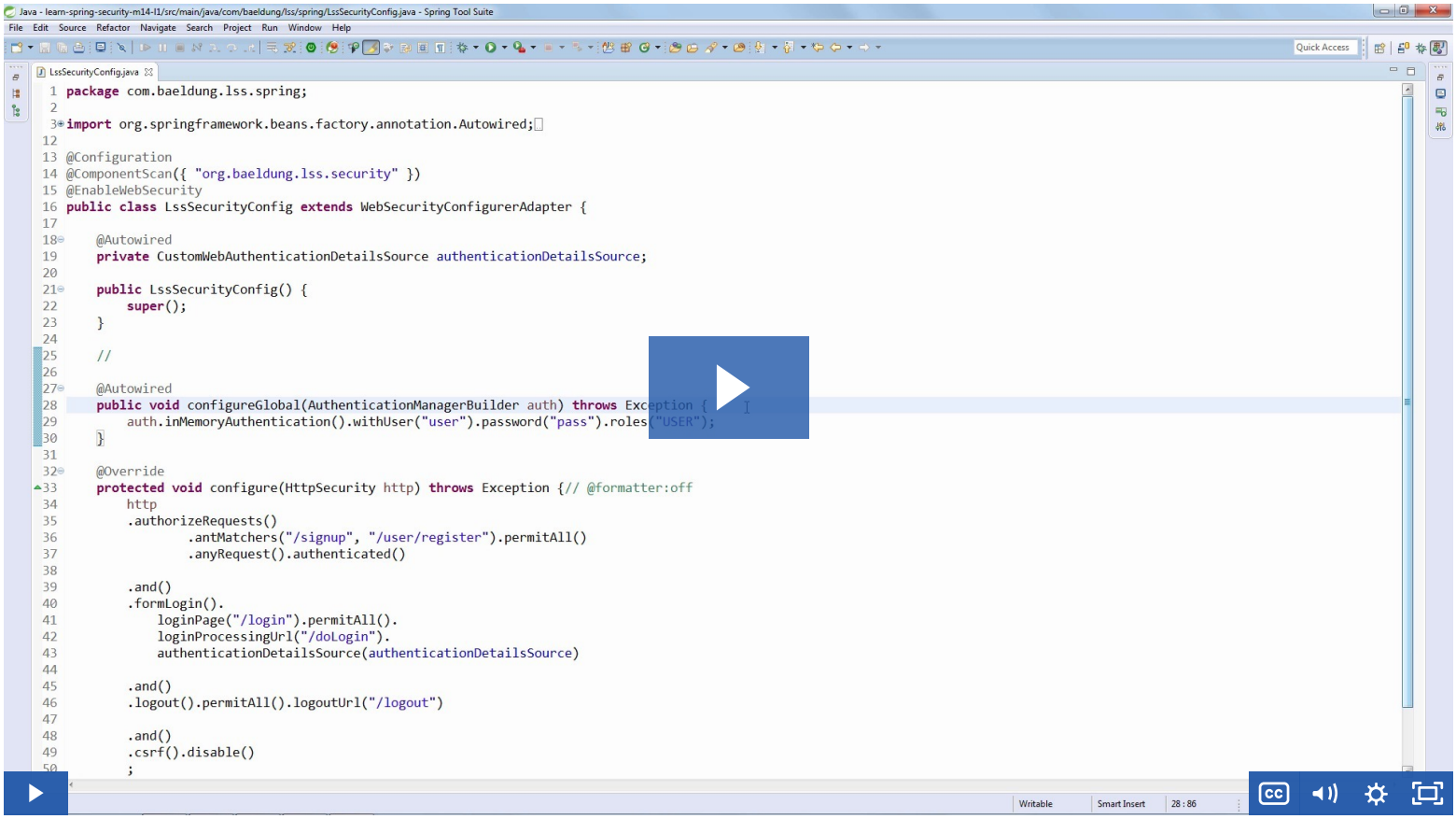
Lesson 1: A Simple Two-Factor Implementation with a Soft Token



1. Goal

The goal of this lesson is to introduce you to two-factor authentication and take you through a full implementation with Spring Security.

