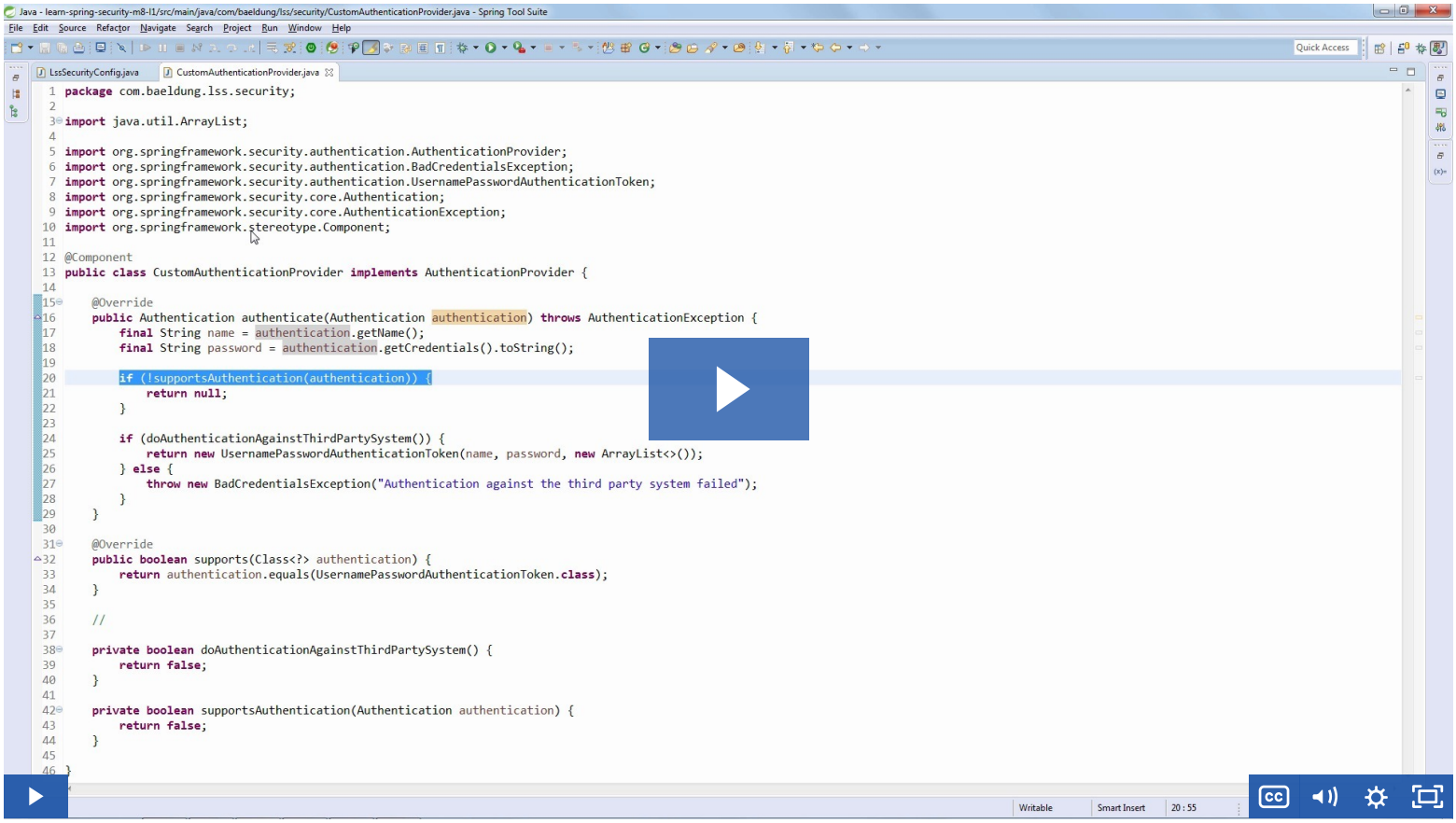
Lesson 1: A Custom Authentication Provider



1. Goals

The goal of this lesson is to guide you through setting up a new, custom authentication provider.

2. Lesson Notes

As you're using the git repository to get the project - you should be on the module8 branch.

Or, on the corresponding Spring Boot 2 based branch: module8-spring-boot-2.

