Contents

Course Overview

Spring Boot is an open source Java-based framework used to create microservices. Spring Boot makes it easy to create standalone production grade Spring-based applications that just work. Spring Boot can be thought of as an extension of the Spring framework, which is used to bring diverse components together to build enterprise applications. Even a simple Spring applications may involve multiple dependencies and managing these dependencies along with their versions can be quite complex.

Spring Boot is an opinionated framework which makes building applications easy by providing sensible defaults for most of the libraries that you need yet, allowing you the flexibility of configuring specific classes. In this course, we'll build Spring Boot microservices to configure our application so that it's able to send emails. We'll use interceptors to pre-process requests to and responses from our application. We'll also integrate with the Twilio platform to send text messages and make phone calls. Once you're done with this course, you'll be able to integrate with external services, such as the Java mail sender and the Twilio platform from within your Spring Boot application. You will also be able to secure your application using in-memory users and users stored in a database, all using Spring security.

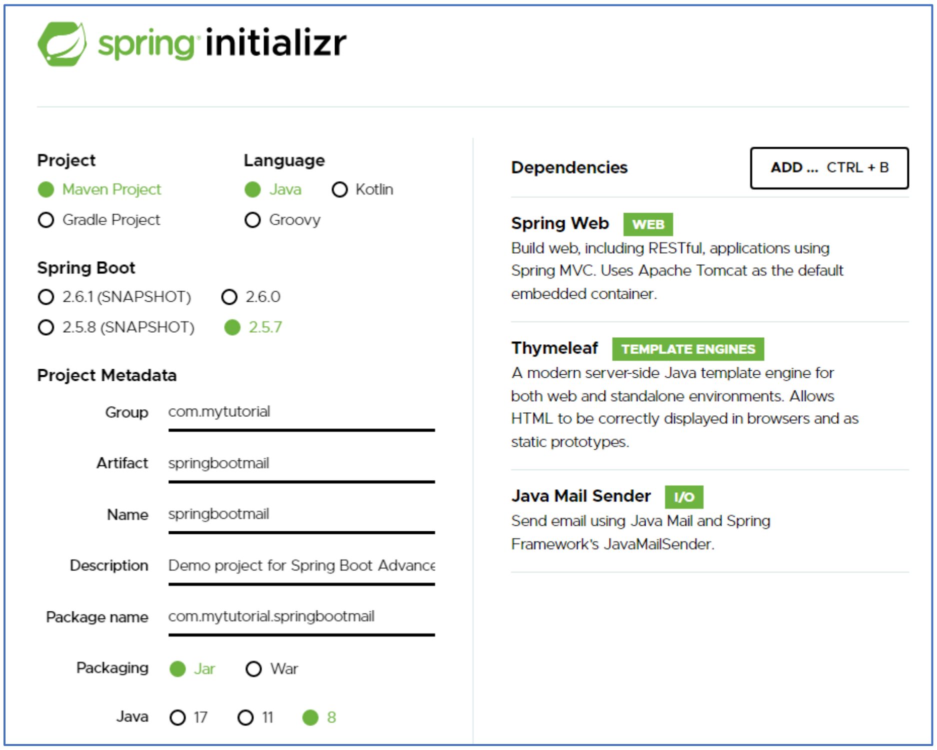
Sending Emails Using the JavaMailSender API

In this Spring Boot demo, we'll see how we can configure our Spring Boot service to send email using the JavaMailSender. We'll see how we can send emails with attachments and without attachments. Here we are on the spring initializr page.

1. Start Maven Project

Once again, we'll set up a Maven Project. We are coding in Java using Spring Boot 2.5.7. I'm going to scroll down and fill in the Project Metadata. I start off with a new project because I need additional dependencies in order to be able to send mail from Spring Boot. Continue to use com.mytutorial.springbootmail as your package.

http://start.spring.io



Project
● Maven Project
○ Gradle Project

Language
● Java ○ Kotlin
○ Groovy

Spring Boot
○ 2.6.1 (SNAPSHOT) ○ 2.6.0
○ 2.5.8 (SNAPSHOT) ● 2.5.7

Project Metadata
Group: com.mytutorial
Artifact: springbootmail
Name: springbootmail
Description: Demo project for Spring Boot Advance
Package name: com.mytutorial.springbootmail
Packaging: ● Jar ○ War
Java: ○ 17 ○ 11 ● 8

Dependencies

Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Thymeleaf TEMPLATE ENGINES
A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes.

Java Mail Sender I/O
Send email using Java Mail and Spring Framework's JavaMailSender.

The real change in our project will be in our Maven file where we specify dependencies.

