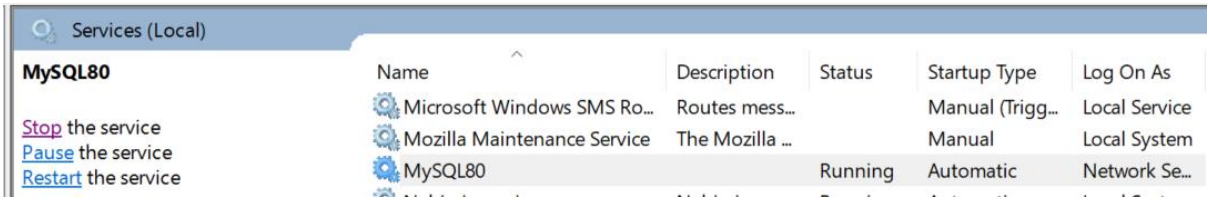
1 Lab setup

Make sure you have the following items installed

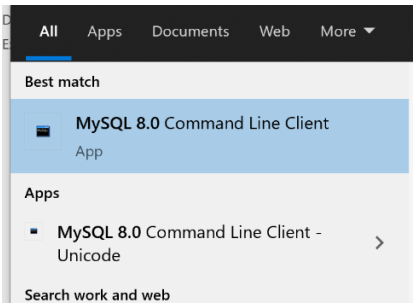
- Latest version of JDK 8 / 11 (note: labs are tested with JDK 8 but should work on JDK 11 with no or minimal changes)
- Spring Tool Suite (STS) IDE
- Latest version of Maven
- A suitable text editor (Notepad ++)
- A utility to extract zip files

2 Setting up a valid user in MySQL

Check that the MySQL server is up and running. This should have already started up as a Windows service by default when your OS boots up. Open the Services utility to check and start the server if it is not already running:



Start up the MySQL command line client from Windows search:



You will be prompted for the admin (root) password. Type this in and you will be presented with the MySQL shell prompt where you can type in SQL statements.

An alternative way to connect to the MySQL server is to open a normal command prompt and type:

```
mysql -u root -p
```

You will again be prompted for the admin (root) password.

To see the list of databases currently available, type:

```
SHOW DATABASES;
```

On a new MySQL installation, you will see some preinstalled databases such as `information_schema`, `performance_schema` and so on.

Check to see whether the database `workshopdb` exists from a previous lab session. If not, create it with this command:

```
CREATE DATABASE workshopdb;
```

Check that you have created it properly by listing the databases again with:

```
SHOW DATABASES;
```

Select the database for use with:

```
USE workshopdb
```

Check whether the table `users` exists with:

```
SHOW TABLES;
```

If it does, delete it with:

```
DROP TABLE users;
```

We will now create a user account and password and grant full privileges to it for accessing this database:

```
CREATE USER 'spiderman'@'%' IDENTIFIED BY 'peterparker';
GRANT ALL ON workshopdb.* TO 'spiderman'@'%';
```

We can check that the user account has been created successfully with:

```
SELECT user FROM mysql.user;
```

We can check the privileges granted to this account with:

```
SHOW GRANTS FOR 'spiderman'@'%';
```

3 Setting up basic Spring MVC Thymeleaf app

The main folder for this lab is `Security-Extra-Demo`

Start up STS. Switch to the Java perspective.

Go to File -> New -> Spring Starter Project. Complete it with the following details:

Name: `SecurityExtraDemo`
Group: `com.workshop.security`
Artifact: `SecurityExtraDemo`
Version: `0.0.1-SNAPSHOT`
Description: `Additional demo of Spring Security Features`
Package: `com.workshop.security`

Add the following dependencies:
```
Web -> Spring Web
Template Engine -> Spring Thymeleaf
Developer Tools -> Project Lombok
Developer Tools -> Spring Boot DevTools
```

In `src/main/resources/templates`, place these files:

```
add.html
delete.html
home.html
update.html
view.html
```

In the package `com.workshop.security` in `src/main/java`, place these files:

```
WebMvcConfiguration
```

Here, we have created a skeleton structure for simple employee management system accessed via a Thymeleaf app. The configuration file `WebMvcConfiguration` provides a simple direct mapping for the URL paths to the view names, which are Thymeleaf HTML templates.