The relevant module you need to import when you're starting with this lesson is: m8-lesson1

If you want to skip and see the complete implementation, feel free to jump ahead and import: m8-lesson2

The credentials used in the code of this lesson are *test@email.com/pass* (PostConstruct)

By default, the main authentication provider is going to be the *DaoAuthenticationProvider*.

We're going to now roll our own provider - which simply authenticates against a third-party system. This is going to replace the default provider.

First, let's discuss **the contract of the *authenticate* method** in the provider:

- if authentication succeeds, a full *Authentication* object (with credentials) is expected as the return
- if the provider doesn't support the *Authentication* input, it will return *null* (and the next provider will be tried)
- if the provider does support it and we attempt authentication and fail - *AuthenticationException*

When we're talking about the exception, it's important to understand that is the base class for many, many more specific exceptions.

In a production implementation - for a real integration with a third party authentication service - where you'll have a lot more info to work with. So, we'll need to throw very specific exceptions based on the actual problem that occurred.

Let's now s wire in this new custom auth provider with Java config:

```
@Autowired private CustomAuthenticationProvider customAuthenticationProvider;
...
auth.authenticationProvider(customAuthenticationProvider);
```

Now, when we start up the system and check the providers - the parent auth manager is the same but the the child manager is now using this new provider as expected.

Also note that - starting with Spring Security 4.1 - you can also define a custom authentication provider without explicitly wiring it in - by just defining it as a new bean.