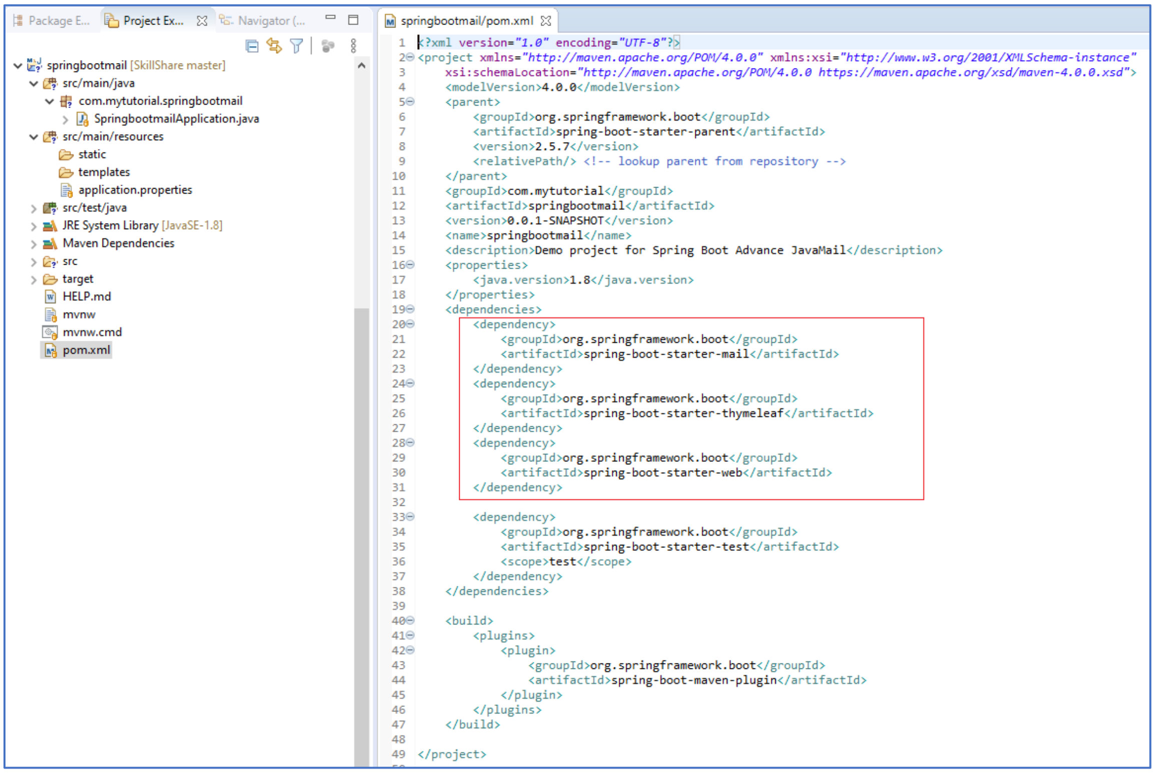
We'll continue to use the **Spring Web** dependencies to set-up a web application. We'll continue to use the **Thymeleaf** template engine as well. Think of this as a standard for any kind of web app. And finally, let's add the starter dependency descriptor that will allow us to build a mail sending service. And Add dependency of the **Java Mail Sender**. Select this, add these to your project dependencies. Click on the GENERATE option.

A zip file will be downloaded onto your local machine. Make sure that it's in the right location, unzip the file and set-up your Maven project in Eclipse using that file.

We are already familiar and comfortable with these steps.

Let's head over to pom.xml and see the starter dependencies that have been added to our pom.

pom.xml



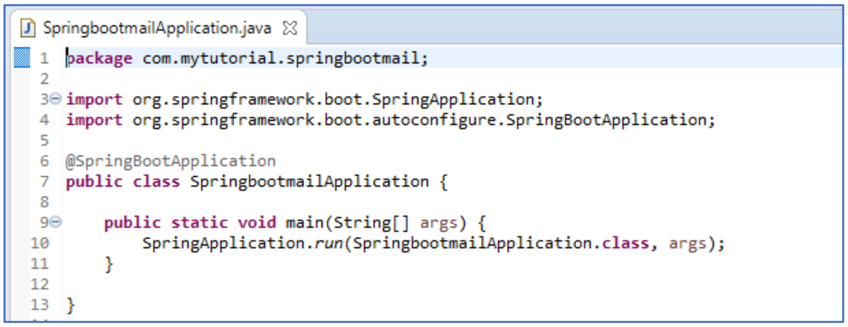
```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.5.7</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.mytutorial</groupId>
    <artifactId>springbootmail</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>springbootmail</name>
    <description>Demo project for Spring Boot Advance JavaMail</description>
    <properties>
        <java.version>1.8</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-mail</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

spring-boot-starter-mail, this is the new dependency which allows us to use the JavaMailSender service. We still have spring-boot-starter-thymeleaf and spring-boot-starter-web.

2. Inspect Main Class of Entry Point Spring Boot Application

There is no real change to the code that makes up the main entry point of our application. The class SpringbootApplication simply calls Application.run, and has the annotation **@SpringBootApplication**.

SpringbootmailApplication.java



```java
package com.mytutorial.springbootmail;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringbootmailApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringbootmailApplication.class, args);
    }