Start up the app in the usual manner from the dashboard.

Go to http://localhost:8080/ and navigate around the various blank pages.

4 Setting up and persisting model for user info with Spring Data JPA

Add the additional starters to the existing project. Right click on the project, select Spring -> Add Starters:

```
SQL -> Spring Data JPA
SQL -> MySQL Driver
Security -> Spring Security
```

Click Next. Select `pom.xml` in the structure compare menu and click Finish.
Do a Maven -> Update Project.

Replace `application.properties` with the version from `changes`.

In the package `com.workshop.security` in `src/main/java`, place these files:

```
AppUser
AppUserRepository
StartupRunner
```

The info relevant for a principal that can be authenticated as a valid user in the Spring Security framework is modelled in `AppUser`. This basically includes some (or all of the fields) required in `UserDetails`, which is the primary interface that provides core information for a generic user that can be authenticated by the Spring Security Framework (based on the DAO pattern). The most important fields that the implementing class must provide and customize is the username and password.
https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/core/userdetails/UserDetails.html

To allow objects from this class to be persisted, we provide a standard repository interface `AppUserRepository`. This has a single derived query method findByUsername to allow us to retrieve a user record.

We create new objects from AppUser and encode the passwords using the bcrypt algorithm in `StartupRunner`.

Restart the app from the Boot Dashboard in the usual manner.

In a MySQL shell, confirm the contents of the newly generated `users` table with:

```
select * from users;
```

Notice that the password is stored in a hashed form based on the bcrypt algorithm, with additional encoding sequences that Spring Security needs in order to use it effectively for password validation during the authentication phase.

$2a$12$DlfnjD4YgCNbDEtgd/ITeOj.jmUZpuz1i4gt51YzetW/iKY2O3bqa

- "$2a$" The version of BCrypt
- "12$" is the cost factor representing 2^{10} iterations of the key derivation
- The first 22 characters afterwards are the salt of the password
- The rest is the hashed password itself

Notice that if we try to restart the app (without first deleting the existing users table), an exception is thrown. This is because the username field in AppUser is marked as being unique using the @Column annotation, and attempting to store two records with identical values for this fields will result in a `java.sql.SQLIntegrityConstraintViolationException: Duplicate entry` exception

Note: The DevTools starter will automatically restart the app every time we save a change, but you can just ignore this exception as it does not affect the functioning of the app.

5 Setting up authentication from a user database table

In the package `com.workshop.security` in `src/main/java`, delete this file:

`StartupRunner`

Since the database table has already being populated with some sample users from a previous run of the app, we no longer need to use this class anymore.

In `src/main/resources/templates`, place this file:

`login.html`

In the package `com.workshop.security` in `src/main/java`, place these files:

```
MainSecurityConfig
CustomUserDetailService
```

The class CustomUserDetailService provides a customized implementation of the UserDetailsService which will be used later by the authentication provider in `MainSecurityConfig`. Here we simply retrieve a particular record from the MySQL table we just created, and then create a new UserDetails object using the relevant fields from the record via a builder pattern. Any field that we don't set explicitly in UserDetails will have a default value assigned to it.

In `MainSecurityConfig`, we configure the DAOAuthenticationProvider using our customized implementation that is autowired. This is subsequently used to configure the main AuthenticationManagerBuilder. In the configure method that we normally use for enforcing authorization rules, we simply specify that all requests be authenticated and also specify a customized login form that everyone can access in order to login.

Restart the app.

Access the app homepage at http://localhost:8080/

Verify that when you click on any of the links to navigate to a different URL, you will be presented with the custom login page. Once you login with any of the users that we initially created earlier, you should be able to navigate through all the remaining pages without any issue. Notice that the server side log shows the specific record with a matching username retrieved from the table by Spring Security in order to perform password validation in this authentication phase.

As usual, clearing the session cookie will again force another login challenge as usual.

Notice that the original password is never available in plaintext form, either in `MainSecurityConfig` or in the database table. The only time we could see it was when we hardcoded the user info details in the previous lab session. Once we add in a registration form to allow a user to register their user name, password and other related information properly, the plaintext password will no longer be available for viewing at any point during the app construction process or execution flow.

