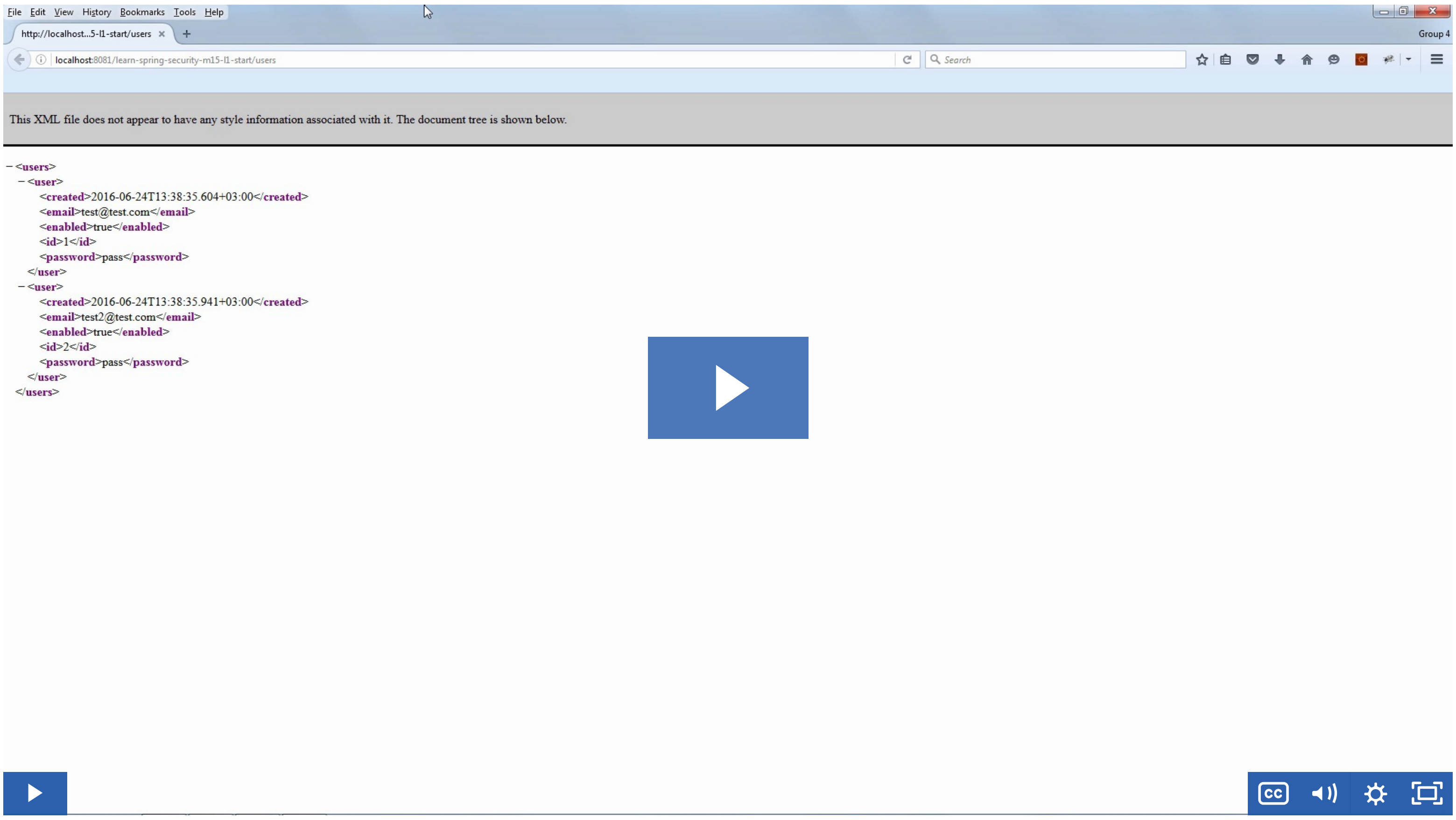
Lesson 1: Spring Security for a non-Spring Application


File Edit View History Bookmarks Tools Help
http://localhost...5-l1-start/users
localhost:8081/learn-spring-security-m15-l1-start/users
Search
Group 4