2. Lesson Notes

The relevant module you need to import when you're starting with this lesson is: m14-lesson1

If you want to skip and see the complete implementation, feel free to jump ahead and import: m14-lesson2

2.1. Intro to 2FA

If you've been paying even marginal attention to the security landscape in recent years, you certainly noticed that most systems nowadays are following the principle:

> Something you know and something you have

That is - simply put - the underlying principle behind 2FA.

Simply put – Two Factor Authentication (2FA) - is an effective way to significantly improve the security of a system.

Given the increasing security risks and the real risk of credentials being compromised (one way or another) - 2FA is almost fully adopted online and simply required for new systems.

With 2FA enabled, if credentials do get compromised – the attacker will still not be able to break into the system.

A **typical and common example** is withdrawing money from an ATM. You need something you know - your PIN number, and something you posses - your credit card.

So, what does the user have:

- Phone
- Hardware token
- Fingerprint

If we're going to use the phone, how will the user obtain the token they need?

One way is with the help of a mobile app - such as Google Authenticator. When you first set up 2FA on an account – you'll be able to scan a QR code with this app.

Another way is SMS - they'll basically receive an SMS message with the token - that's generally a six digit number, in the message.

Finally, a quick clarifying note before moving on to the implementation. 2FA is a new concept in the course and is unrelated to most other topics we discussed until this point (for example, it's unrelated to OAuth2). It can and should of course be used in conjunction with these other security concerns.

2.2. 2FA Impl - Extract the Tenant

Remember that, when the filter (*UsernamePasswordAuthenticationFilter*) is performing the authentication process, Spring Security allows us to set extra details on the auth request that gets passed into the auth manager.

We're going to make good use of this extension point and we're going to include a new piece of information - the tenant - extracted from the request.

This is controlled by an *authenticationDetailsSource* - so that's what we're going to define to define first:

```
@Component
public class CustomWebAuthenticationDetailsSource
  implements AuthenticationDetailsSource<HttpServletRequest, WebAuthenticationDetails> {
    @Override
    public WebAuthenticationDetails buildDetails(HttpServletRequest context) {
        return new CustomWebAuthenticationDetails(context);
    }
}
```

And the actual details:

```
public class CustomWebAuthenticationDetails extends WebAuthenticationDetails {
    private final String verificationCode;
    public CustomWebAuthenticationDetails(HttpServletRequest request) {
        super(request);
        verificationCode = request.getParameter("code");
    }
    public String getVerificationCode() {
        return verificationCode;
    }
}
```

Now of course we need to make sure that's all wired in correctly in our security config:

```
@Autowired
private CustomWebAuthenticationDetailsSource authenticationDetailsSource;
...
.formLogin().
  loginPage("/login").permitAll().
  loginProcessingUrl("/doLogin").
  authenticationDetailsSource(authenticationDetailsSource)
  ...
```

And that's it - we're now going to have access to this verification code later on in the authentication flow.

2.3. 2FA Impl - Use the Tenant

OK, now with this new piece of information available to us in the authentication flow, let's actually use it.

We're naturally going to define a new authentication provider to do that:

```
@Component
public class CustomAuthenticationProvider implements AuthenticationProvider {
    @Autowired
    private UserRepository userRepository;
    @Override
    public Authentication authenticate(Authentication auth) throws AuthenticationException {
        String username = auth.getName();
        String password = auth.getCredentials().toString();
        String verificationCode =
          ((CustomWebAuthenticationDetails) auth.getDetails()).getVerificationCode();
        User user = userRepository.findByEmail(username);
        if ((user == null) || !user.getPassword().equals(password)) {
            throw new BadCredentialsException("Invalid username or password");
        }
        return new UsernamePasswordAuthenticationToken(
          user, password, Arrays.asList(new SimpleGrantedAuthority("ROLE_USER")));
    }
    @Override
    public boolean supports(Class<?> authentication) {
        return authentication.equals(UsernamePasswordAuthenticationToken.class);
    }
}
```

Notice that we're not yet doing anything with the verification code. For now, we're simply dealing with the username and the password.