This is absolutely critical for ensuring the safety of the web app, particularly if the user accounts protected with passwords have extensive privileges in the system.

6 Password encoders and delegating password encoder

In a MySQL client shell, delete the existing users table with:

```
drop table users;
```

In the package `com.workshop.security` in `src/main/java`, create a new file based on this:

```
StartupRunner-v2
```

Add in the following Bouncy Castle dependency to allow us to use the following PasswordEncoder instances: (`SCryptPasswordEncoder`, `Argon2PasswordEncoder`) in our code:

```
<dependency>
	<groupId>org.bouncycastle</groupId>
	<artifactId>bcprov-jdk15on</artifactId>
	<version>1.68</version>
</dependency>
```

You can select an appropriate recent version to use:
https://mvnrepository.com/artifact/org.bouncycastle/bcprov-jdk15on

Spring 5 mandates that all passwords that are configured for authentication must be in an encoded form using any one of the password hashing algorithms that we have seen earlier that are based on KDF such as Argon2, Bcrypt, Scrypt. We have concrete implementations for all of this:

- `org.springframework.security.crypto.argon2.Argon2PasswordEncoder`
- `org.springframework.security.crypto.scrypt.SCryptPasswordEncoder`

- `org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder`

Spring 5 additionally introduces the concept of a DelegatingPasswordEncoder, which provides support for the following features:

a) Encode passwords for multiple users using different encoding algorithms and validate these easily and correctly
b) Encode password using the latest encoding standard recommendations
c) Support for a smooth upgrading of the current encoding option in the future

When we use password encoding delegation, each generated encoded password is preceded with an `{id}` tag that helps identify the specific encoder instance that was used to encode it, in order so that the identical encoder instance is used to match passwords during the authentication stage. We then store the encoded password with its `{id}` tag in memory or in a database table.
Subsequently, we will use a DelegatingPasswordEncoder during authentication which can determine the correct password encode instance to use for matching passwords based on the `{id}` tag of the retrieved encoded password.

https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/crypto/password/DelegatingPasswordEncoder.html

The PasswordEncoderFactories class provides a default DelegatingPasswordEncoder that already includes a mapping between standard `{id}` prefixes and their matching PasswordEncoder instances. The default DelegatingPasswordEncoder is also preconfigured with BCryptPasswordEncoder as the default PasswordEncoder instance.
https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/crypto/factory/PasswordEncoderFactories.html

Restart the app (ignore any warning messages if restarted too many times)

In a MySQL client shell, check the contents of the newly created users table with:

```
select * from users;
```

Notice that the encoded passwords are still in the standard bcrypt format but are now preceded with the `{brcypt}` id to identify them.

Next, we will reconfigure our MainSecurityConfig to authenticate using encoded passwords in this new format:

In the package `com.workshop.security` in `src/main/java`, make the following change:

`MainSecurityConfig-v2`

Also delete the following file as we no longer need to use it:

`StartupRunner`

Restart the app and verify that it performs exactly as it did before with regards to authenticating a valid user.

Lets assume that we now wish to encode the user passwords using two additional hashing algorithms: scrypt and argon2, as well as the default of bcrypt.

In a MySQL client shell, delete the existing users table with:

```
drop table users;
```

In the package `com.workshop.security` in `src/main/java`, create a new file based on this:

```
StartupRunner-v3
```

Restart the app and check the contents of the newly created users table with:

```
select * from users;
```

Notice that the passwords for all 4 users are encoded using different encoders, with the prefix `{id}` identifying the specific encoding algorithm that was used.

In the package `com.workshop.security` in `src/main/java`, delete this file:

```
StartupRunner
```

In the package `com.workshop.security` in `src/main/java`, make the following changes:

```
MainSecurityConfig-v2
```

In `src/main/resources/templates`, make the following changes:

```
home-v2
```

This simply introduces an additional logout button and configures MainSecurityConfig to implement the logout functionality to invalidate the existing session to simplify testing logging in with all these 4 4 different usernames and passwords

Restart the app. Verify that you can login and logout successfully using the 4 different usernames and passwords that we created earlier.

