

Lesson 1: A Simple Registration Flow

```
Java - spring-security-m2-11/src/main/resources/templates/registrationPage.html - Spring Tool Suite
File Edit Source Navigate Search Project Run Window Help
Quick Access
registrationPage.html
9
10 </head>
11 <body>
12 <div class="container">
13
14 <h1>Registration page</h1>
15
16 <form id="userForm" th:action="@{/user/register}" th:object="${user}" action="#" method="post" class="form-horizontal">
17
18 <div class="form-group" th:classappend="${#fields.hasErrors('email')} ? 'has-error'">
19 <label class="control-label col-xs-2" for="email">Email</label>
20 <div class="col-xs-10">
21 <input id="email" type="email" title="email" th:field="${email}" /> <span th:if="${#fields.hasErrors('email')}" th:errors="*{em
22 </div>
23 </div>
24
25 <div class="form-group" th:classappend="${#fields.hasErrors('password')} ? 'has-error'">
26 <label class="control-label col-xs-2" for="password">Password</label>
27 <div class="col-xs-10">
28 <input id="password" type="password" title="password" th:field="${password}" /> <span th:if="${#fields.hasErrors('password')}"
29 Error</span>
30 </div>
31 </div>
32 <div class="form-group" th:classappend="${#fields.hasErrors('passwordConfirmation')} ? 'has-error'">
33 <label class="control-label col-xs-2" for="passwordConfirmation">Password Confirmation</label>
34 <div class="col-xs-10">
35 <input id="passwordConfirmation" type="password" title="passwordConfirmation" th:field="${passwordConfirmation}" /> <span th:if
36 th:errors="*{passwordConfirmation}" class="help-block">Password Confirmation Error</span>
37 </div>
38 </div>
39
40 <div class="form-actions col-xs-offset-2 col-xs-10">
41 <input type="submit" class="btn btn-primary" value="Register" />
42 </div>
43
44 </form>
45
46 </div>
```

1. Main Goal

This lesson will show you how to set up a very simple registration process for your MVC app.

2. Lesson Notes

If you're using the git repository to get the project - you should be using [the module2 branch](#).

The relevant module you need to import when you're starting with this lesson is: [m2-lesson1](#)

If you want to skip and see the complete implementation, feel free to jump ahead and import: [m2-lesson2](#)

The credentials used in the code of this lesson are: user/pass (in memory).

2.1. The Service Layer

In the previous module we were using the repository layer directly; in this module **we're introducing a simple service layer** between our raw repositories and the controller layer.

At this point we're only going to have a **single service - the user service**: *IUserService* (and *UserService*)

Also note that even though we now have a service layer, we can (and are) still using the repository in controllers when it's convenient. We will have a stricter separation of layers as we move to the more advanced modules.

2.2. Simple User Registration

First, we are going to create a basic registration page on the client side of the app.

We are then going to define a **controller to handle displaying that new form**:

```
@Controller
class RegistrationController {
    @Autowired
    private IUserService userService;
    @RequestMapping(value = "/signup")
    public ModelAndView registrationForm() {
        return new ModelAndView("registrationPage", "user", new User());
    }
}
```

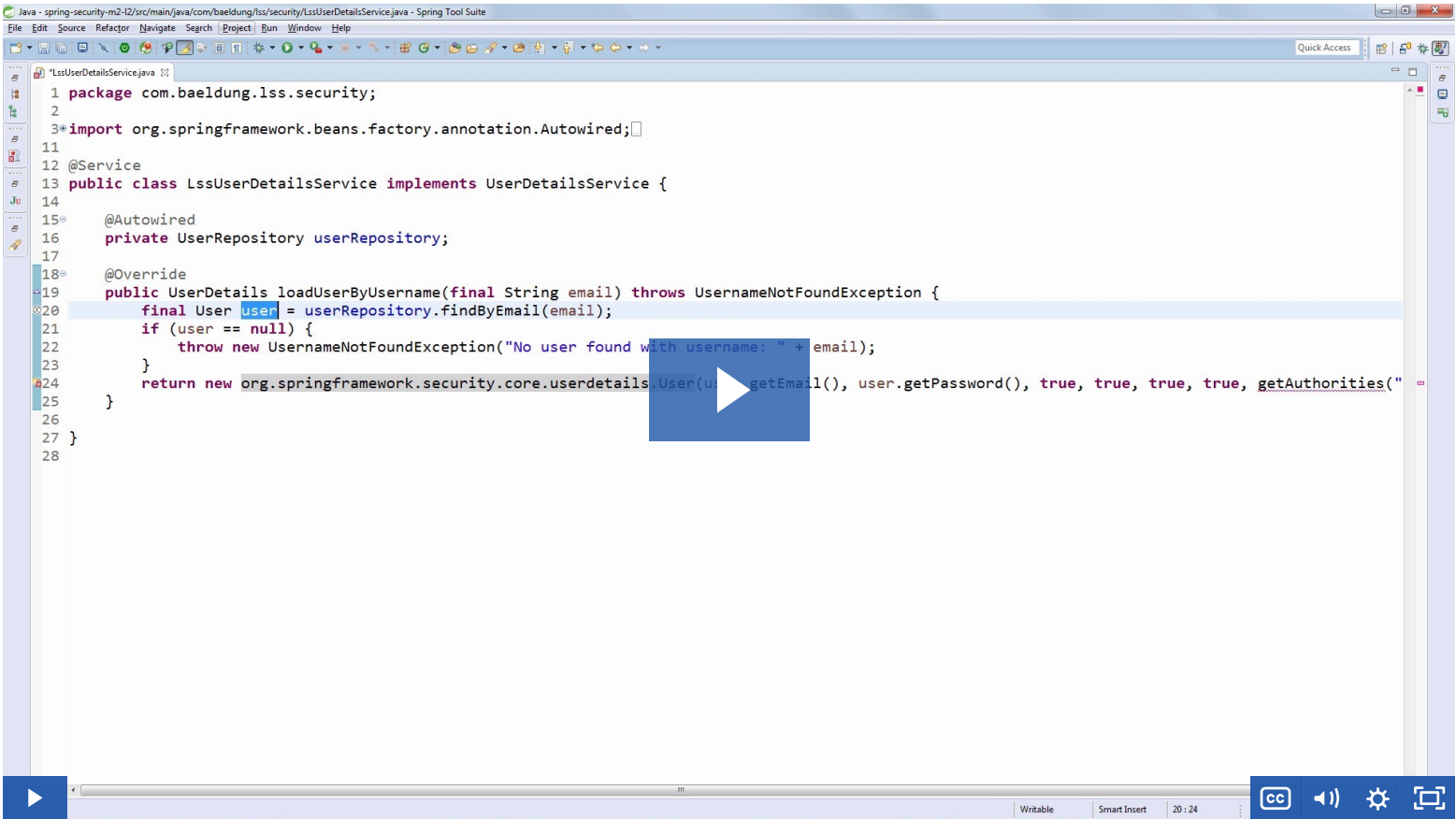
Finally, we'll implement the actual action - the registration logic:

```
@RequestMapping(value = "/user/register")
public ModelAndView registerUser(@Valid User user, BindingResult result) {
    if (result.hasErrors()) {
        return new ModelAndView("registrationPage", "user", user);
    }
    try {
        userService.registerNewUser(user);
    } catch (EmailExistsException e) {
        result.addError(new FieldError("user", "email", e.getMessage()));
        return new ModelAndView("registrationPage", "user", user);
    }
    return new ModelAndView("redirect:/login");
}
```