}
```

3. Setup IMAP from Mail provider in application configuration properties

IMAP is what our Java mail sending service uses in order to send emails.

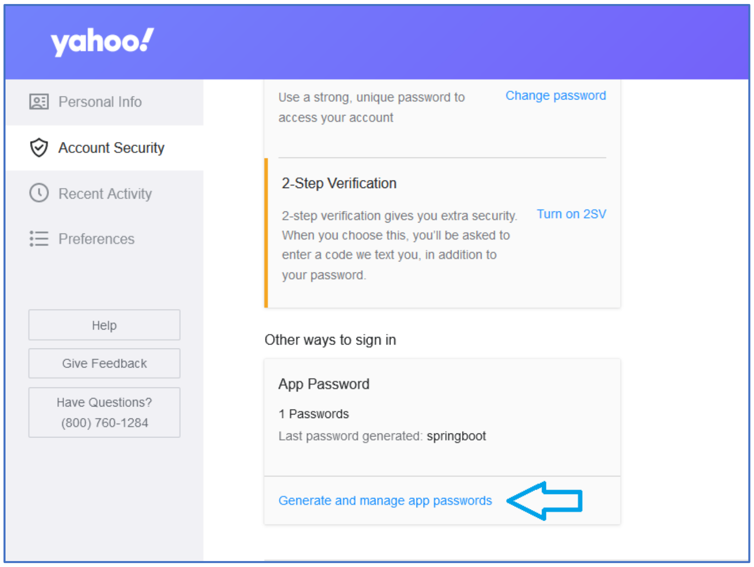
To Setup Your Yahoo.com Account with Your Email Program Using IMAP
To access your Yahoo.com email account from a email program, you'll need the IMAP and SMTP settings below:

Yahoo.com (Yahoo! Mail) IMAP Server	: imap.mail.yahoo.com
IMAP port	: 993
IMAP security	: SSL / TLS
IMAP username	: Your full email address
IMAP password	: Your Yahoo.com password

Yahoo.com (Yahoo! Mail) SMTP Server	: smtp.mail.yahoo.com
SMTP port	: 465
SMTP security	: SSL / TLS
SMTP username	: Your full email address
SMTP password	: Your Yahoo.com password

Less secure apps need to enable so that yahoo mail allowing you to send email message from our java spring application. Follows is the step :

1. Position the mouse cursor over your name in the top Yahoo! Mail navigation bar.
2. Select Account Info on the sheet that appears.
3. Open the Account security category.



Use a strong, unique password to access your account

2-Step Verification

2-step verification gives you extra security. When you choose this, you'll be asked to enter a code we text you, in addition to your password.

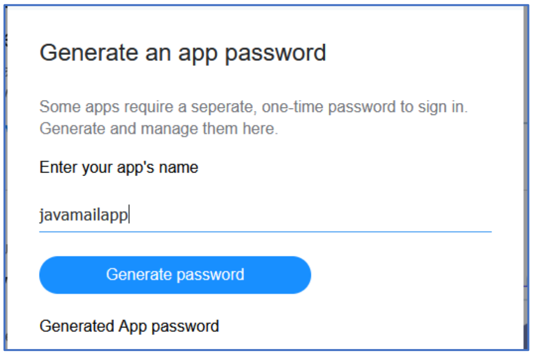
Other ways to sign in

App Password

1 Passwords

Last password generated: springboot

4. Select Generate app password under Account security.



Generate an app password

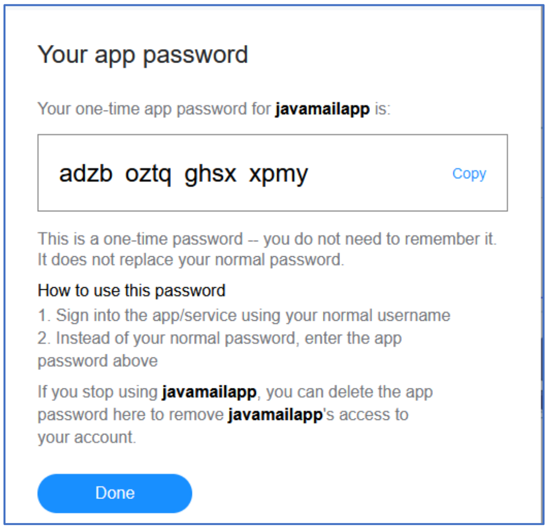
Some apps require a seperate, one-time password to sign in. Generate and manage them here.

Enter your app's name

javamailapp

Generated App password

5. Choose Other App from the list. type the name of the program over Enter custom name.



Your app password

Your one-time app password for **javamailapp** is:

adzb oztq ghsx xpmy

This is a one-time password -- you do not need to remember it. It does not replace your normal password.

How to use this password

1. Sign into the app/service using your normal username
2. Instead of your normal password, enter the app password above

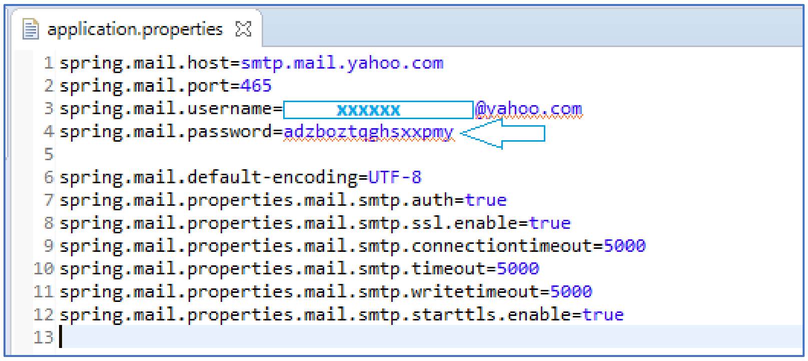
If you stop using **javamailapp**, you can delete the app password here to remove **javamailapp**'s access to your account.

6. Click Generate.
7. Use the generated password which is being displayed BOTH IN IMAP and SMTP section in your email account settings in Hotter.
8. Click Done.

If you're using a service other than Yahoo Mail, these configurations need to be specified for that email service.

Now that we've configured our email account such that it can send emails, we can head over to Eclipse and set-up some properties in the **application.properties** file. These are properties that correspond to the JavaMailSender service that we'll be using. You can see that all of these properties start with the prefix spring.mail.

Use the generated password from previous step to set password value in **application.properties**. do not use email plain "real" password for security reason.



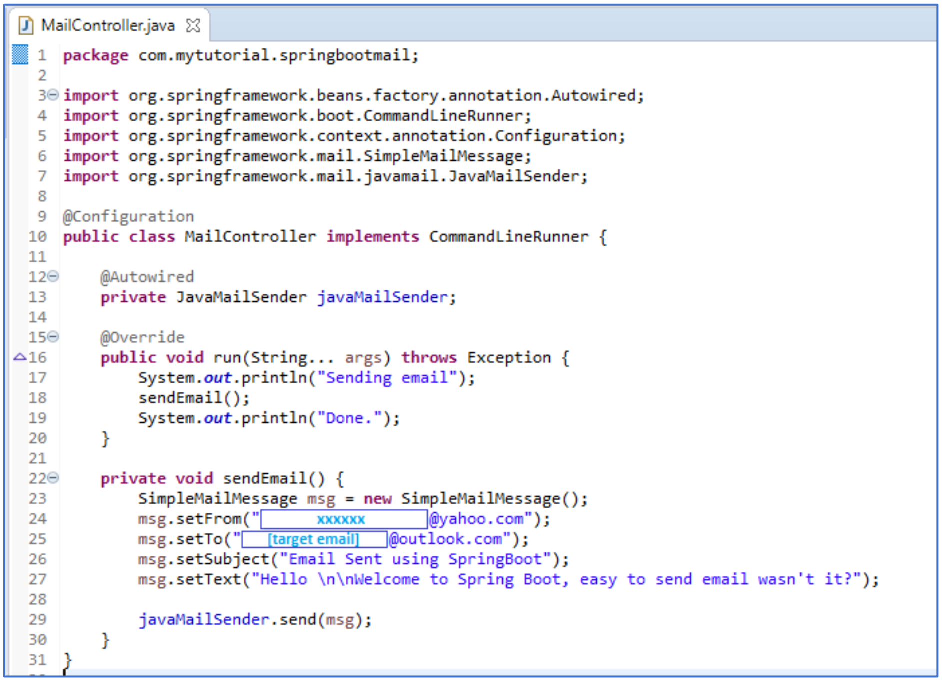
```
spring.mail.host=smtp.mail.yahoo.com
spring.mail.port=465
spring.mail.username=xxxxxx@yahoo.com
spring.mail.password=adzboztqghsxxpmy