This demonstrates that using a single DelegatingPasswordEncoder for the authentication phase which has mappings for all the standard encoding algorithms in the Spring Security framework is all that we need to perform password matching for passwords encoded with any of these algorithms during the authentication phase.

```
@Bean
public PasswordEncoder passwordEncoder() {

   return PasswordEncoderFactories.createDelegatingPasswordEncoder();

}
```

Another possible variation on this theme of customizing encoders is utilizing different instantiations of the same encoder instance using different constructor parameters in order to fine tune the security of the encoded password (i.e. how resistant it is to a rainbow attack or brute force attack).

Consider BCryptPasswordEncoder
https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/crypto/bcrypt/BCryptPasswordEncoder.html
We might consider that there are 2 possible instances of BCryptPasswordEncoder that we could use - a "weak" and a "strong" instance:

- "weak" - strength of 4 (min possible)
- "strong" - strength of 31 (max possible) with an additional SecureRandom instance specified

Similarly for Pbkdf2PasswordEncoder
https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/crypto/password/Pbkdf2PasswordEncoder.html
We might consider that there are 2 possible instances of Pbkdf2PasswordEncoder that we could use - a "weak" and a "strong" instance:

- "weak" - iteration of 5000, salt length of 8, hash width of 128 (sample small values for each parameter)
- "strong" - iteration of 200000, salt length of 32, hash width of 512 (sample large values for each parameter)

For more detailed discussion on the exact parameters to use for PBKDF2, see:
https://security.stackexchange.com/questions/110084/parameters-for-pbkdf2-for-password-hashing
https://security.stackexchange.com/questions/17994/with-pbkdf2-what-is-an-optimal-hash-size-in-bytes-what-about-the-size-of-the-s

In a MySQL client shell, delete the existing users table with:

```
drop table users;
```

In the package `com.workshop.security` in `src/main/java`, create a new file based on this:

```
StartupRunner-v4
```

Restart the app and check the contents of the newly created users table with:

```
select * from users;
```

Now try to interact with the app again its homepage at http://localhost:8080/

Notice that this time you will not be able to authenticate successfully (even with the correct username / password combination), this is because there is no matching for the custom ids that we used in our original encoding operations (i.e. `pbkdf2-weak`, `pbkdf2-strong`, etc). You can verify this from the exception thrown from the server side:

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

To allow authentication to work properly, we must now reuse the custom DelegatingPasswordEncoder that we crated in the initial encoding of passwords in `StartupRunner`

In the package `com.workshop.security` in `src/main/java`, delete this file:

`StartupRunner`

In the package `com.workshop.security` in `src/main/java`, make the following changes:

`MainSecurityConfig-v3`

Now try to interact with the app and verify that you are able to login and logout successfully using the 4 different usernames and passwords.

Notice the difference in login delay between the usernames encrypted with the strong and weak versions of PBKDF2 respectively. The significantly increased time for login with the strong version is due to the increased number of iterations necessary to compute the hash during the password validation step. This becomes an important deterrent against rainbow table and brute force attacks.

On the other hand, having such a long delay in validating passwords makes the system vulnerable to possible DOS attacks, where an attacker floods the system with a large number of bogus login requests that slows down the system to the point where it is not able to effectively login a valid user within an acceptable period of time. We then have to incorporate specific mechanisms to protect against this, such as locking or disabling a user account after a certain number of unsuccessful login attempts or blacklisting IP addresses where a large amount of login traffic is originating from.

In terms of deciding which PasswordEncoder to use, here is a useful discussion on this topic:
https://security.stackexchange.com/questions/193351/in-2018-what-is-the-recommended-hash-to-store-passwords-bcrypt-scrypt-argon2

7 Using Spring Security Tags for Thymeleaf

Thymeleaf provides a separate Spring security integration module to allow the incorporation of security related expressions in the standard Thymeleaf dialect.

To use it, first add in this managed dependency in the project POM. Remember to perform a Maven -> Update project when you are done.

```
        <dependency>
```

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

In `src/main/resources/templates`, make the following changes:

`add-v2`
`home-v3`

Restart the app.

Login with any username and verify that this is displayed correctly along with the corresponding role. Logout and verify that the username and role are no longer visible at the home page.