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
- <users>
  - <user>
      <created>2016-06-24T13:38:35.604+03:00</created>
      <email>test@test.com</email>
      <enabled>true</enabled>
      <id>1</id>
      <password>pass</password>
    </user>
  - <user>
      <created>2016-06-24T13:38:35.941+03:00</created>
      <email>test2@test.com</email>
      <enabled>true</enabled>
      <id>2</id>
      <password>pass</password>
    </user>
  </users>
```

1. Goal

The simple goal of this lesson is to show you how to use Spring Security to **secure a non-Spring web application**.

2. Lesson Notes

The relevant module you need to import when you're starting with this lesson is: m15-lesson1-start

If you want to skip and see the complete implementation, feel free to jump ahead and import: m15-lesson1

You'll notice the naming of the modules in this lesson is a bit different - we now have 2 projects over in the repository, dedicated to this lesson.

So, if you're following along with the lesson building your implementation as well - go ahead and start with the *_start* project.

2.1. The non-Spring App

Up until this point, we've been only working fully within the Spring ecosystem.

But of course - we don't have to. There is a lot of flexibility built into Spring Security, and there's nothing that requires us to only work with Spring apps.

In this lesson, we're going to be **securing a non-Spring project with Spring Security.**

Let's start looking at our non-Spring, non-secured project.

This is a very simple REST API, built with JAX-RS (and EJBs), and exposing a User resource. So, it's similar to our previous application - which was also managing users.

The implementation is of course very simple and not the focus here. The focus is of course going to be how we're going to secure this application, not the application itself.

So, the structure of the app is what you might expect:

- we have the Entity
- a simple DAO with JPA - that does some basic persistence
- a Service layer and
- the API/Controller layer

The only operation we're exposing is the get all users - so, clearly very simple.

2.2. Running the App

Let's run the non-secured app and let's consume it from the client side.

We're going to be using **a TomEE version 7** to run this API. This is of course not optional, so make sure you don't run it in a standard Tomcat server.

Also note that - if you want to add the TomEE server into Eclipse (you don't have to) - there's unfortunately an open bug that makes it a bit more difficult to do that. There's a workaround here which works until the real fix is in.

And note that, on startup, a *StartupListener* creates 2 test users.

After deploying the system, we can access it at:

`http://localhost:8081/learn-spring-security-m15-l1-start/users`

And that's it - we are now exposing these 2 users resources, using an XML representation

Now, let's secure the application.

2.3. Securing the App

OK, so let’s now start setting up Spring Security. We first need to add the dependencies to our pom:

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</artifactId>
    <version>4.1.0.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
    <version>4.1.0.RELEASE</version>
</dependency>
```

We then need to define our simple security configuration:

```
@EnableWebSecurity
@Configurationpublic
class LssSecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception { // @formatter:off
        auth.inMemoryAuthentication().withUser("test@test.com").password("pass").roles("USER");
    } // @formatter:on
    @Override
    protected void configure(HttpSecurity http) throws Exception { // @formatter:off
        http.authorizeRequests().anyRequest().authenticated().and().httpBasic();
    } // @formatter:on
}
```

Notice that we’re simply doing basic authentication; and, for now - we’re using an in-memory user hardcoded here.

Of course, since the application is actually managing users - we could also simply define a user details service and use these instead.

OK - all that’s left to do is **enable the new security configuration**.

We’re going to define an security initializer - which is going to basically register the main *DelegatingFilterProxy* that Spring Security needs in order to function:

```
public class SecurityWebApplicationInitializer extends AbstractSecurityWebApplicationInitializer {
    public SecurityWebApplicationInitializer() {
        super(LssSecurityConfig.class);
    }
}
```

Notice that the framework makes that available to use, so we don’t have to do much here. Just extend it and point it to our configuration - and that’s it - we are done.

Let’s fire up the application - and let’s see it in action.