We'll get to the code after we have a look at the actual soft token support - and for that, we're going to use the Google Authenticator mobile app.

2.4. 2FA Impl - the Mobile App

So let's integrate our project with the mobile application.

But first, let's clear up what the application does. Google Authenticator is a simple application that generates a one-time password based on TOTP algorithm (Time-based One-time Password).

To be clear - notice that the one-time passwords are re-generated every few seconds by the app.

This one-time password is our verification token.

At a very high level, our system - our web application - will:

- generate a secret key
- provide the secret key to the user (via a QR-code)
- verify tokens entered by the user using this secret key

OK, so let's start the integration by first defining the Maven dependency:

```
<dependency>
    <groupId>org.jboss.aerogear</groupId>
    <artifactId>aerogear-otp-java</artifactId>
    <version>1.0.0</version>
</dependency>
```

Then we're going to add these extra fields in our *User* entity to store them:

```
public class User {
    ...
    private String secret;
    public User() {
        super();
        this.secret = Base32.random();
    }
    ...
}
```

So we are storing this secret code along with the rest of the user info.

Now, back to our authentication provider - let's add in the new logic that's going to check the code:

```
Totp totp = new Totp(user.getSecret());
try {
    if (!totp.verify(verificationCode)) {
        throw new BadCredentialsException("Invalid verfication code");
    }
} catch (Exception e) {
    throw new BadCredentialsException("Invalid verfication code");
}
```

And there we go - the library makes this logic quite trivial to write.

2.5. The Front-End Changes

As mentioned in the video, note that we also need to introduce some front end changes as well - since 2FA does affect the UX of the application - both the registration as well as the authentication process.

Remember that everything is fully implemented in the code corresponding to Lesson 2 of this Module.

If you're following along with the video and making the changes on the Lesson 1 code however, the front end artifacts that are going to be relevant are:

- *RegistrationController - registerUser*
- *qrcode.html*
- *loginPage.html*

Also remember that these new pages need to be made accessible in the security configuration.

2.6. Live 2FA

OK, time to fire up and test this implementation.

We're first going to register a new user to show the initial part of the process - scanning the QR code into the soft token application.

We're then going to authenticate.

We're going to get promoted for the token.

We're going to generate the token via our phone (this part will be off-screen).

And we're going to provide that token value.