Login with usernames corresponding to the ADMIN and USER roles, and then navigate to the Add page. Verify that the correct messages are displayed based on the specific role of the currently authenticated user.

8 Customizing authorization at method level

Spring Security supports authorization semantics at the method level. It allows the securing of the service layer by, for example, restricting which roles are able to execute a specific method.

In the package `com.workshop.security` in `src/main/java`, add the following files:

`MethodSecurityConfig`
`EmployeeController`

In the package `com.workshop.security` in `src/main/java`, make the following changes:

`WebMvcConfiguration-v2`

We provide configuration for method level authorization via the @EnableGlobalMethodSecurity in MethodSecurityConfig, specifying all the authorization method security option attributes.

We move the mapping of the URL paths to view names from `WebMvcConfiguration` to `EmployeeController` to allow us to demonstrate the use of the @Secured annotation.

Restart the app

Try logging in and logging out with different usernames. Navigate to the various paths from the home page and observe which ones return a 403 Forbidden Whitelabel error page. Verify that this matches with the @Secured annotation on the respective methods in `EmployeeController`

We can provide a customized error page to replace the Whitelabel error page.

In `application.properties` place in this additional property:

```
server.error.whitelabel.enabled=false
```

In `src/main/resources/templates`, create a new folder `error`

In this folder, place this new file:

`403.html`

Restart the app

Try logging in and logging out with different usernames. Navigate to the various paths from the home page and verify that attempting to navigate to a page which the currently authenticated user does not have authorization to access observe returns the customized `403.html`

In the package `com.workshop.security` in `src/main/java`, make the following changes:

`EmployeeController-v2`

We can use the @PreAuthorize and @PostAuthorize annotations provide expression-based access control. The @PreAuthorize annotation checks the given expression before entering the method, whereas the @PostAuthorize annotation verifies it after the execution of the method.

Thus for the case of @PreAuthorize, if the given expression does not evaluate to true, the body of the method is not executed at all. On the other hand with @PostAuthorize will always result in the execution of the method body with an authorization exception thrown at the end if the evaluated expression is not true. For the case of @PostAuthorize, we can even use the result returned from the method as part of the expression to be evaluated, for e.g.

```
@PostAuthorize("returnObject == 'update'")
```

Restart the app

Try logging in and logging out with different usernames. Navigate to the various paths from the home page and verify the effect of the @PreAuthorize and @PostAuthorize annotations by checking the log output from the server.

Modify the security expression for `updatePage` below so that it returns false

```
@PostAuthorize("returnObject == 'update'")
```

and then verify that the customized error page is displayed when navigating to the corresponding path.

We will further build up the app to demonstrate the use of the @PreFilter and @PostFilter annotations to filter lists of objects based on custom security rules.

In the package `com.workshop.security` in `src/main/java`, place these new files:

`Hero`

`IHeroService`
`HeroService`

In the package `com.workshop.security` in `src/main/java`, make the following changes:

`EmployeeController-v3`

In `src/main/resources/templates`, make the following changes:

`view-v2`
`add-v3`
`home-v4`

A Hero model has been created that contains the following fields:

- a name that matches the username of the AppUser model, which is used for storing user info for purposes of authentication).
- a list of Strings that represent the various powers that a particular Hero possesses

`HeroService` implements `IHeroService` interface in order to provide functionality to retrieve the existing list of heroes as well as add new powers to the heroes in this list. Its constructor instantiates with a preconfigured list of heroes (whose names match the usernames of the 4 users currently stored in the password database).

The UI of the app has been modified to reflect this change.

At this point, no authorization rules are configured anywhere.

Restart the app.

Login in with any username. View the list of existing heroes, try adding new powers to any of them and checking the list again. Verify that the results are as expected.

In the package `com.workshop.security` in `src/main/java`, make the following changes:

`IHeroService-v2`
`HeroService-v2`

Here we add in @PostFilter and @Prefilter annotation to these two methods in `IHeroService` which essentially limits the retrieval of the heroes list as well as any additions of powers to them to users that have the ADMIN role.