2.4. Running the App

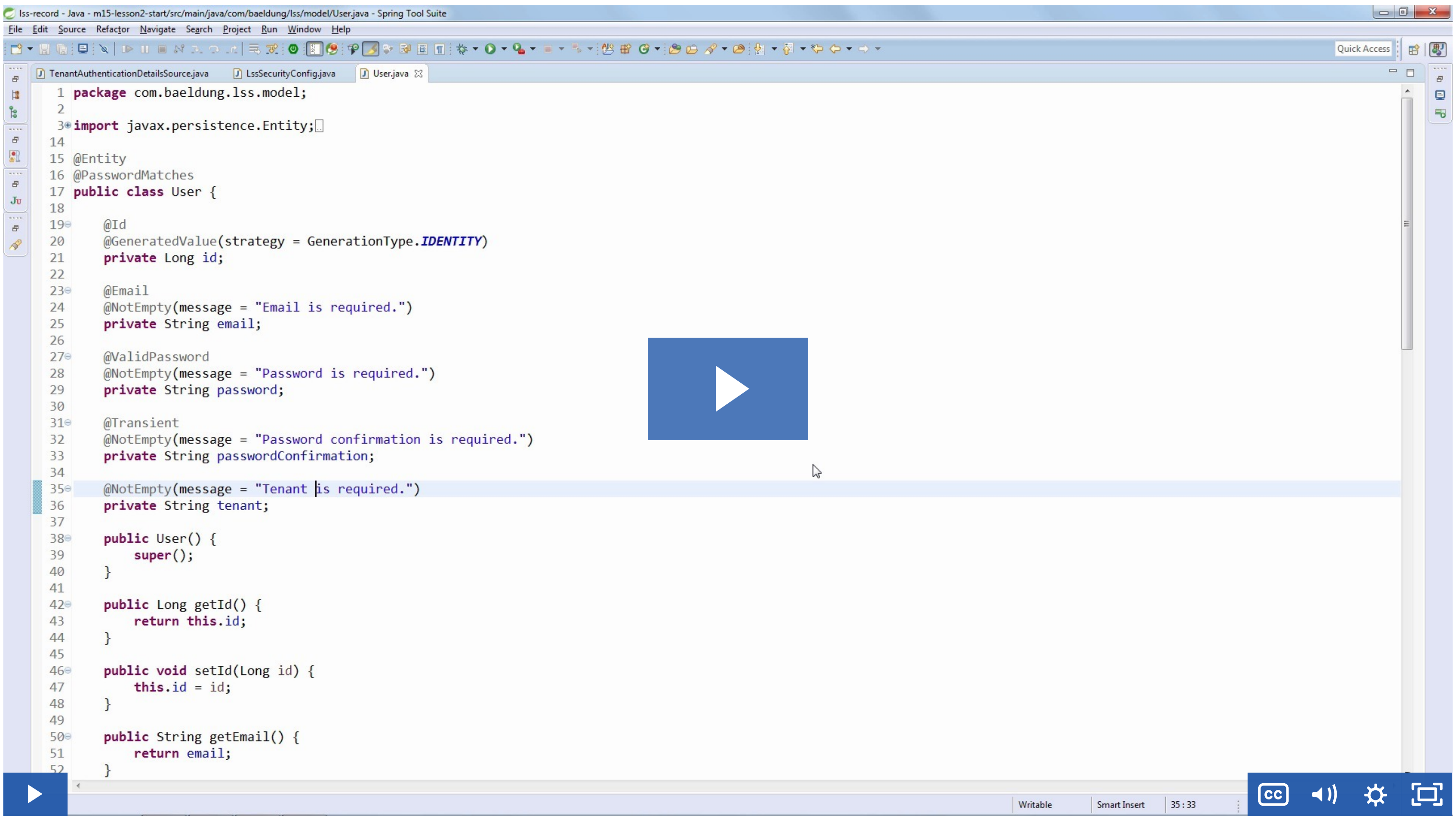
A final note here when you run the app. Because this is secured using Basic Authentication, keep in mind that the browser caches Basic Auth credentials - so you'll only be promoted for them once.

Also - when you're working with Basic Auth - it's important to always use a Incognito/Private window. That way the results aren't affected by any previous runs you might have done.

3. Resources

- Apache TomEE download

Lesson 2: Multi-Tenancy with Spring Security


lss-record - Java - m15-lesson2-start/src/main/java/com/baeldung/lss/model/User.java - Spring Tool Suite
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
TenantAuthenticationDetailsSource.java
LssSecurityConfig.java
User.java
Writable
Smart Insert
35 : 33

```
package com.baeldung.lss.model;

import javax.persistence.Entity;

@Entity
@PasswordMatches
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Email
    @NotEmpty(message = "Email is required.")
    private String email;

    @ValidPassword
    @NotEmpty(message = "Password is required.")
    private String password;

    @Transient
    @NotEmpty(message = "Password confirmation is required.")
    private String passwordConfirmation;

    @NotEmpty(message = "Tenant is required.")
    private String tenant;

    public User() {
        super();
    }

    public Long getId() {
        return this.id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getEmail() {
        return email;
    }
```

1. Goal

The simple goal of this lesson is to show you the options and considerations of a multi-tenant implementation with Spring Security.