3. Resources

- [The Spring Security Registration Series](#) on Baeldung

Lesson 2: Authentication using Real Users



1. Main Goal

The main focus here will be making sure our authentication process actually uses the real, persisted, newly registered users.

2. Lesson Notes

The relevant module you need to import when you're starting with this lesson is: [m2-lesson2](#)

If you want to skip and see the complete implementation, feel free to jump ahead and import: [m2-lesson3](#)

The credentials used in the code of this lesson are: user/pass (in memory).

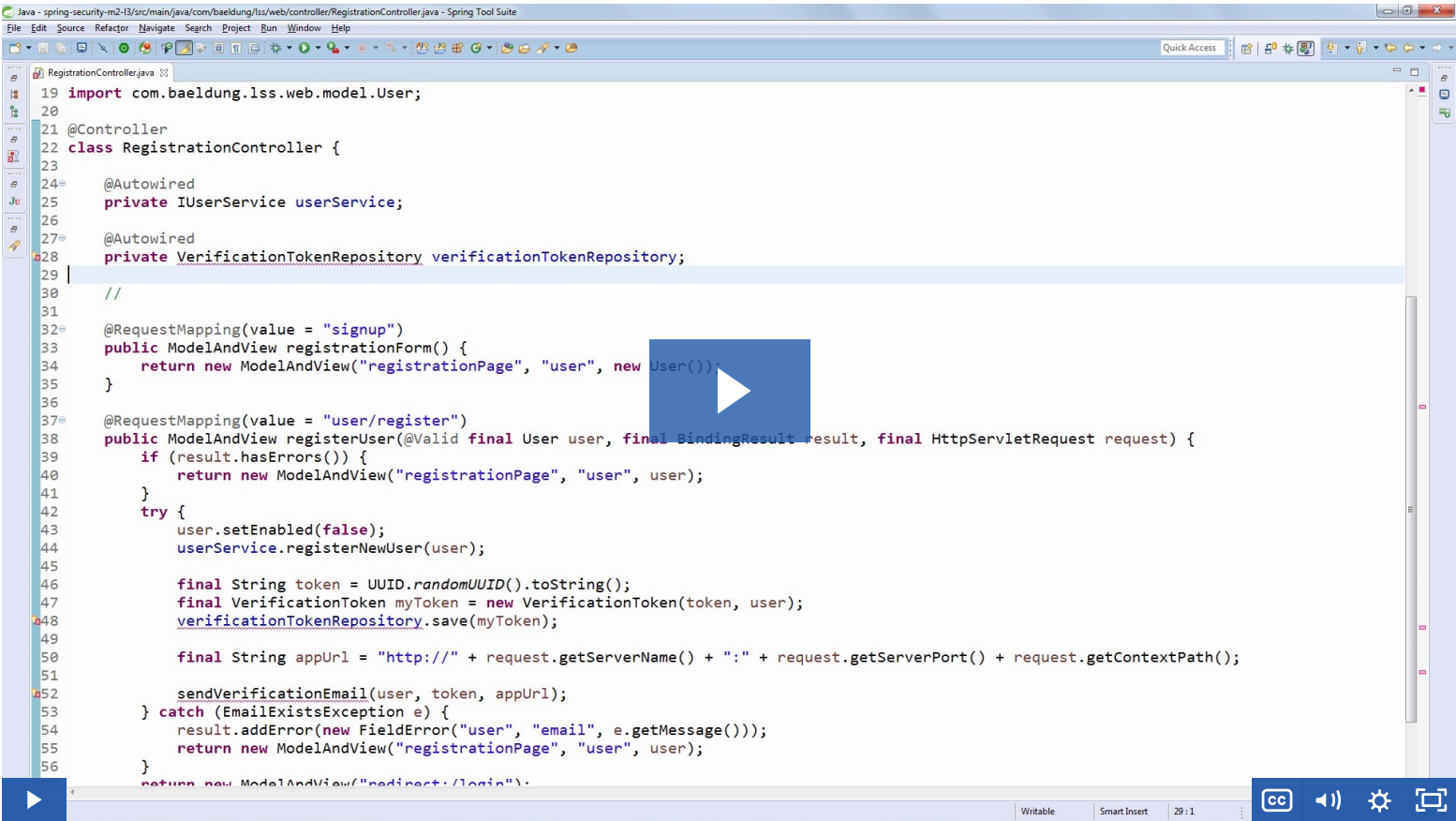
We are going to start by defining a users details service **to drive our authentication process**:

```
@Service
public class LssUserDetailsService implements UserDetailsService {
    @Autowired
    private UserRepository userRepository;
    public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException {
        User user = userRepository.findByEmail(email);
        if (user == null) {
            throw new UsernameNotFoundException("No user found with username: " + email);
        }
        return new org.springframework.security.core.userdetails.User(
            user.getEmail(),
            user.getPassword(),
            true, true, true, true,
            getAuthorities("ROLE_USER"));
    }
    private Collection<? extends GrantedAuthority> getAuthorities(String role) {
        return Arrays.asList(new SimpleGrantedAuthority(role));
    }
}
```