spring.mail.default-encoding=UTF-8
spring.mail.properties.mail.smtp.auth=true
spring.mail.properties.mail.smtp.ssl.enable=true
spring.mail.properties.mail.smtp.connectiontimeout=5000
spring.mail.properties.mail.smtp.timeout=5000
spring.mail.properties.mail.smtp.writetimeout=5000
spring.mail.properties.mail.smtp.starttls.enable=true
```

The username here is the account from which you will be sending the email, *xxxxxxx@yahoo.com* is the email we have chosen. You also need to specify the password to log into this email. So make sure that this is an email address that everyone on your team can work with because it's not really secure. The password is specified here in plain text. Make sure smtp.auth is set to true. We have a connectiontimeout, writetimeout, and so on. The remaining configuration properties are good sensible defaults that you can use with the JavaMailSender.

4. Create Sending Mail Class

The actual code to send email we'll write in the **MailController.java** file. Now, despite its name, it's not really a controller. It's tagged using **@Configuration** indicating that it specifies configuration properties for our application.

MailController.java



```java
package com.mytutorial.springbootmail;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.context.annotation.Configuration;
import org.springframework.mail.SimpleMailMessage;
import org.springframework.mail.javamail.JavaMailSender;

@Configuration
public class MailController implements CommandLineRunner {

    @Autowired
    private JavaMailSender javaMailSender;

    @Override
    public void run(String... args) throws Exception {
        System.out.println("Sending email");
        sendEmail();
        System.out.println("Done.");
    }

    private void sendEmail() {
        SimpleMailMessage msg = new SimpleMailMessage();
        msg.setFrom("xxxxxx@yahoo.com");
        msg.setTo("[target email]@outlook.com");
        msg.setSubject("Email Sent using SpringBoot");
        msg.setText("Hello \n\nWelcome to Spring Boot, easy to send email wasn't it?");

        javaMailSender.send(msg);
    }
}
```

For the purposes of this demo, I have the MailController implement the **CommandLineRunner** interface. This means that this particular bean will be considered an executable by Spring.

So, once the Spring application starts up, an object of this controller class will be instantiated and executed by invoking it's run method. Implementing the **CommandLineRunner** interface makes this bean an executable bean.

Now, this bean makes use of the JavaMailSender in order to send email. So, I have tagged the JavaMailSender member variable using **@Autowired** so that it's automatically injected into this bean.

We also need to implement the run method that takes in a variable number of arguments. This is part of the CommandLineRunner interface.

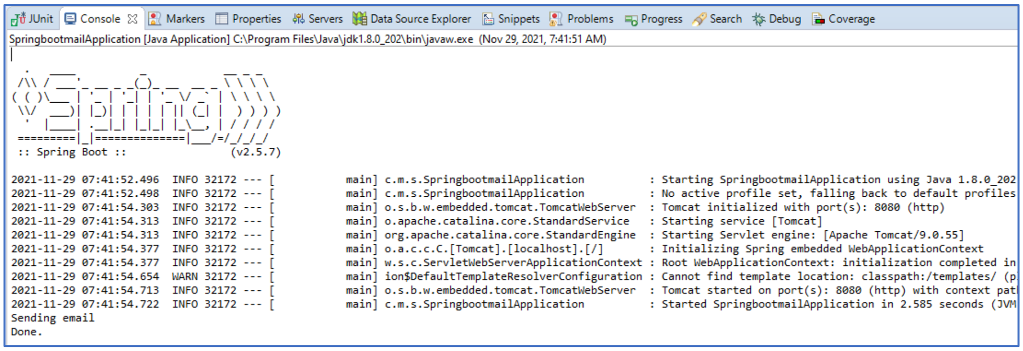
Within this run method, we print out to screen sending email, we then invoke the sendEmail method, and then say Done.

The code for actually using the JavaMailSender to send email is in this sendEmail function.

We instantiate an object of the type **SimpleMailMessage**. It's a simple mail message because it doesn't contain any attachments. You set the To field of the message, you're going to send an email to *[target email]@outlook.com*.

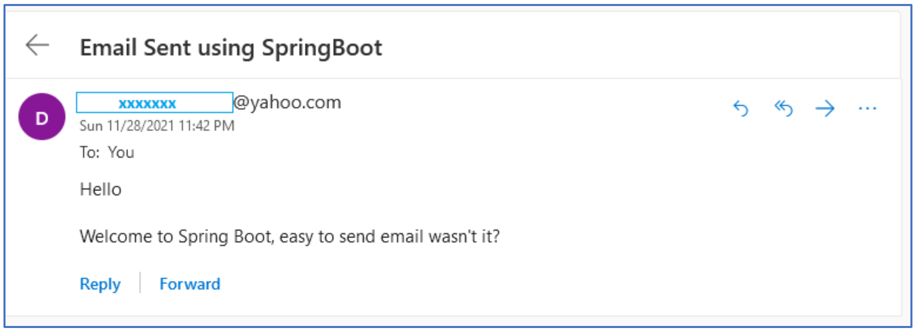
You can then set the subject and text as well. Our subject says Email sent using SpringBoot and the text says *Hello! Welcome to Spring Boot, it's easy to send email*, and it is. Then, simply invoke **javaMailSender.send**.

Let's verify that everything works. Go ahead and run this code. You'll see the message Sending email and then Done.