2. Lesson Notes

The relevant module you need to import when you're starting with this lesson is: m15-lesson2-start

If you want to skip and see the complete implementation, feel free to jump ahead and import: m15-lesson2

2.1. Intro to Security Multi-Tenancy

First, let's discuss the scope of this lesson; multi-tenancy at the persistence level isn't particularly relevant here - what we're focusing on is **the security aspects of multi-tenancy**.

The first question we need to answer is - do we need the tenant information during authentication?

We have a few possible answers to this question:

- no, we don't need it (it's only relevant after we authenticate)
- we need it for checks, but not for the retrieval of the user from persistence
- we need it to get to the user

Now, discussing the first option - in this case the auth process works exactly as in a standard system.

In the other two cases the tenant info should be present in the initial authentication request, and we'll need to extract it and pass it along to the authentication process.

2.2. The Impl - Scenario 2

And so the first thing we're going to do is - we're going to use the *UsernamePasswordAuthenticationFilter* and extract the tenant info from the request via an *authenticationDetailsSource*.

Let's implement the custom authentication details source - and extract the info from the HTTP request and store it in the *authRequest* to be used later on:

```
@Component
public class TenantAuthenticationDetailsSource
  implements AuthenticationDetailsSource<HttpServletRequest, String> {
    @Override
    public String buildDetails(HttpServletRequest request) {
        return request.getParameter("tenant");
    }
}
```

Let's wire that into the security config:

```
@Autowired
private TenantAuthenticationDetailsSource authenticationDetailsSource;
...
.formLogin()
...
.authenticationDetailsSource(authenticationDetailsSource)
```

Now, when we get to the *AuthorizationManager* and to the auth providers - we'll have the tenant information available.

So, next, let's define the auth provider:

```
@Component
public class TenantAuthProvider extends DaoAuthenticationProvider {
    @Autowired
    private UserDetailsService userDetailsService;
    @PostConstruct
    private void after() {
        this.setUserDetailsService(userDetailsService);
    }
}
```

And wire that in:

```
auth.authenticationProvider(tenantAuthProvider);
```

Next, we'll change our user details service to now **return a custom principal**, that also has tenant information:

```
Principal principal = new Principal(
  user.getEmail(), user.getPassword(), true, true, true, true, getAuthorities(ROLE_USER));
principal.setTenant(user.getTenant());
return principal;
```

Let's of course also create the custom Principal as well:

```
public class Principal extends User {
    private String tenant;
    public Principal(
      String username, String password, boolean enabled, boolean accountNonExpired, boolean credentialsNonExpired, boolean accountNonLocked, Collection<? extends GrantedAuthority> authorities) {
        super(username, password, enabled, accountNonExpired, credentialsNonExpired, accountNonLocked, authorities);
    }
    public String getTenant() {
        return tenant;
    }
    public void setTenant(String tenant) {
        this.tenant = tenant;
    }
}
```

Finally, we'll add an extra check in the provider:

```
@Override
protected void additionalAuthenticationChecks(
  UserDetails userDetails, UsernamePasswordAuthenticationToken authentication) throws AuthenticationException {
    Principal principal = (Principal) userDetails;
    if (!principal.getTenant().equals(authentication.getDetails())) {
        throw new BadCredentialsException("Incorrect tenant information");
    }
    super.additionalAuthenticationChecks(userDetails, authentication);
}
```

Notice that the UI now has a tenant field as well.

2.3. The Impl - Scenario 3

Now - let's revisit the initial question - what do we need this tenant information for?

Our previous implementation used the tenant just for the check.

Not let's quickly see **how we could use it to also retrieve the user from the persistence layer.**

That might, for example, needed if we're using a multi-tenancy strategy that actually separates the data of each tenant into a separate DB or schema.

In that case, we first need to know which DB to go to when retrieving the user.

The only thing we'll need to change is the auth provider.

The *DaoAuthenticationProvider* isn't flexible enough to allow us to properly do what we need.

Now that we're not only doing a simple check, but we're actually changing the way we access the persistence layer and retrieve the user - we'll have to change our provider and actually implement the *retrieveUser* method.

That way we have access to the original auth request object (which has the tenant info). And with this info, we can simply retrieve the user from the right DB.

2.4. After Authentication

Once we're authenticated, we'll need to keep track of the tenant info in a custom *UserDetails* impl.

Now let's look at the standard, authenticated requests.

Here, again, we have two scenarios - depending on the way multi-tenancy is implemented.

- we either need to check, when data is accessed - that the tenant of the data is the same as the tenant of the user accessing that data
- or we need the tenant information to actually access the data

In the first scenario - the authorization check can be performed in most cases, with the help of a simple *@PostAuthorize* expression - that checks that the tenant info on the data (an id for example) is the same as the tenant info on the currently authenticated principal.