Note that the initial operation of filtering on the list of objects returned from a method annotated with @PostFilter will truncate that list to the filtered result. Thus, if that list is representative of some data structure that must be persisted in memory through the lifetime of the app (such as `myHeroes` in this example), make sure you return a copy (a shallow copy is adequate) rather than the list itself.

Restart the app.

Login with any username with an assigned ADMIN role. Verify that the results are exactly as was the case prior to these annotations being introduced. Then login with any other username that does not have the ADMIN role. Notice now that you are not able to view any heroes and any new powers you specify to be added for any existing hero will not take effect.

In the package `com.workshop.security` in `src/main/java`, make the following changes:

`IHeroService-v3`

We now add in an additional condition in the @PreFilter and @PostFilter so that for users that do not have ADMIN role, they will only be able to view their respective powers and set new powers for themselves only. The filtering by matching on name accomplishes this effect. Users with ADMIN role are still able to view all heroes and their corresponding powers, as well as set new powers for any hero.

Restart the app.

Login alternatively with users with USER and ADMIN roles. Verify the effect just described.

9 Session management

The Spring security framework provides different options for controlling and customizing sessions, including configuring when the session will be created and how users interact with the session. The primary options available for configuration

a) `SessionCreationPolicy.ALWAYS` – A session will always be created (if it does not exist).
b) `SessionCreationPolicy.IF_REQUIRED` – Spring Security will only create a HttpSession if required (this is the default configuration). For form based login authentication, the default of `SessionCreationPolicy.IF_REQUIRED` is adequate for the vast majority of cases.
c) `SessionCreationPolicy.NEVER` – Spring Security will never create a HttpSession, but will use a HttpSession if it already exists (made available through application server)
d) `SessionCreationPolicy.STATELESS` – Spring Security will never create a HttpSession or use one even if it is made available by the application server

Before making the next change, restart the app again and verify

- The home page and add page is able to display the authenticated user's identity properly
- If you logout, and then manually type in the URL for the view page: http://localhost:8080/view in the address bar, you will be redirected to the customized login page and once you login successfully you will be redirected to the view page.

In the package `com.workshop.security` in `src/main/java`, make the following changes:

`MainSecurityConfig-v4`

To change the session creation policy in Spring security, we add to the builder chain for HTTPSecurity by adding `sessionManagement()`. Here we set the policy to STATELESS to see the effect it has on our app.

Set up Eclipse TCP/IP monitor to listen on port 9090 (or some other appropriate free port) and redirect to the port that the app is running on (8080).

Navigate to http://localhost:9090/ by typing this into the address bar. Notice that the JSESSIONID is still being submitted from the browser, but the previously authenticated user's identity is not displayed on the page. This is because even though session cookies are still sent by the underlying application server (the embedded Tomcat server that the app is deployed in), this cookie is not used by Spring Security to track the user identity.

Click on any of the links on the home page. You will be redirected to the customized login page, and after logging in successfully you are redirected back to the home page again, and not to the link that you were attempting to navigate to. This is again due to the lack of session tracking whereby Spring Security does not keep track of the page that you were attempting to navigate to prior to authenticating successfully.

Instead, it will just simply redirect to the configured default URL path (/) on every successful login attempt. The redirection to the customized login page, and then redirection back to the home page repeats endlessly, making the app essentially unusable due to no session tracking being performed.

When STATELESS is configured, we effectively disable all session tracking by Spring Security and delegate the responsibility for any kind of session tracking to the app. It will need to explicitly implement session tracking by manually accessing and manipulating the session cookies that are produced by the Tomcat server and returned by the browser in the HTTP interaction. Such a situation is only called for in very special cases where we need to perform some kind of customized session handling that differs from the way that Spring handles it.

Session timeout is another useful feature to implement from a security viewpoint. If the server does not receive a request from the browser after a certain period of time, it would be useful to automatically terminate the session, forcing the user to login in again if they need to access any particular resource that is configured to require authentication. This is very useful for situations when the user is working at a publicly accessible machine and forgets to logout from their account (which is necessary to terminate the session). This means the next user of that machine can still continue to access sensitive resources on that user's account.

The default session timeout for Spring Security is determined by the default for the underlying application server (for e.g. Tomcat has a default time out of 30 minutes). Note that if we are using Tomcat as the deployment server, timeout is only supported to minute precision, with a minimum of one minute. This means that if we specify a timeout value of 170s for example, it will result in a 2 minutes timeout.