We'll then simply wire this new service into our security configuration and use it:

```
@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
    auth.userDetailsService(userDetailsService);
}
```

Lesson 3: Activate a New Account via Email



1. Main Goal

The focus here is no longer registering users directly but requiring them to verify their email first.

2. Lesson Notes

The relevant module you need to import when you're starting with this lesson is: [m2-lesson3](#)

If you want to skip and see the complete implementation, feel free to jump ahead and import: [m2-lesson4](#)

The credentials used in the code of this lesson are: test@email.com/pass (data.sql).

An important note here is that, to keep things simple in the video, we're handling the email logic in the *RegistrationController*.

That all works perfectly fine, but in the code of the lesson we're going one step further. We're decoupling the email logic via an event and moving the email details into a listener - *RegistrationListener*.

2.1. The Verification Token

We're going to drive the functionality via a verification token:

```

@Entity
public class VerificationToken {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String token;
    @OneToOne(targetEntity = User.class, fetch = FetchType.EAGER)
    @JoinColumn(nullable = false, name = "user_id")
    private User user;
    private Date expiryDate;
    // standard getters and setters
}

```

We're then going to do define a simple Spring Data JPA repository to have a simple persistence API for this new entity:

```

public interface VerificationTokenRepository extends JpaRepository<VerificationToken, Long> {
    VerificationToken findByToken(String token);
}

```

And finally, we're going to add an *enabled* flag into our *User* entity:

```

@Column
private Boolean enabled;

```

2.2. The Registration Logic

During registration, we're now going to send out an email with a unique link where they user can verify their email:

```

String appUrl = "http://" + request.getServerName() + ":" +
    request.getServerPort() + request.getContextPath();
String token = UUID.randomUUID().toString();
VerificationToken myToken = new VerificationToken(token, user);
verificationTokenRepository.save(myToken);
sendVerificationEmail(user, token);

```

2.3. Verifying Registration

Finally, when the user gets the email and clicks the link:

```
@RequestMapping(value = "/registrationConfirm")
public
ModelAndView confirmRegistration(
    Model model,
    @RequestParam("token") String token,
    RedirectAttributes redirectAttributes) {
    VerificationToken verificationToken = userService.getVerificationToken(token);
    User user = verificationToken.getUser();
    user.setEnabled(true);
    userService.saveRegisteredUser(user);
    redirectAttributes.addFlashAttribute("message", "Your account verified successfully");
    return new ModelAndView("redirect:/login");
}
```

Note that we're skipping over the error handling logic here to simplify the logic - but the full implementation does contain the error handling code as well.

2.4. STMP Config

When you're running the application locally, you'll of course need to configure the SMTP server in order to be able to send email.

Any valid SMTP server will be perfectly fine - the details are set up in *application.properties*.