Here's a quick example doing that check:

```
@PostAuthorize("returnObject.tenant.id == principal.tenant.id")
```

In the second scenario, we'll have to pass in the tenant info (from the currently authenticated principal) to the persistence layer, give the fact that it's needed to access the data.

And once we know that the data is correctly assessed, we don't need any extra checks on that data, of course - since it's implicitly checked by the fact that it's segregated per tenant.

2.5. The Tenant during Registration

Up until this point we talked about sending the tenant information during the authentication process and then in standard requests, after authentication.

Let's also briefly touch on handling the Tenant info during Registration - even though this is not necessarily a security concern, but more of a persistence (and user experience) one.

At a high level, there are two clear options.

For enterprise clients - the Tenants is usually pre-created during the client onboarding. Large enterprise customers are more hands-on, and don't generally register a full tenant themselves.

For smaller, hands-off clients, the implementation usually allow them to fill in the name of their company and the system can create the tenant based on that. The first-time registration process basically creates both the Tenant and the first user.

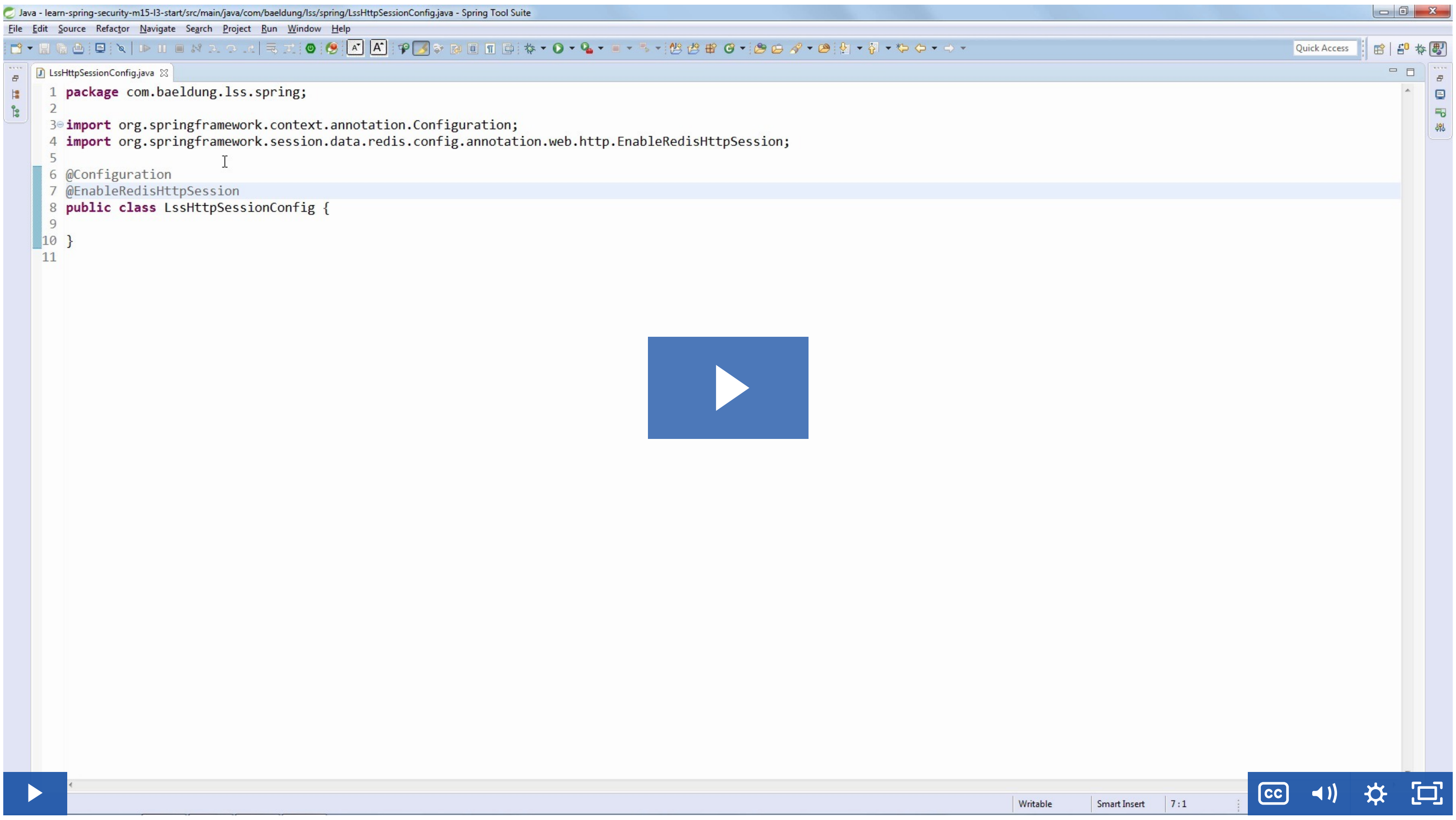
3. Resources

- [Declarative multi-tenant security with Spring Security and Spring-MVC]

- [Robert Winch on Spring Security and Multi-Tenant Applications on the Cloud]

- [Extending Spring Security OAuth for Multi-Tenant]

Lesson 3: Session Management with spring-session


Java - learn-spring-security-m15-l3-start/src/main/java/com/baeldung/lss/spring/LssHttpSessionConfig.java - Spring Tool Suite
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
LssHttpSessionConfig.java
Writable
Smart Insert
7 : 1

```java
package com.baeldung.lss.spring;

import org.springframework.context.annotation.Configuration;
import org.springframework.session.data.redis.config.annotation.web.http.EnableRedisHttpSession;

@Configuration
@EnableRedisHttpSession
public class LssHttpSessionConfig {

}
```

1. Goal

in this lesson we're going to use the relatively new spring-session project to store our session data in Redis instead of in the web server, where the HTTP session data is stored by default.

2. Lesson Notes

The relevant module you need to import when you're starting with this lesson is: m15-lesson3-start

If you want to skip and see the complete implementation, feel free to jump ahead and import: m15-lesson3

2.1. The Implementation

First, let's talk about the persistence aspect of storing the session data of our application.