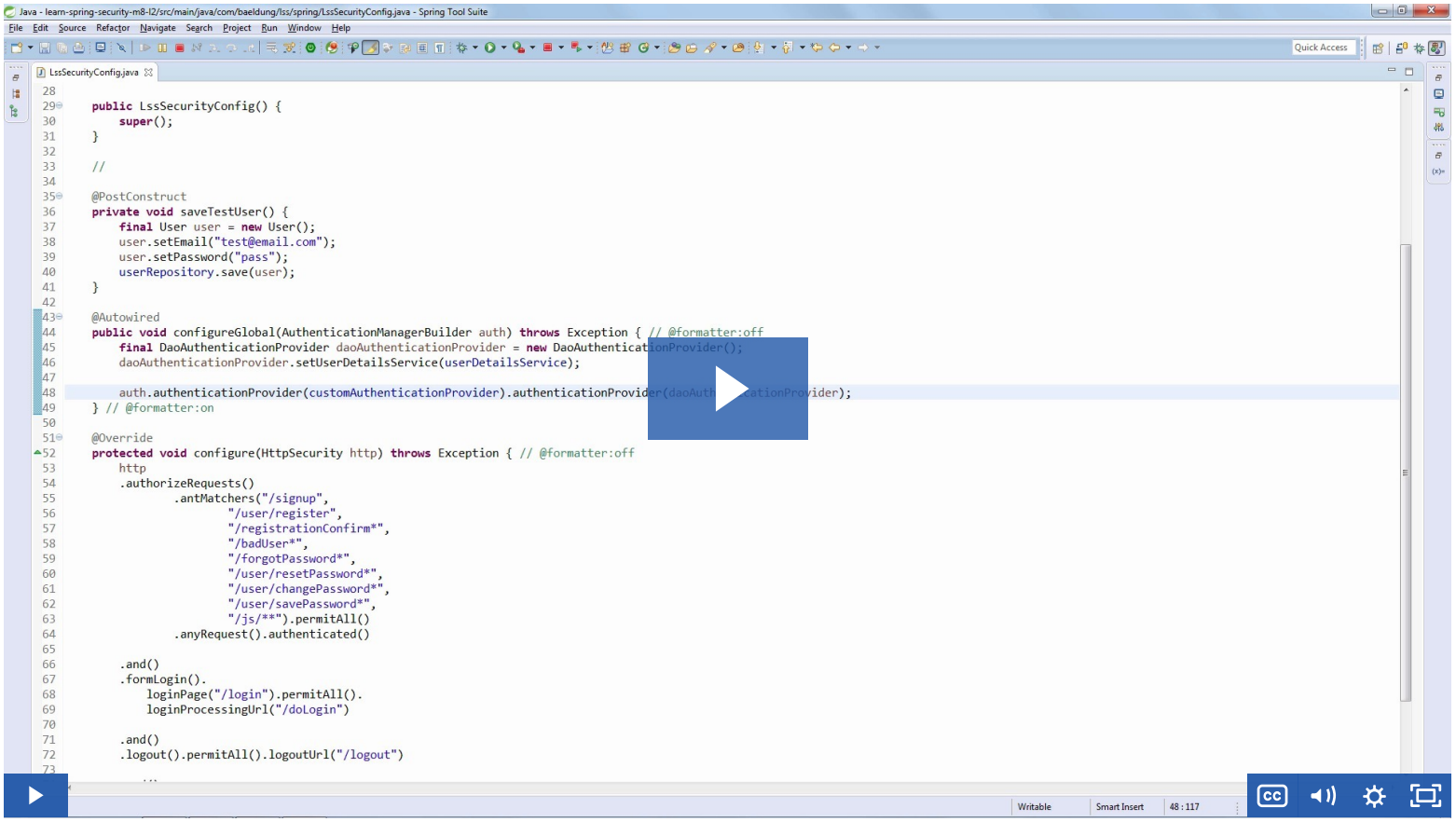
3. Resources

- Using other Authentication Providers with XML configuration

- The AuthenticationManager, ProviderManager and AuthenticationProvider

- Spring Security Authentication Provider

Lesson 2: Multiple Providers and the Authentication Manager



1. Goals

The goal of this lesson is to go deeper into the core authentication artifacts and **set up multiple providers** within the authentication manager.

2. Lesson Notes

The relevant module you need to import when you're starting with this lesson is: m8-lesson2

If you want to skip and see the complete implementation, feel free to jump ahead and import: m8-lesson3

The credentials used in the code of this lesson are *test@email.com/pass* (PostConstruct)

Moving forward from the previous lesson, we're going to start with a more complex authentication scenario.

We're going to have multiple ways to authenticate - a primary way (the third party system) and a fallback (our own DB).

In order to achieve that, we'll need multiple authentication providers.

2.1. Multiple Auth Providers

So, let's define a second provider here into our config (*configureGlobal*):

```
DaoAuthenticationProvider daoAuthProvider = new DaoAuthenticationProvider();
daoAuthProvider.setUserDetailsService(userDetailsService);
auth.authenticationProvider(daoAuthProvider).authenticationProvider(customAuthenticationProvider);
```

So, we now have 2 providers wired into our auth manager.

2.2. A New Auth Manager

There are very rare instances where we need to define an auth manager bean.

That doesn't mean actually rolling your own implementation (you'll almost never do that) - we're simply going to instantiate the *ProviderManager*.

And remember - if an auth manager cannot authenticate, it goes to the parent.

So this is basically a chance to change that hierarchy and plug in a sort of a fallback auth manager (the top parent) - in case all others fail:

```
auth.parentAuthenticationManager(
  new ProviderManager(Lists.newArrayList(customAuthenticationProvider)));
```

2.3. Configure the Auth Manager

And now that we're defining the auth manager - we can also configure it explicitly.

By default, the *ProviderManager* will clear sensitive credentials information from the *Authentication* object (which is returned by a successful authentication request). This prevents information like passwords being retained longer than necessary.

In some rare cases, we need to change that - for example, say we're storing these authentication objects into a cache (maybe they're expensive to get back). We can disable the clearing of credentials.