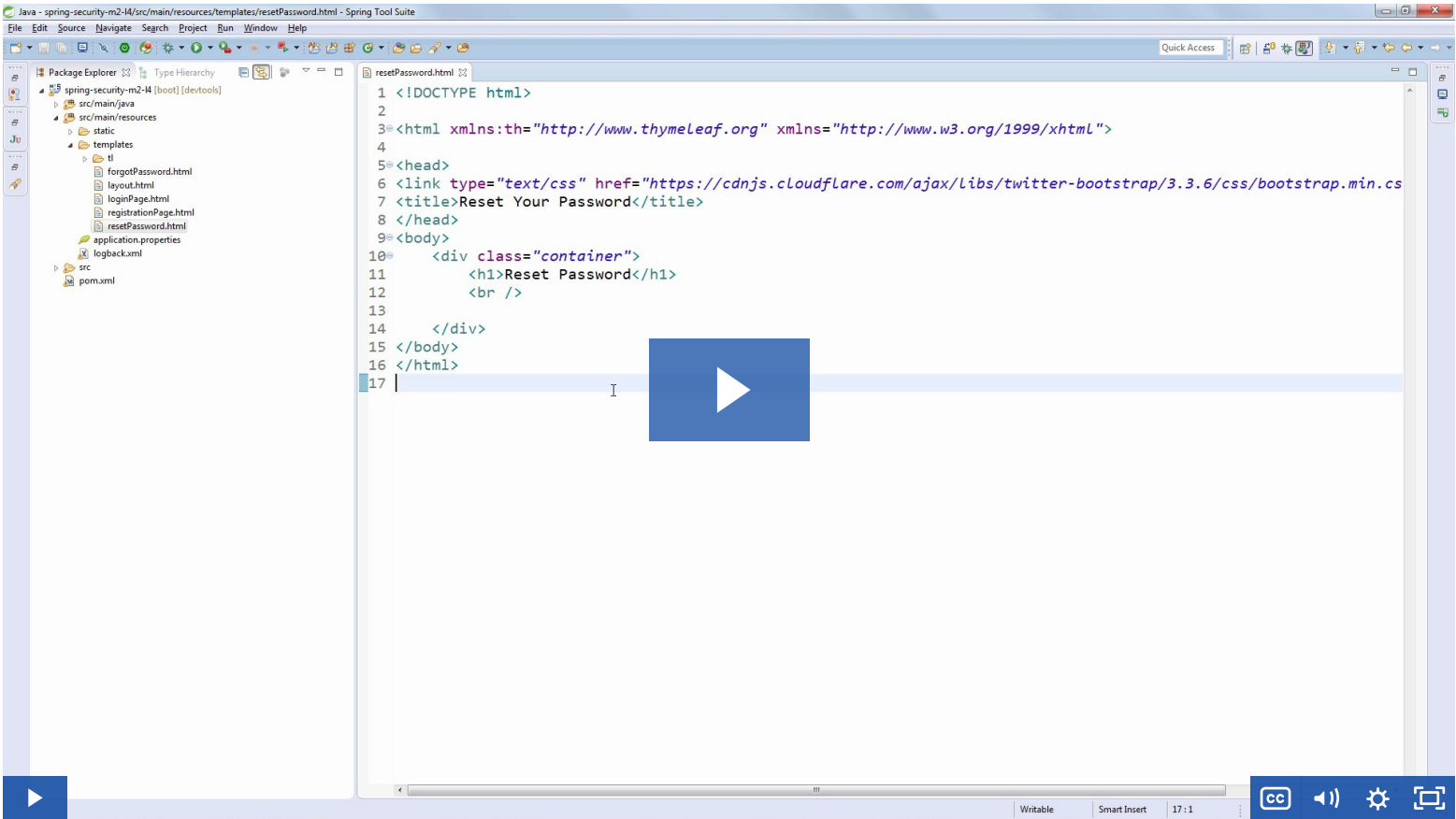
As a suggestion, you can use GMail as an SMTP server, or services like Amazon SES or Postman.

2.5. Errata

There is a known issue in the video - the VerificationToken is shown to be created in the wrong package; the correct package for it is: *com.baeldung.lss.model*

[The codebase](#) is of course correct - the problem is just in the video. And before the video itself is re-recorded with the correct package, it's important to be aware of the issue and create the token entity in the right package as you're going through the lesson.

Lesson 4: Deal with “I forgot my password”



1. Main Goal

The focus here is to allow the user to recover their password in a safe way, in case they forget it.

2. Lesson Notes

The relevant module you need to import when you're starting with this lesson is: [m2-lesson4](#)

If you want to skip and see the complete implementation, feel free to jump ahead and import: [m2-lesson5](#)

The credentials used in the code of this lesson are: test@email.com/pass (data.sql).

2.1. Moving the Registration Verification Email Logic

Remember that in the previous lesson, we implemented the simple logic of sending the verification email (with the token), when the user first registers.

We did that right in the *RegistrationController* - to keep things simple.

In the code of this lesson, you'll notice that I moved that logic into a *RegistrationListener* - simply to keep the controller clean and this logic separate.

2.2. The New Forgot Password Page

First, we are going to create the "I forgot my password" page to ask the user for the email address and start the recovery process.

Note that the video skips over the basic setup of that page (since we've seen that before):

- in *LssWebMvcConfiguration*:

```
registry.addViewController("/forgotPassword").setViewName("forgotPassword");
```

- in *LssSecurityConfig* - add the page to the *allowed* list:

```
http
    .authorizeRequests()
    .antMatchers(
        "/forgotPassword*",
        ...
    ).permitAll()
```

2.3. Triggering the Reset Process

We are then going to set up another token - very similar to the verification token we used in the previous lesson - called *ResetPasswordToken*.

Then, we're going to implement the server-side logic that handles the actual recovery process for the user:

```
@RequestMapping(value = "/user/resetPassword", method = RequestMethod.POST)
@ResponseBody
public ModelAndView resetPassword(
    HttpServletRequest request,
    @RequestParam("email") String userEmail,
    RedirectAttributes redirectAttributes) {
    User user = userService.findUserByEmail(userEmail);
    if (user != null) {
        String token = UUID.randomUUID().toString();
        PasswordResetToken myToken = new PasswordResetToken(token, user);
        passwordTokenRepository.save(myToken);
        sendResetEmail(token, user);
    }
}
```

```
        redirectAttributes.addFlashAttribute("message",
            "You should receive an Password Reset Email shortly");
        return new ModelAndView("redirect:/login");
    }
}
```

2.4. Verifying the Reset Process

Once the user gets the "reset your password" email and clicks on the unique link - they'll land on a simple page that allows them to change their password.