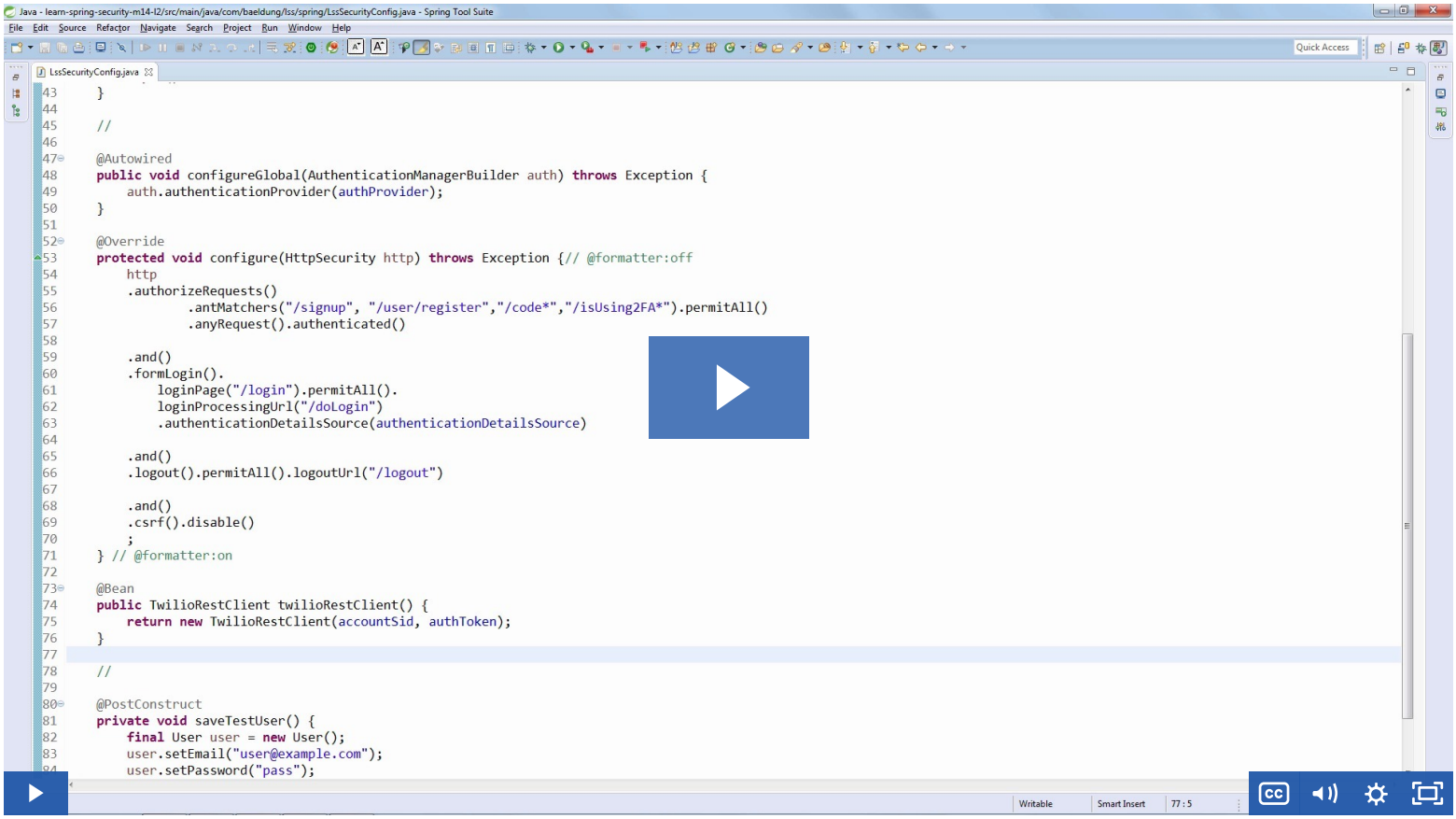
And we are logged in.

3. Resources

- multifactor authentication (MFA)

- Time-based One-time Password Algorithm

Lesson 2: A Two-Factor Impl with SMS



1. Goal

The goal of this lesson is to show you how to go from the previous 2FA implementation to a SMS driven solution.

2. Lesson Notes

The relevant module you need to import when you're starting with this lesson is: m14-lesson2

If you want to skip and see the complete implementation, feel free to jump ahead and import: m14-lesson3

2.1. 2FA via SMS

In the previous lesson we implemented 2FA with the Google Authenticator mobile app. That works perfectly fine if the users of your application have a smartphone - but for some applications that's not necessarily a correct assumption.

So, now, let's actually switch from QR codes and mobile apps to a more traditional way of using the phone - SMS.

SMS is of course much more widespread and doesn't require any special kind of phone - any mobile phone out there does SMS just fine.

Let's switch things around here and have a look at the high level flow first:

- the user logs in with username and password
- then, they quickly get an SMS with a unique (and time sensitive) token/code
- the system prompts them for that code - they fill it in and they're authenticated

And that's it - a simple 2FA flow that will work for most applications.

Now, for that, we'll obviously need some SMS sending capabilities in our application - which, luckily, is easy to do now. It wasn't so easy a few years ago, but things are moving quite fast in this space.

We're going to make use of Twilio - an online service that does SMS very well (among many other things).

Note that you'll need to sign up to Twilio yourself, and create a number there (it's free) in order to be able to set up the code properly.

2.2. The Implementation

OK, so let's get right into the code - first, we need the Maven dependency for Twilio:

```
<dependency>
    <groupId>com.twilio.sdk</groupId>
    <artifactId>twilio-java-sdk</artifactId>
    <version>3.4.5</version>
</dependency>
```

Let's now define the Twilio details as properties in *application.properties*:

```
twilio.sid=YOUR_ACCOUNT_SID
twilio.token=YOUR_AUTH_TOKEN
twilio.sender=YOUR_TWILIO_PHONE_NUMBER
```

That's basically all that we need to have configured.