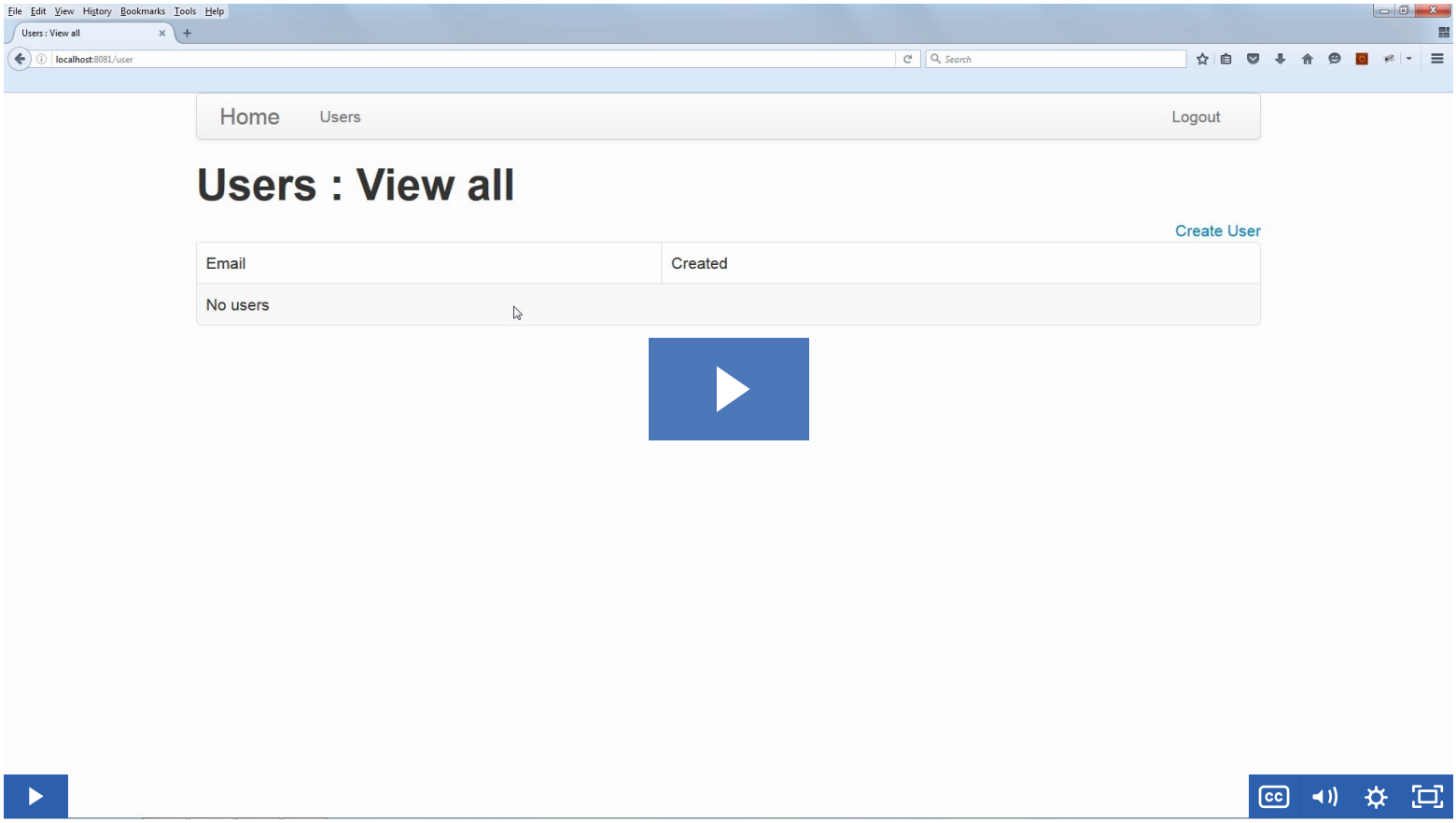
But keep in mind that the config API allows that without messing with the actual manager bean:

```
auth.eraseCredentials(false).userDetailsService(userDetailsService);
```

3. Resources

- The AuthenticationManager, ProviderManager and AuthenticationProvider

Lesson 3: In-Memory, JDBC and Hibernate/JPA User Storage



1. Goals

The goal of this lesson is to guide you through **implementing user storage with Spring Security** - from a simple, in-memory solution to a JPA backed solution.

2. Lesson Notes

The relevant module you need to import when you're starting with this lesson is: m8-lesson3

If you want to skip and see the complete implementation, feel free to jump ahead and import: m8-lesson4

The credentials used in the code of this lesson are *test@email.com/pass* (PostConstruct)

2.1. In-Memory User Storage

The first thing we're going to set up authentication with a simple in-memory user solution.

```
auth.inMemoryAuthentication().
  withUser("test@email.com").password("pass").roles("USER");
```

Note that this is what we've been using in earlier modules, so you should already be familiar with it.

You can also define multiple users via this fluent config API:

```
auth.inMemoryAuthentication().
  withUser("test@email.com").password("pass").roles("USER").and().
  withUser("test2@email.com").password("pass2").roles("ADMIN");
```

2.2. JDBC User Storage

Now, let's look at JDBC - we'll first need a datasource.