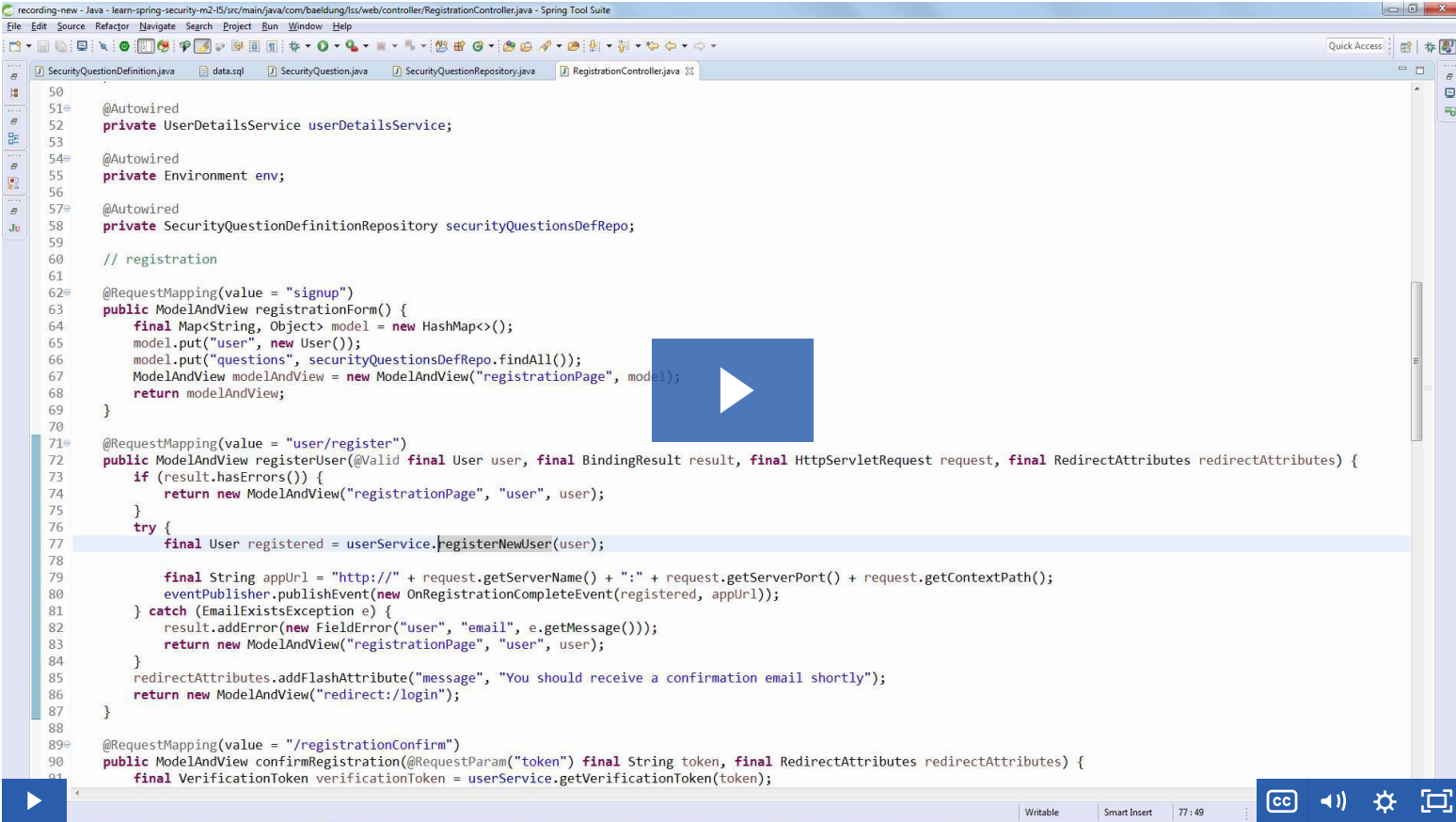
We'll first set up that page and set up the logic for displaying it:

```
@RequestMapping(value = "/user/resetPassword", method = RequestMethod.GET)
public ModelAndView showChangePasswordPage(
    @RequestParam("id") long id,
    @RequestParam("token") String token,
    RedirectAttributes redirectAttributes) {
    User user = passwordTokenRepository.findByToken(token).getUser();
    Authentication auth = new UsernamePasswordAuthenticationToken(
        user, null, userDetailsService.loadUserByUsername(user.getEmail()).getAuthorities());
    SecurityContextHolder.getContext().setAuthentication(auth);
    return new ModelAndView("resetPassword");
}
```

Then, we'll implement the server side logic that does the actual password change:

```
@RequestMapping(value = "/user/savePassword", method = RequestMethod.POST)
@ResponseBody
public ModelAndView savePassword(
    @RequestParam("password") String password,
    @RequestParam("passwordConfirmation") String passwordConfirmation,
    RedirectAttributes redirectAttributes) {
    User user = (User) SecurityContextHolder.getContext().getAuthentication().getPrincipal();
    userService.changeUserPassword(user, password);
    redirectAttributes.addFlashAttribute("message", "Password reset successfully");
    return new ModelAndView("redirect:/login");
}
```


Lesson 5: Doing Security Questions Right (NEW)



1. Goal

The simple goal of this lesson is to introduce security questions and show you how you can use these to add extra security for sensitive operations in your application.

2. Lesson Notes

The relevant module you need to import when you're starting with this lesson is: [m2-lesson5](#)

If you want to skip and see the complete implementation, feel free to jump ahead and import: [m2-lesson6](#)

Very simply put, security questions improve the security of our systems.

The basic idea is - whenever the user does something sensitive - for example updating their account password - we want to add some extra security around that operation.

One way to do that is by asking the user a security question that they've already provided their answer to.

2.1. A Security Question Definition

First we're going to define a few of these questions in our system.

We could of course hard-code them, but we could also persist them - in case we want to modify them at runtime.