We are going to use a Redis server - so you'll have to set that up locally before starting to work through this lesson.

Let's start by defining the properties to access it in our application.properties file:

```
spring.redis.host=localhost
spring.redis.port=6379
```

Next, we'll then add *spring-session* to our pom, as well as the redis starter from Spring Boot:

```
<dependency>
    <groupId>org.springframework.session</groupId>
    <artifactId>spring-session</artifactId>
    <version>1.2.0.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-redis</artifactId>
</dependency>
```

If you're not using Spring Boot, you can add the Redis dependency as follows:

```
<dependency>
    <groupId>org.springframework.session</groupId>
    <artifactId>spring-session-data-redis</artifactId>
    <version>1.2.0.RELEASE</version>
    <type>pom</type>
</dependency>
```

Next, we need to create a new Servlet Filter that basically replaces the default HttpSession implementation.

This will of course ties session data to the HTTP session - with an implementation backed by Spring Session and Redis.

We'll do this by creating a simple session config class - *LssHttpSessionConfig* class - annotated it *@EnableRedisHttpSession*.

Because we're using Boot, it also creates the Resis connection factory for us. But without Boot, we have to define that bean ourselves:

```
@Bean
public JedisConnectionFactory connectionFactory() {
    return new JedisConnectionFactory();
}
```

Next, we'll need to make sure that Tomcat will actually use this new filter when processing requests.

Again, this step is automatically done by Boot. Without Boot, we'll have to do this manually by defining an initializer manually and extending the base class provided by Spring Session - *AbstractHttpSessionApplicationInitializer*.

Finally, we'll make sure to add the new configuration into our standard Boot runner.

And that's it - we're ready to run the app.

2.2. A Live Run

Now, to verify everything’s working as expected, let’s have a look into Redis and let’s verify that data is actually reaching Redis.

First, let’s make sure there’s no data before we log in:

```
> KEYS *
```

Now, let’s log in (which will create a session) and let’s make sure that session data reaches and is stored in Redis:

```
> KEYS *
```

And there we go - session data is now properly stored in our running Redis instance here.

Finally, let’s have a look at the new cookie.

Because we’re using a different mechanism to store our sessions - that’s of course going to have a direct impact on the way cookies work by default as well.

The main difference is that the old *JSESSIONID* cookie is now gone - and replaced by a simple *SESSION* cookie.

The value of the cookie is the key name in Redis.

And of course, if the key is deleted in Redis - the session is invalidated.