And because the codebase already uses persistence, **we have a datasource available**; we can simply wire that in and use it:

```
auth.
  jdbcAuthentication().dataSource(dataSource).withDefaultSchema().
```

And the API for defining users is exactly the same:

```
withUser("test@email.com").password("pass").roles("USER");
```

2.3. JDBC with MySQL

If we want to use MySQL, the *withDefaultSchema()* API won't work unfortunately (the script isn't following MySQL syntax).

We'll have to set up our own schema structure:

```
create schema if not exists lss83;
USE lss83;
create table users(
  username varchar(50) not null primary key,
  password varchar(500) not null,
  enabled boolean not null
);
create table authorities (
  username varchar(50) not null,
  authority varchar(50) not null,
  constraint fk_authorities_users foreign key(username) references users(username)
```

```
);
create unique index ix_auth_username on authorities (username,authority);
```

With the new schema, we don't need *withDefaultSchema()* any longer.

And a quick side-note - we'll set up the schema manually but there are of course tools that can be used for that when the application starts up.

2.4. Non-Standard DB Structure

Finally, if we need to adhere to a pre-existing and non-standard DB structure - the configuration allows use to set up these scripts:

```
.usersByUsernameQuery( ... )
.authoritiesByUsernameQuery( ... )
```

And basically use our own structure instead of the default one - when doing these two operations.