So let's **define the model** for this question definition:

```
import javax.persistence.Id;
import javax.persistence.Entity;
@Entity
public class SecurityQuestionDefinition {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column
    @NotEmpty
    private String text;
}
```

And the basic Spring Data repository:

```
public interface SecurityQuestionDefinitionRepository extends JpaRepository<SecurityQuestionDefinition, Long> {
}
```

Next, we're going to use the Spring Boot - *data.sql* - to define these:

```
insert into security_question_definition (id, text) values (1, 'What is the last name of the teacher who gave you your first failing grade?');
insert into security_question_definition (id, text) values (2, 'What is the first name of the person you first kissed?');
insert
into security_question_definition (id, text) values (3, 'What is the
name of the place your wedding reception was held?');
insert into
security_question_definition (id, text) values (4, 'When you were young,
what did you want to be when you grew up?');
insert into security_question_definition (id, text) values (5, 'Where were you New Year's 2000?');
insert into security_question_definition (id, text) values (6, 'Who was your childhood hero?');
```

2.2. The Security Question of the User

Next, let's define the user-specific security question model:

```
import javax.persistence.Id;
import javax.persistence.Entity;
@Entity
public class SecurityQuestion {
    @Id
```

```

@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
@OneToOne(targetEntity = User.class, fetch = FetchType.EAGER)
@JoinColumn(nullable = false, name = "user_id", unique = true)
private User user;
@OneToOne(targetEntity = SecurityQuestionDefinition.class, fetch = FetchType.EAGER)
@JoinColumn(nullable = false, name = "securityQuestionDefinition_id")
private SecurityQuestionDefinition questionDefinition;
private String answer;
}

```

This is a simple structure that will hold the data when a user selects a security question for themselves.

Now, the repository:

```

public interface SecurityQuestionRepository extends JpaRepository<SecurityQuestion, Long> {
    //
}

```

And we're going to also define a specific retrieval operation there:

```

SecurityQuestion findByQuestionDefinitionIdAndUserIdAndAnswer(Long questionDefinitionId, Long userId, String answer);

```

2.3. The Front-End

Let's use these in the front-end (*registrationPage.html*):

```

<div class="form-group">
  <label class="control-label col-xs-2" for="question">Security Question:</label>
  <div class="col-xs-10">
    <select id="question" name="questionId">
      <option th:each="question : ${questions}"
        th:value="${question.id}"
        th:text="${question.text}">Question</option>
    </select>
  </div>
</div>
<div class="form-group">
  <label class="control-label col-xs-2" for="answer">Answer</label>
  <div class="col-xs-10">
    <input id="answer" type="text" name="answer"/>
  </div>
</div>

```

2.4. The New Registration Logic

Now that we're mostly data with the preparatory work, and can finally move to writing the registration logic.

First, let's display the security questions available for the user during registration; we're going to add these to the model in *registrationForm* (manually):

```

@RequestMapping(value = "signup")
public ModelAndView registrationForm() {
    Map<String, Object> model = new HashMap<>();
    model.put("user", new User());
    model.put("questions", securityQuestionDefinitionRepository.findAll());
    return new ModelAndView("registrationPage", model);
}

```

And let's handle the new data - the security question - that the front-end will be sending during the registration.

We'll go back to the RegistrationController - the registerUser method - and, after registering the user, we'll persist the security question as well:

```

SecurityQuestionDefinition questionDefinition = securityQuestionsDefRepo.findOne(questionId);
securityQuestionRepo.save(new SecurityQuestion(user, questionDefinition, answer));

```

2.5. Securing the "Reset Password" Operation

Finally, now that we have the security question for our user, let's start using it to secure a sensitive operation - resetting the password.

So, on the *resetPassword* page, we're going to now ask the user to answer their security question:

```

<div class="form-group">
  <label class="control-label col-xs-2" for="question">Security Question:</label>
  <div class="col-xs-10">
    <select id="question" name="questionId">
      <option th:each="question : ${questions}"
        th:value="${question.id}"
        th:text="${question.text}">Question</option>
    </select>
  </div>
</div>
<div class="form-group">
  <label class="control-label col-xs-2" for="answer">Answer</label>
  <div class="col-xs-10">
    <input id="answer" type="text" name="answer"/>
  </div>
</div>