```
SpringbootmailApplication [Java Application] C:\Program Files\Java\jdk1.8.0_202\bin\javaw.exe  (Nov 29, 2021, 7:41:51 AM)

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.5.7)

2021-11-29 07:41:52.496  INFO 32172 --- [           main] c.m.s.SpringbootmailApplication          : Starting SpringbootmailApplication using Java 1.8.0_202
2021-11-29 07:41:52.498  INFO 32172 --- [           main] c.m.s.SpringbootmailApplication          : No active profile set, falling back to default profiles
2021-11-29 07:41:54.303  INFO 32172 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2021-11-29 07:41:54.313  INFO 32172 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2021-11-29 07:41:54.313  INFO 32172 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.55]
2021-11-29 07:41:54.377  INFO 32172 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2021-11-29 07:41:54.377  INFO 32172 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in
2021-11-29 07:41:54.654  WARN 32172 --- [           main] ion$DefaultTemplateResolverConfiguration : Cannot find template location: classpath:/templates/ (p
2021-11-29 07:41:54.713  INFO 32172 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context pat
2021-11-29 07:41:54.722  INFO 32172 --- [           main] c.m.s.SpringbootmailApplication          : Started SpringbootmailApplication in 2.585 seconds (JVM
Sending email
Done.
```

Now, you can switch over to *[target email]@outlook.com*, log into the account to which you've sent the email and see whether [Target Email] has received the email.



Email Sent using SpringBoot

xxxxxxx@yahoo.com
Sun 11/28/2021 11:42 PM
To: You

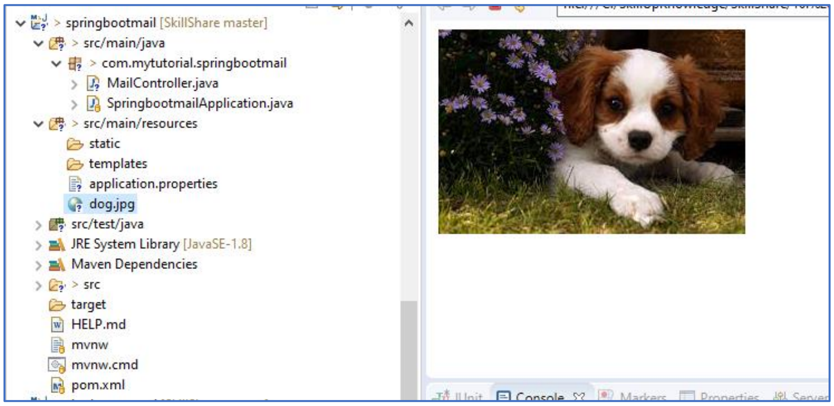
Hello

Welcome to Spring Boot, easy to send email wasn't it?

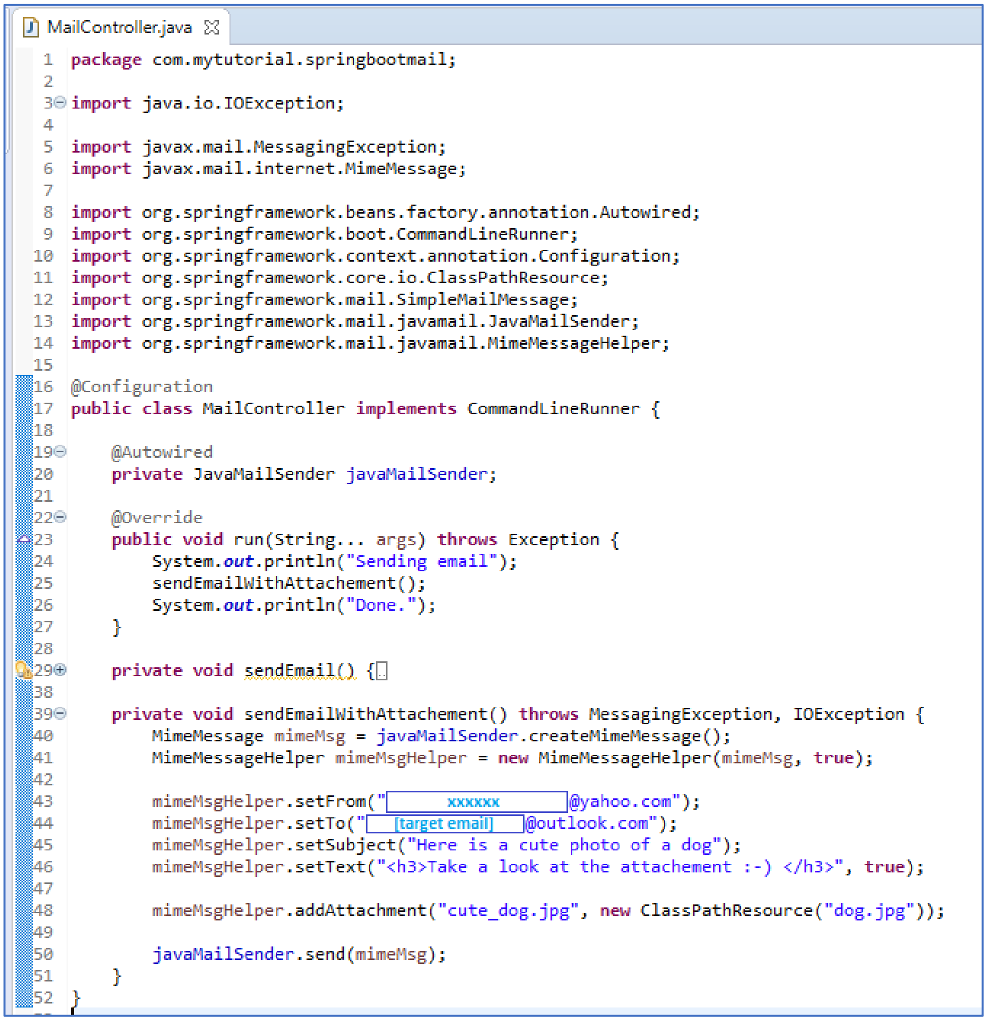
And yes, indeed he has. You can click through and see the mail that we received from *xxxxxxx@yahoo.com*. This is the same mail that we had sent using our Java mail sending application.

5. Send Email with Attachement

Let's head back to Eclipse here. And this time, we'll configure the JavaMailSender to send an email that includes an attachment. I'm going to add in a new file within src/main/resources which we'll use as the attachment. This is the dog.jpg file that you see here.



I'll also update the code in my MailController class to send an email with an attachment.