Let’s wrap up by testing that:

```
> FLUSHALL
> KEYS *
```

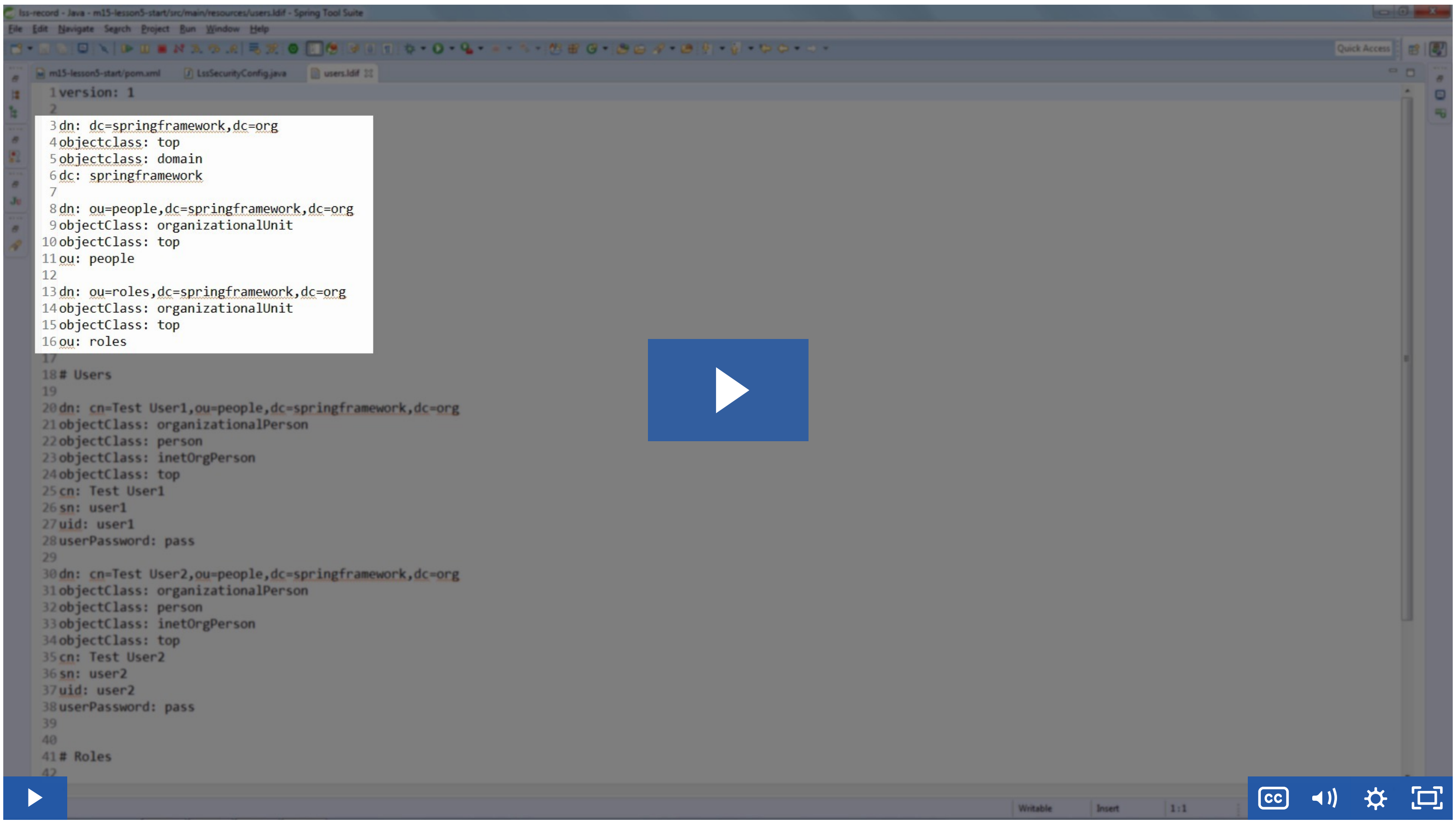
After clearing Redis, let refresh the application; we’re redirected to the login page - just as expected.