```

And, on the server side, we're going to make use of that information (*savePassword*):

```

if (securityQuestionRepo.findByUserIdAndAnswer(user.getId(), answer) == null) {
    final Map<String, Object> model = new HashMap<>();
    model.put("errorMessage", "Answer to security question is incorrect");
    model.put("questions", securityQuestionsDefRepo.findAll());
    return new ModelAndView("resetPassword", model);
}

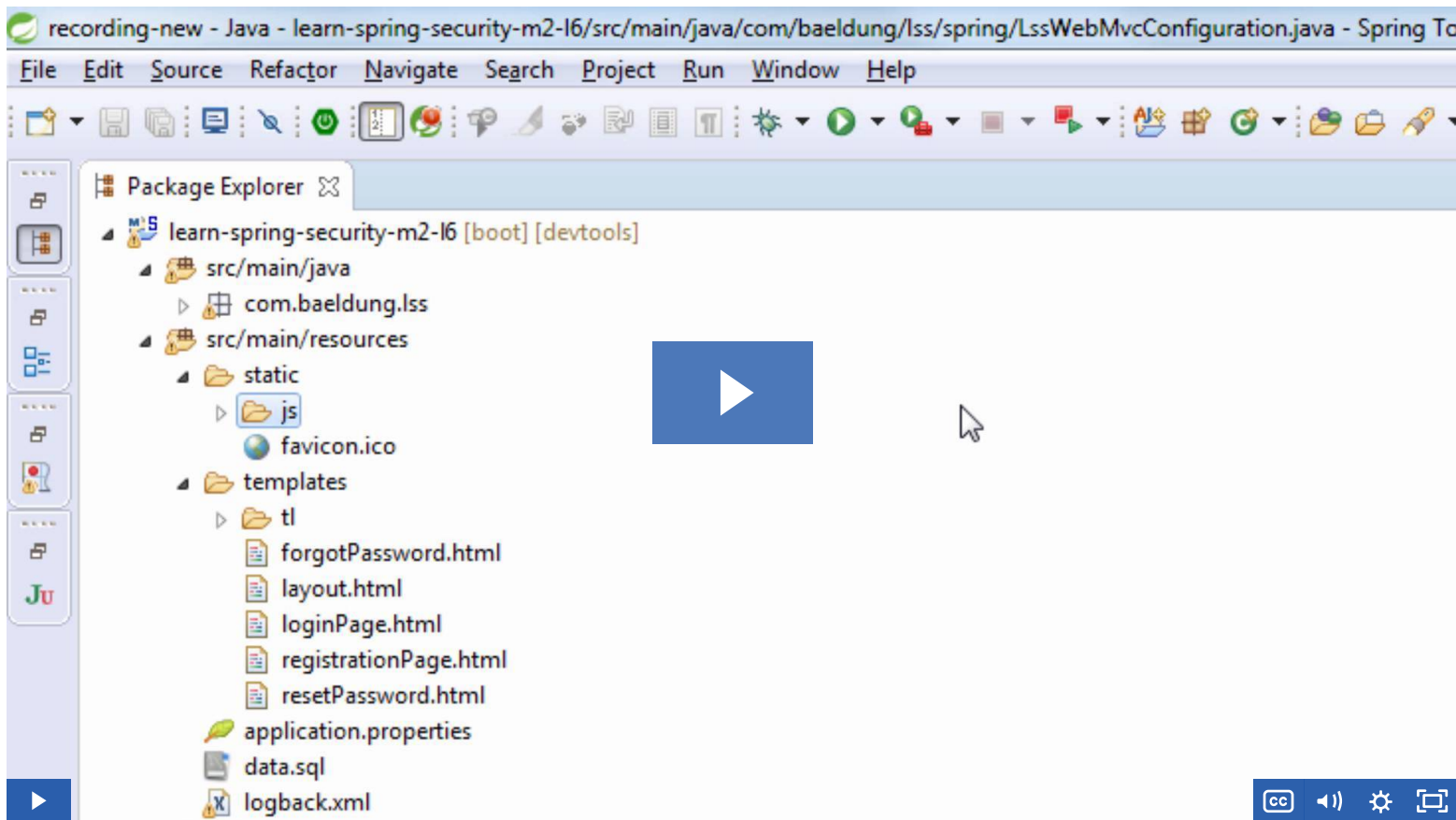
```

Right after the user is retrieved, we're going to run this query and see if we find a match based on the user id and the answer.

If there's no match, we simply return an error and of course don't change the password of the user.

And we're done - we can finally start up the server and use the new functionality from start to finish.

Lesson 6: Ensure Password Strength during Registration - part 1 (NEW)



1. Goal

In this lesson we'll focus on helping the user chose a good password.

2. Lesson Notes

The relevant module you need to import when you're starting with this lesson is: [m2-lesson6](#)

If you want to skip and see the complete implementation, feel free to jump ahead and import: [m2-lesson7](#)

2.1. Why "Strong" Passwords?

Significant data breaches are more common than we might think.

And it's the strength of the passwords in the system that determines the outcome of such as event.

Simply put, helping our users to select strong passwords can have a huge positive impact on the overall security of our system.

Our goal here is to set some constraints about how strong the password of a user needs to be, and give the user immediate feedback when they set their password.

2.2. Password Strength on the The Front End

We're going to start on the client side and make sure we help the user set a strong password there first.

We're going to be using a simple JQuery plugin called *jQuery Password Strength Meter for Twitter Bootstrap* ([link](#)):

```
<script src="/js/jquery-1.7.2.js"></script>
<script src="/js/pwstrength.js"></script>
```

We also need to configure the resolution mechanism for these static resources - to make sure we can actually refer to them directly like that:

```
@Override
public void addResourceHandlers(ResourceHandlerRegistry registry) {
    registry.addResourceHandler("/**").addResourceLocations(new String[] { "classpath:/static/" });
}
```

Let's also use the new mechanism on the page:

```
<script type="text/javascript">
    $(document).ready(function () {
        options = {
            common: {minChar:8},
            ui: {
                showVerdictsInsideProgressBar:true,
                showErrors:true,
                errorMessages:{
                    wordLength: 'Your password is too short',
                }
            }
        };
        $('#password').pwstrength(options);
    });
</script>
```

And we're done.

When we now register, we're going to see the new password criteria being verified in real-time with proper feedback provided to the user.

Remember that, while the front-end is definitely helpful, we cannot rely on just the front end. We also need the back-end to verify the same rules for the password and of course refuse passwords that don't adhere to these rules.

2.3. Password Strength in the Back End

Similarly to the front-end, we're going to use a library here as well to define our password constraints. Doing that manually would simply be error-prone and a lot more work.

Let's start by defining Passay ([link](#)) - which basically provides a clean, simple way to define password rules without having to roll these out ourselves:

```
<dependency>
  <groupId>org.passay</groupId>
  <artifactId>passay</artifactId>
  <version>1.2.0</version>
</dependency>
```

Now, we're going to implement some simple validation logic to check these password rules.