```java
package com.mytutorial.springbootmail;

import java.io.IOException;

import javax.mail.MessagingException;
import javax.mail.internet.MimeMessage;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.io.ClassPathResource;
import org.springframework.mail.SimpleMailMessage;
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.mail.javamail.MimeMessageHelper;

@Configuration
public class MailController implements CommandLineRunner {

    @Autowired
    private JavaMailSender javaMailSender;

    @Override
    public void run(String... args) throws Exception {
        System.out.println("Sending email");
        sendEmailWithAttachement();
        System.out.println("Done.");
    }

    private void sendEmail() {

    private void sendEmailWithAttachement() throws MessagingException, IOException {
        MimeMessage mimeMsg = javaMailSender.createMimeMessage();
        MimeMessageHelper mimeMsgHelper = new MimeMessageHelper(mimeMsg, true);

        mimeMsgHelper.setFrom("xxxxxx@yahoo.com");
        mimeMsgHelper.setTo("[target email]@outlook.com");
        mimeMsgHelper.setSubject("Here is a cute photo of a dog");
        mimeMsgHelper.setText("<h3>Take a look at the attachement :-) </h3>", true);

        mimeMsgHelper.addAttachment("cute_dog.jpg", new ClassPathResource("dog.jpg"));

        javaMailSender.send(mimeMsg);
    }
}
```

Now the basic components of the MailController remains the same. We still have the **@Configuration**. We still Autowired the JavaMailSender to be injected and we still implement the CommandLineRunner and we still override the run method.

The real change is in the actual mail sending. We invoke the method sendEmailWithAttachment and let's scroll down below and see the changes that we made to our code here.

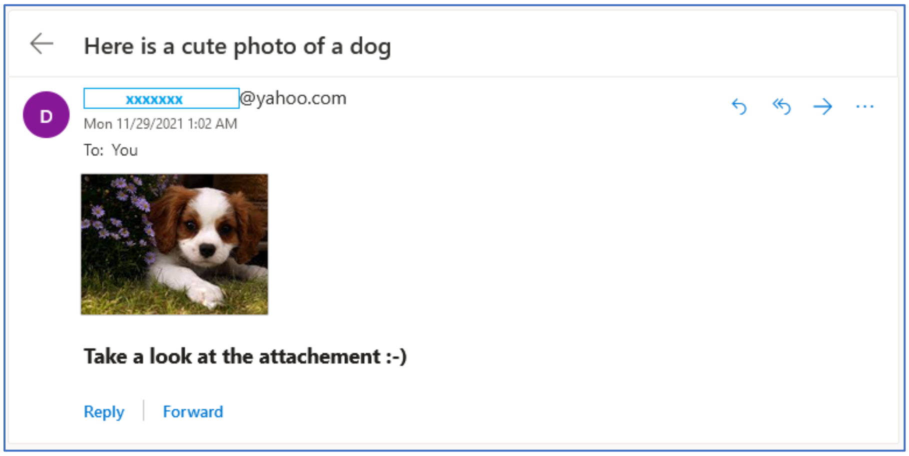
The first change that you'll notice is in the object that we work with. Rather than instantiating a simple mail message, we use the JavaMailSender to create a MimeMessage. This MimeMessage used along with the MimeMessageHelper allows us to send emails with attachments as well as inline resources. We then use this mimeMsg to instantiate the MimeMessageHelper on line 39. And the second input argument true indicates that this is a multi-part message.

Typically, when you're sending attachments, you want to send multi-part message to true. We'll then configure the standard details for any email message. We'll set the To field to [target email]@outlook.com, the subject to Here is a cute photo of a dog! and we'll set the text to Take a look at the attachment.

But notice that we have specified some HTML content for our text. The second input argument true that we have specified to setText tells the mimeMsgHelper that the text should be interpreted as HTML. The HTML that we have specified here is of course, very simple, but you can set any arbitrary HTML that you want to. Go ahead and invoke addAttachment on the MimeMessageHelper, this we do on line 48.