We can specify a new timeout of 1 minute (the minimum) in `application.properties`

```
server.servlet.session.timeout=1m
```

In the package `com.workshop.security` in `src/main/java`, we will change back to the previous version of MainSecurityConfig where session is maintained properly :

`MainSecurityConfig-v3`

Restart the app.

Login with any username. Navigate to the view page and then return back to the home page.
Wait for slightly longer than a minute, then attempt to navigate to the view page again. Notice that this time you will be redirected to the customized login page as the current session has timed out.

We can also specify a specific URL to redirect to in the event that a session cookie is received with an incoming request that references a timed-out session. We would also need to ensure that this URL has its authorization set to `permitAll()` since all users redirected to this page would be unauthenticated.

Typically, we would still want the user to be redirected to the customized login page, but it would be useful to include an additional message to indicate the reason for being forced to login in. We can use a specific query parameter as a marker for this case (similar to the way we have done for identifying unsuccessful login or successful logout at the customized login page).

In the package `com.workshop.security` in `src/main/java`, make the following changes:

`MainSecurityConfig-v5`

In `src/main/resources/templates`, make the following changes:

`login-v2`

Restart the app.

Login with any username. Navigate to the view page and then return back to the home page.
Wait for slightly longer than a minute, then attempt to navigate to the view page again. Again, you will be redirected to the customized login page with a customized error message.

The primary problem with this approach is that performing a logout will also redirect back to the custom login page with the same error as a timeout. This is because a logout essentially invalidates the session, resulting in the redirection to the path indicated by `invalidSessionUrl()`

We need a way to distinguish between a session time out and a logout at the point when the login page is being displayed since both actions will invalidate the current session. One possible way is to provide a custom LogoutHandler to implement redirection to the login page with the correct query parameter (`logout=true` rather than `timeout=true`), but we can also implement a simpler solution.

In the package `com.workshop.security` in `src/main/java`, make the following changes:

`MainSecurityConfig-v6`
`EmployeeController-v4`

In `src/main/resources/templates`, make the following changes:

`login-v3`

What we can do is to simply track the time at which a request is made to any of the valid mapped paths for this application. Then, when the login page is retrieved again, we calculate the time lapsed since the last request. If this time is more than the session time out (retrieved from the HTTPSession

object), then we can deduce that the login page was retrieved due to a session timeout, otherwise it was retrieved due to a logout.

Restart the app.

Login with a valid username and logout before the session timeout expires. Check the message on the login page.
Login again and wait until the session time out expires, then click on any of the links in the home page. Again, verify the correct message appears in the login page that is returned.

There are certain situations where we want to limit multiple concurrent logins for the same user from a security viewpoint. It's also useful for SaaS that is based on a subscription model that bills on a per-user basis (typically licenses are sold on the basis of allowing a maximum number of users accessing the service at any one time). When a user has being authenticated and attempts to re-authenticate without logging out first, the application can be configured to respond in one of the following ways:

a) Invalidate the existing session and create a new authenticated session.
b) Keep exiting session and display an error message for the new login attempt.
c) Allow both sessions to run concurrently

Spring Security provides the ability to limit multiple logins for the same user. To enable this feature, we add the HttpSessionEventPublisher listener via a @Bean factory method and also configure the maximum number of sessions that can be open at any one time.

In the package `com.workshop.security` in `src/main/java`, make the following changes:

`MainSecurityConfig-v7`

We have also added in the specification for the number of `maximumSessions` for a given user in the builder chain for HttpSecurity

Restart the app.

Login with any username. Navigate to any page.

Using a different browser from the one that you used for previous login, login again with the same username. Navigate to any page.

Return to the first browser that you logged in with. Attempt to navigate to any page. You should see an error message:

```
This session has been expired (possibly due to multiple concurrent
logins being attempted as the same user).
```

If you refresh the page again, you will be redirected again to the login page. If you now login successfully on the current browser, and attempt to navigate to a different link on the other browser, the same issue will arise again. This ensures that only a maximum of one active session can be established with that particular username at any one time with the app.