3. Resources

- Redis Comands

- Scaling out with Spring Session

Lesson 4: Spring Security with LDAP


lss-record - Java - m15-lesson5-start/src/main/resources/users.ldif - Spring Tool Suite
File Edit Navigate Search Project Run Window Help
Quick Access
m15-lesson5-start/pom.xml
LssSecurityConfig.java
users.ldif
Writable
Insert
1:1

```
version: 1

dn: dc=springframework,dc=org
objectclass: top
objectclass: domain
dc: springframework

dn: ou=people,dc=springframework,dc=org
objectClass: organizationalUnit
objectClass: top
ou: people

dn: ou=roles,dc=springframework,dc=org
objectClass: organizationalUnit
objectClass: top
ou: roles

# Users

dn: cn=Test User1,ou=people,dc=springframework,dc=org
objectClass: organizationalPerson
objectClass: person
objectClass: inetOrgPerson
objectClass: top
cn: Test User1
sn: user1
uid: user1
userPassword: pass

dn: cn=Test User2,ou=people,dc=springframework,dc=org
objectClass: organizationalPerson
objectClass: person
objectClass: inetOrgPerson
objectClass: top
cn: Test User2
sn: user2
uid: user2
userPassword: pass


# Roles
```

1. Goal

This lesson is entirely focused on LDAP and on getting to a fully working implementation with Spring Security.

2. Lesson Notes

The relevant module you need to import when you're starting with this lesson is: m15-lesson4-start

If you want to skip and see the complete implementation, feel free to jump ahead and import: m15-lesson4

2.1. Why LDAP?

LDAP is a very mature, application layer protocol with wide industry support.

The main benefit of using LDAP is this very support - most mature languages and frameworks will be able to interact with LDAP for security. This basically means that a lot of different and entirely unrelated systems in an organization will be able to share the same common way to authenticate, which is very useful.

Also, LDAP fits the role of an authentication provider well because it's a hierarchical system - which makes it well suited for an organizational structure.

But going into in-depth details about LDAP itself is not the purpose of this lesson - so you should definitely have at least a basic understanding of it.

2.2. The Impl

We're going to start with Maven and add our LDAP dependencies:

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-ldap</artifactId>
</dependency>
<dependency>
    <groupId>org.apache.directory.server</groupId>
    <artifactId>apacheds-server-jndi</artifactId>
    <version>1.5.5</version>
</dependency>
```

We'll need to configure the global authentication of the system - and to do that we need to extend the *GlobalAuthenticationConfigurerAdapter*.

However, because our main security configuration is already extending a base class, we'll create a separate config class for the global security config:

```
@Configuration
protected static class AuthenticationConfiguration
  extends GlobalAuthenticationConfigurerAdapter { // @formatter:off
    @Override
    public void init(AuthenticationManagerBuilder auth) throws Exception {
        auth
        ...
    }
}// @formatter:on
```

It's here that we're going to start configuring LDAP. First, we're going to start defining it:

```
auth
  .ldapAuthentication()
  .contextSource()
  .ldif("classpath:users.ldif")
;
```

So, what are we doing here exactly?

We're simply defining an the configuration of an embedded LDAP server that Spring Security will set up for us.

All we really need is to define the data within this server - and that's what we're doing with that LDIF file. That's a file that basically sets up our entire LDAP structure.

Let’s have a look at that:

- so, the data starts off by defining a domain component as the of our organization
- and then an organizational unit called *people*
- and another called *roles*
- then we’re defining 2 users - *user1* and *user2*
- and 2 roles - *USER* and *ADMIN*
- notice that the roles are defining its members

And that’s it - a simple structure for our LDAP data.

Note that the location of this LDIF file is in */src/main/resources* - so it will be present on the classpath.

Next, let’s provide the pattern to search for users:

```
.userSearchBase("ou=people")
.userSearchFilter("(uid={0})")
```

The search base is the root node in the hierarchy of LDAP data - from which the search will start.

And the filter username - which is necessary for the username to be bound during the authentication process - in our case, the *uid*.

A quick side-note here is that - for simple LDAP structures, we could replace these two configurations - the search base and the search filter - with a single one, via the following API:

```
.userDnPatterns()
```

We’re going to do the same thing for groups - which in our case are the roles we defined:

```
.groupSearchBase("ou=roles")
.groupSearchFilter("member={0}")
```

Remember that - in the LDIF file - we define the roles within the *roles* organizational unit.

The reason we need to customize this search process as well is - because we’re using a more custom and complex LDAP structure, which is very common in production environments.

We can’t and shouldn’t assume that the LDAP data is going to be defined exactly how we’re expecting it - it’s us that need to adapt to the structure of that data.

Finally, notice that, here in the configuration - we’re allowing users to hit /user. But we’re actually requiring ADMIN for everything else.

So, when we log in - we’re going to be able to see the main page, but for example we won’t be able to do a *create*.

And that’s about it - we should now be fully up and running with our LDAP system.

3. Resources

- LDAP Authentication in the Spring Security Reference