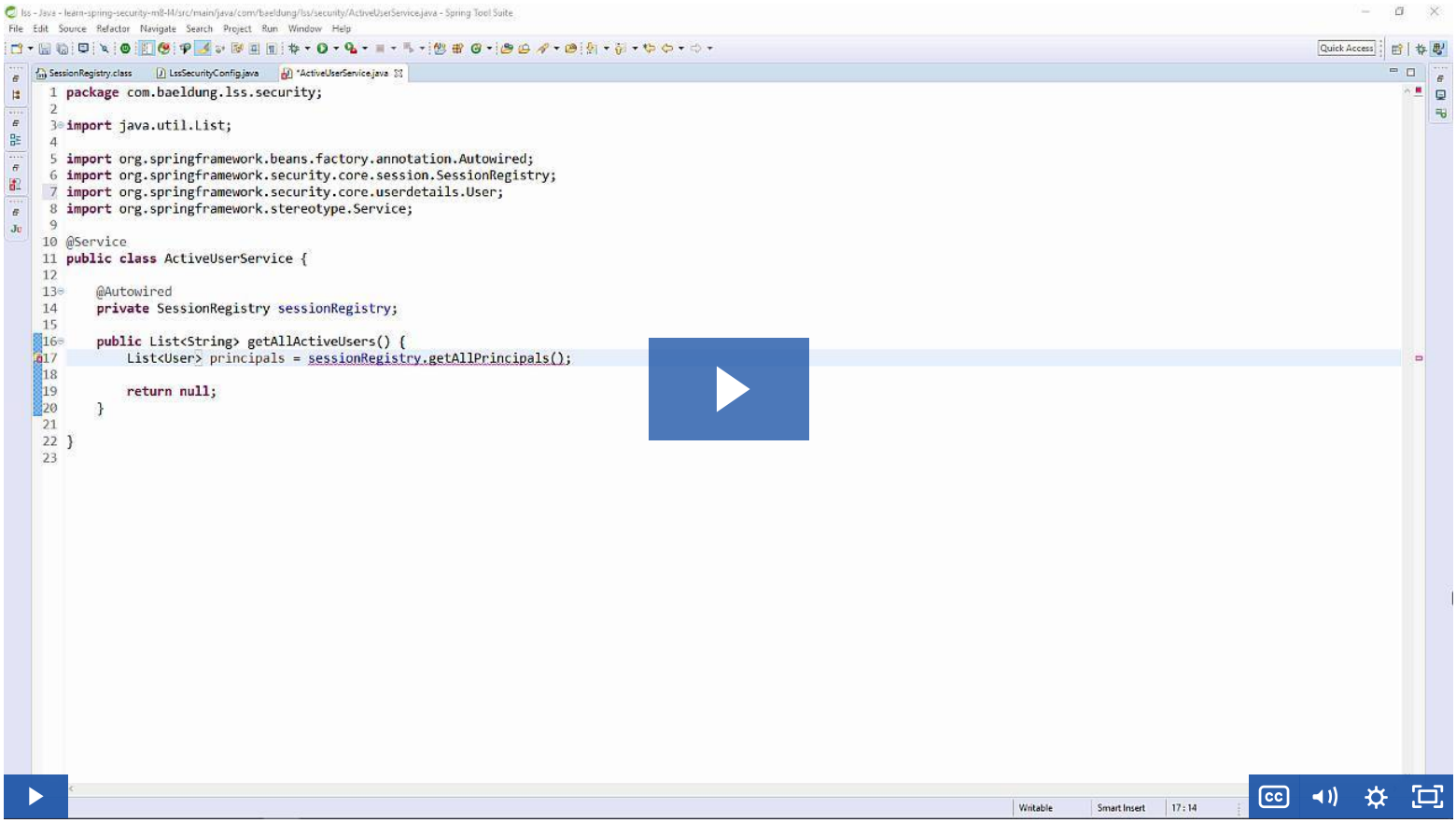
2.5. JPA User Storage

Using JPA for user storage is simple - mainly because **that what we've been doing** when we were using our custom user details service. That implementation was based on Spring Data JPA - so all we need to do here is simply wire that back in.

3. Resources

- Authentication (in the Spring Reference)

Lesson 4: Tracking Logged-in Users (NEW)



1. Goals

The goal of this lesson is teach you how to track the currently authenticated users in a Spring Security application.

2. Lesson Notes

The relevant module you need to import when you're starting with this lesson is: m8-lesson4

If you want to skip and see the complete implementation, feel free to jump ahead and import: m8-lesson5

The credentials used in the code of this lesson are *test@email.com/pass* (PostConstruct)

2.1. The SessionRegistry

This class manages users and sessions; it has a simple API:

- *getAllPrincipals* - to obtain a list of users
- *getAllSessions* - obtain a list of session for a user

First, let’s make sure we register this session registry into our security configuration, so we can use it later on:

```
.and().sessionManagement().maximumSessions(1).sessionRegistry(sessionRegistry())
```

Note that you’ll need to close out the configuration here by also defining session fixation:

```
.and().sessionFixation().none()
```

2.2. Keeping Track of Users

Next, let’s define our primary API for these operations - the ActiveUserStore:

```
@Service
public class ActiveUserService {
    @Autowired
    private SessionRegistry sessionRegistry;
    public List<String> getAllActiveUsers() {
        // TODO: implement
        return null;
    }
}
```

Now, let’s get the currently active users out of the session registry:

```
List<User> principals = sessionRegistry.getAllPrincipals();
```

Notice that we have a problem - the API is giving us a list of objects, and we need a list of Spring User objects.

We know these are actually Spring Users, because we’re using our own user details service - which is actually creating that particular object.

Because there's no good way to cast a List<Child> to a List<Parent>, we'll go through an array here:

```
List<Object> principals = sessionRegistry.getAllPrincipals();
User[] users = principals.toArray(new User[principals.size()]);
```

And with that, we can now create a stream out of it and finally build our logic:

```
Arrays.stream(users)
```

Let’s now filter by using the session registry again and only picking up the active users:

```
.filter(u -> !sessionRegistry.getAllSessions(u, false).isEmpty())
```

Let’s just get the username from the user - as we don’t need anything else here:

```
.map(u -> u.getUsername())
```

And finally, let’s collect:

```
.collect(Collectors.toList());
```

And that’s it - the list of usernames for all authenticated users in the system

2.3. The Controller

Finally, let’s wire that into the controller layer so that we can make sure things actually work on the front-end; we're going to be working in the UserController:

```
@Autowired
private ActiveUserService activeUserService;
```

We’re going to use this active service to get the usernames:

```
activeUserService.getActiveUsers()
```

Now, because this particular method is returning Users not Strings, let’s wrap up the usernames in User objects:

```
.stream().map(s -> new User(s)).collect(Collectors.toList());
```

And that's it - - we’re now returning only the live users here, not the full userbase.

3. Resources

- Keep Track of Logged In Users with Spring Security

- SessionRegistry javadoc