Now, we're going to define the client bean for Twilio:

```
@Value("${twilio.sid}")
private String accountSid;
@Value("${twilio.token}")
private String authToken;
@Bean
public TwilioRestClient twilioRestClient() {
    return new TwilioRestClient(accountSid, authToken);
}
```

OK, so with the client defined and usable, let's actually write the logic that's going to be doing the sending of the unique code to the client.

First, we'll define a controller and prepare the Twilio client there:

```
@Controller
public class VerificationCodeController {
    @Autowired
    private TwilioRestClient twilioRestClient;
}
```

Then, let's create the verification code sending logic:

```
@Value("${twilio.sender}")
private String senderNumber;
...
@RequestMapping(value = "/code", method = RequestMethod.GET)
@ResponseStatus(HttpStatus.OK)
public void sendCode(@RequestParam("username") String username) throws TwilioRestException {
    User user = userRepository.findByEmail(username);
    if (user == null) {
        return;
    }
    List<NameValuePair> params = new ArrayList<NameValuePair>();
    String code = new Totp(user.getSecret()).now();
    params.add(new BasicNameValuePair("Body", "The verification code is " + code));
    params.add(new BasicNameValuePair("To", user.getPhone()));
    params.add(new BasicNameValuePair("From", senderNumber));
    MessageFactory messageFactory = twilioRestClient.getAccount().getMessageFactory();
    Message message = messageFactory.create(params);
}
```

The interesting this here is that - we're actually going to be able to use most of the logic we have developed in the previous lesson.

The custom details, the authentication provider, even the Totp library - everything's the same.

The only thing that's different is basically how the user obtains the verification code.

In the previous lesson they obtained it via the mobile app - and here they get it via SMS.

But, once they have the code, everything's exactly the same from that point on.

2.3. Securing the Send Code Operation

As discussed in the lesson, the */code* URI needs to be secured with Basic Authentication.

Here's a simple way we can do that.

First - let's secure the HTTP endpoint here:

```
@Configuration
@Order(Ordered.HIGHEST_PRECEDENCE)
public static class BasicSecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.antMatcher("/code*").authorizeRequests().anyRequest().hasRole("TEMP_USER").and().httpBasic();
    }
}
```

Now, notice that we're using a role called *TEMP_USER*.

Let's make sure our Principals actually get that role when authenticating:

```
if (auth.getDetails() instanceof CustomWebAuthenticationDetails) {
    ...
} else {
    return new UsernamePasswordAuthenticationToken(user.getEmail(), password, Arrays.asList(new SimpleGrantedAuthority("ROLE_TEMP_USER")));
}
```

Notice that we're looking at the type of authentication request. That's because - in case of an HTTP Request that's using Basic Authentication, the type of requent will be *WebAuthenticationDetails* (instead of *CustomWebAuthenticationDetails*).

Finally, on the front-end site, we'll make sure that request is actually secured and using Basic Auth:

```
function doNext(){
    $("#firstDiv").hide();
    $("#secondDiv").show();
    var username = $("#username").val();
    var password = $("#password").val();
    $.ajax({
        url: "/code",
        type: "GET",
        beforeSend: function(xhr){xhr.setRequestHeader("Authorization", "Basic " + btoa(username + ":" + password));}
     });
}
```

So - that's it - we can now trigger this logic from the client side and make sure the code is being sent to the user during the login process. So the user will have the code when they log in.

Also, a quick side-note here. Given that the implementation is a bit more complex here, it's a good idea to keep an eye on the final state of the lesson, where everything's already set up - while you're working through this lesson.

2.4. 2FA with SMS Live Demo

OK, let's fire off the system and let's go through a registration sequence first.

We're registering a simple user and specifying a phone number here.

Now, we're going to authenticate.

And of course we're going to get promoted for the token.

We'll generate the token via SMS (this part will be off-screen) - and provide the token value.

And we are correctly authenticated and logged into to system.