And, let's attach a ClassPathResource. This ClassPathResource will automatically look under Resources folder for the dog.jpg file. And we'll attach that file with the name cute_dog.jpg. Then we just call javaMailSender.send and send this mimeMsg.

All that's left to do is to run this code and test it out. It seems like we're done with sending the mail with attachment. Let's head over to *[target email]@outlook.com*. And you'll see that there is another message here from Alice. The message seems to have an attachment, let's click through and take a look at it.



Here is a cute photo of a dog

D

xxxxxxx@yahoo.com

Mon 11/29/2021 1:02 AM

To: You

Take a look at the attachement :-)

And here is our cute dog.jpg file. You can click on it. The attachment has been sent successfully.

Using Interceptors

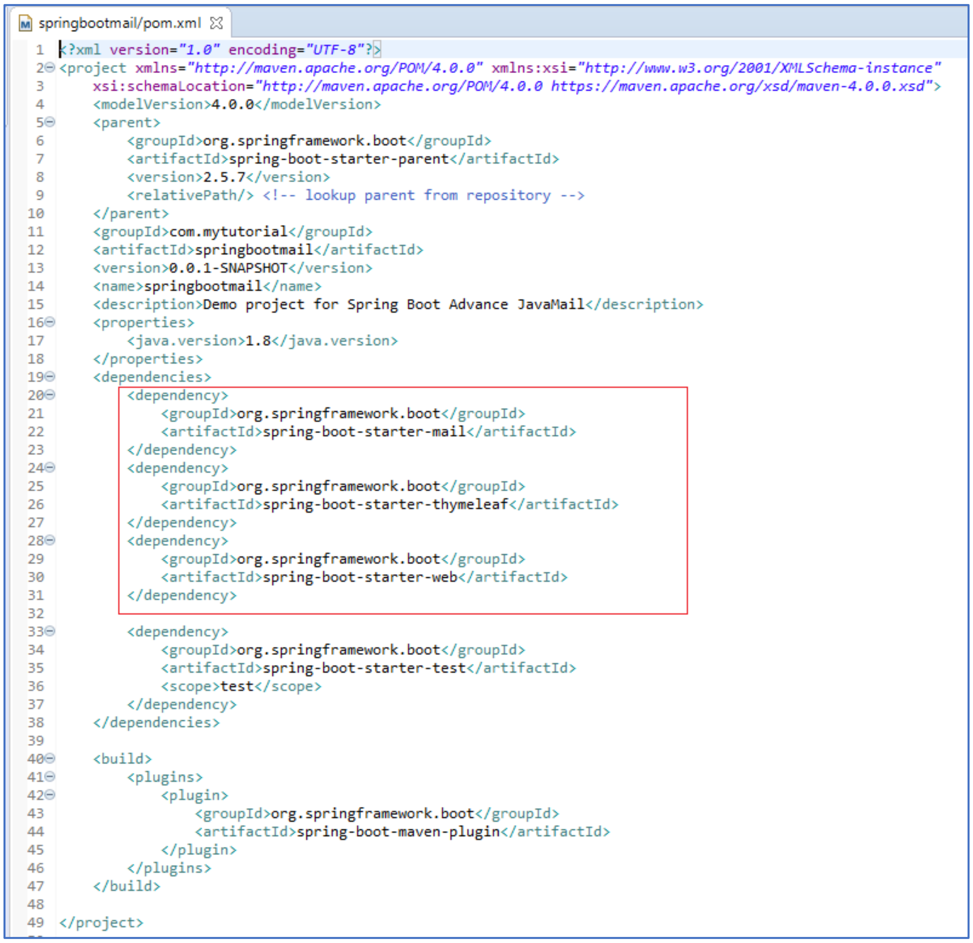
In this demo, we'll see how we can work with interceptors in Spring. ***Interceptors*** are an integral part of the Spring MVC web framework. When working with Spring MVC, any incoming web request is sent to the dispatcher servlet, which then looks up the handler mappings that you have configured to find the right method to handle this particular incoming request.

This is where interceptors come in. Interceptors in Spring allow you to intercept client requests to handler mappings and process these requests before they are handled by our controller methods. You can use interceptors to make sure all of the input parameters are in the right format.

You can use them to add-in additional parameters that you want to send to your handler. You can also use interceptors to perform some kind of action in the background for every client request. In this demo, we'll see how we can use Spring interceptors.

1. Prepare Spring Boot Dependency library in Pom.xml

in order to send an email to a client each time we receive a particular kind of request. Here are the dependencies that I specify in my **pom.xml** file, **spring-boot-starter-mail**, **spring-boot-starter-thymeleaf**, and **spring-boot-starter-web**.

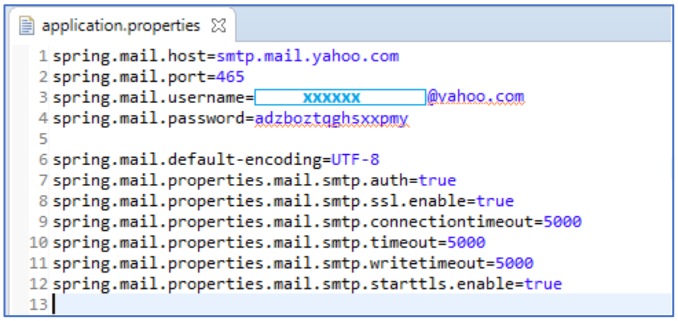


springbootmail/pom.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.5.7</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.mytutorial</groupId>
    <artifactId>springbootmail</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>springbootmail</name>
    <description>Demo project for Spring Boot Advance JavaMail</description>
    <properties>
        <java.version>1.8</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-mail</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

2. Prepare the Configuration Properties

Because we are using the **JavaMailSender** service, we'll configure all of the properties for this mail sender within **application.properties**.

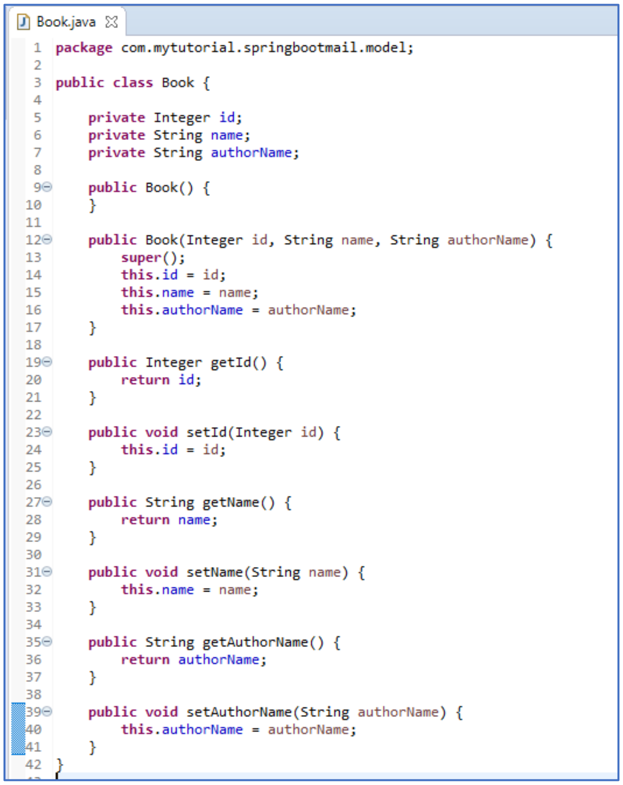