We could do that manually of course, but we can also do it via a custom validation annotation:

```
package com.baeldung.lss.validation;
import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
@Documented
@Constraint(validatedBy = PasswordConstraintValidator.class)
@Target({ TYPE, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
public @interface ValidPassword {
    String message() default "Invalid Password";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

And of course the missing piece of the puzzle here is the actual password validation logic:

```
public class PasswordConstraintValidator implements ConstraintValidator<ValidPassword, String> {
}
```

And the actual implementation to verify/validate the strength of a password:

```
PasswordValidator validator = new PasswordValidator(Arrays.asList(new LengthRule(8, 30)));
RuleResult result = validator.validate(new PasswordData(password));
if (result.isValid()) {
    return true;
}
context.buildConstraintViolationWithTemplate(
    Joiner.on("\n").join(validator.getMessages(result)))
    .addConstraintViolation();
return false;
```

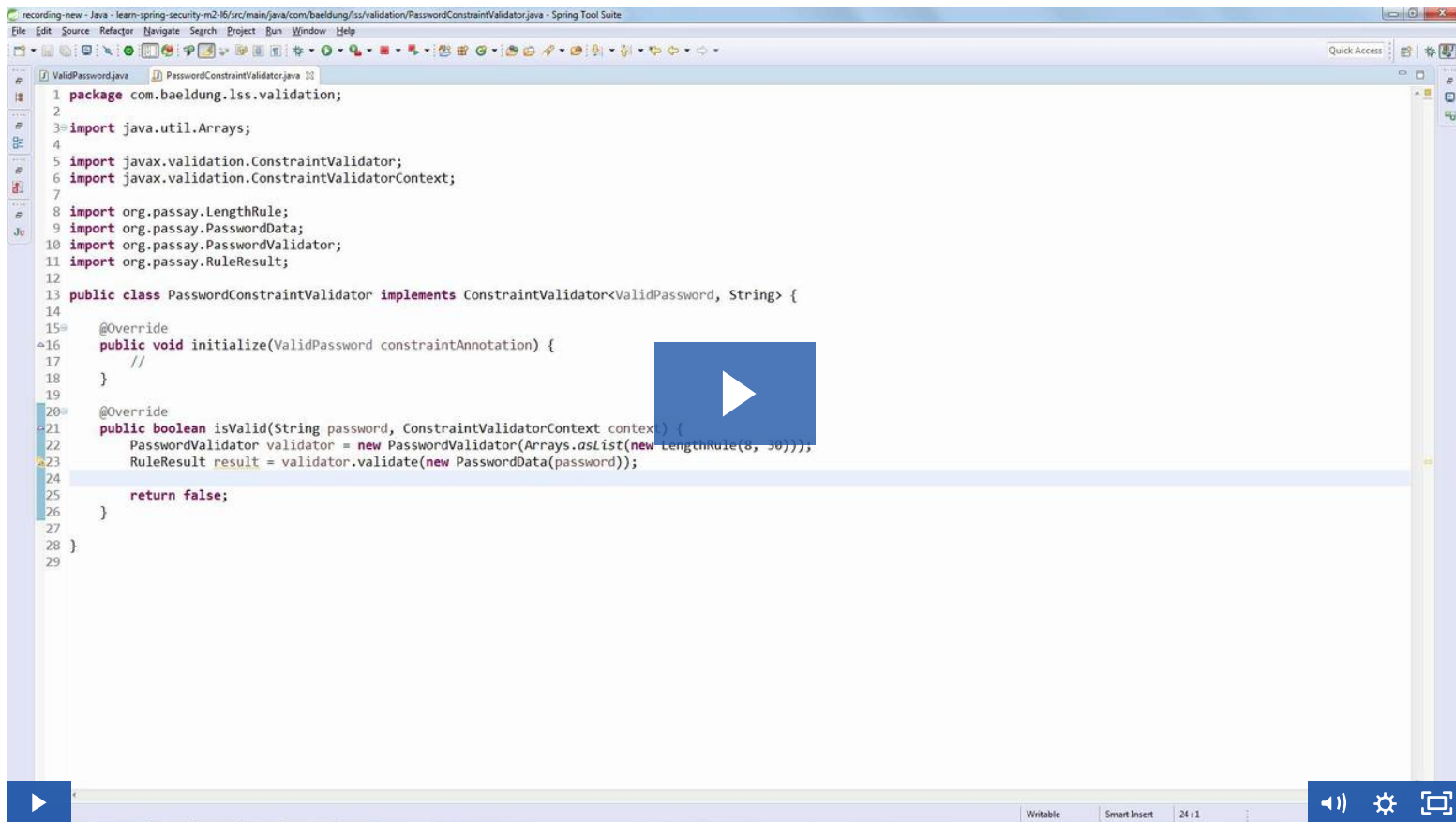
So a relatively simple implementation here as you can see.

And now, with everything set up, we can finally use the new annotation in our User

```
@ValidPassword
private String password;
```

And we're done. Now the back-end will also validate the strength of the passwords send from the front-end.

Lesson 6: Ensure Password Strength during Registration - part 2 (NEW)



```
1 package com.baeldung.lss.validation;
2
3 import java.util.Arrays;
4
5 import javax.validation.ConstraintValidator;
6 import javax.validation.ConstraintValidatorContext;
7
8 import org.passay.LengthRule;
9 import org.passay.PasswordData;
10 import org.passay.PasswordValidator;
11 import org.passay.RuleResult;
12
13 public class PasswordConstraintValidator implements ConstraintValidator<ValidPassword, String> {
14
15     @Override
16     public void initialize(ValidPassword constraintAnnotation) {
17         //
18     }
19
20     @Override
21     public boolean isValid(String password, ConstraintValidatorContext context) {
22         PasswordValidator validator = new PasswordValidator(Arrays.asList(new LengthRule(8, 30)));
23         RuleResult result = validator.validate(new PasswordData(password));
24
25         return false;
26     }
27 }
28
29
```

Writable Smart Insert 24:1