```
spring.mail.host=smtp.mail.yahoo.com
spring.mail.port=465
spring.mail.username=xxxxxx@yahoo.com
spring.mail.password=adzboztqghsxxpmy

spring.mail.default-encoding=UTF-8
spring.mail.properties.mail.smtp.auth=true
spring.mail.properties.mail.smtp.ssl.enable=true
spring.mail.properties.mail.smtp.connectiontimeout=5000
spring.mail.properties.mail.smtp.timeout=5000
spring.mail.properties.mail.smtp.writetimeout=5000
spring.mail.properties.mail.smtp.starttls.enable=true

```

We'll continue to use YahooMail, `smtp.mail.yahoo.com` is the host, port is `465`, and the username is *xxxxx@yahoo.com*.

3. Prepare the Model Object

In order to respond to incoming web requests, I'm going to set-up a very simple in-memory application, which returns a few books to the user. Here is my model, Book.java.

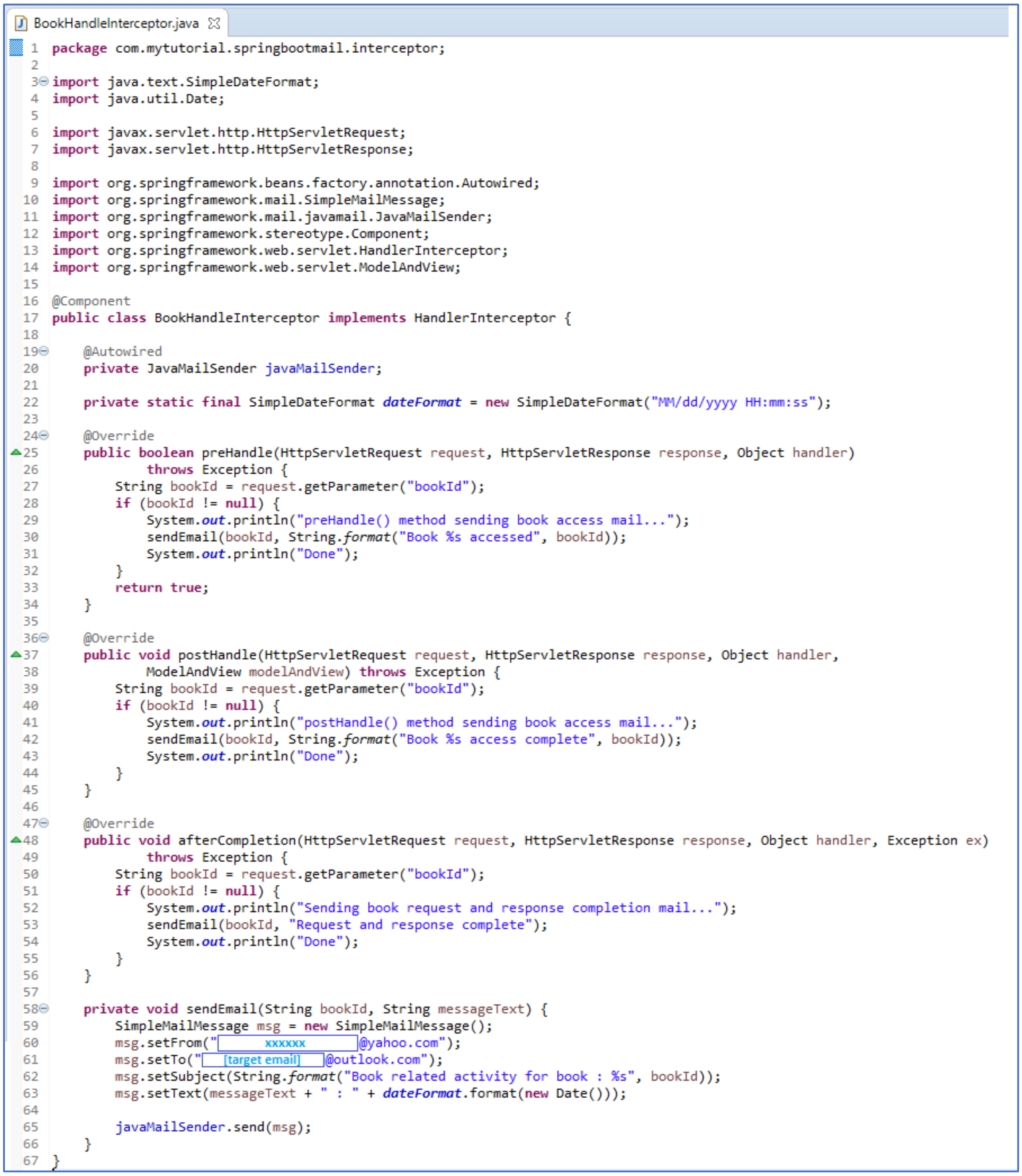


```java
package com.mytutorial.springbootmail.model;

public class Book {

    private Integer id;
    private String name;
    private String authorName;

    public Book() {
    }

    public Book(Integer id, String name, String authorName) {
        super();
        this.id = id;
        this.name = name;
        this.authorName = authorName;
    }

    public Integer getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getAuthorName() {
        return authorName;
    }

    public void setAuthorName(String authorName) {
        this.authorName = authorName;
    }
}
```

For every book, we have the id, name, and the authorName. We have constructors here and getters and setters for all of the member variables for our Book model class, which is just a POJO, a Plain Old Java Object.

4. Create Interceptor Class

All of the interesting code is in the **BookHandlerInterceptor.java** file.



```java
package com.mytutorial.springbootmail.interceptor;

import java.text.SimpleDateFormat;
import java.util.Date;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.mail.SimpleMailMessage;
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.stereotype.Component;
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;

@Component
public class BookHandleInterceptor implements HandlerInterceptor {

    @Autowired
    private JavaMailSender javaMailSender;

    private static final SimpleDateFormat dateFormat = new SimpleDateFormat("MM/dd/yyyy HH:mm:ss");

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)
            throws Exception {
        String bookId = request.getParameter("bookId");
        if (bookId != null) {
            System.out.println("preHandle() method sending book access mail...");
            sendEmail(bookId, String.format("Book %s accessed", bookId));
            System.out.println("Done");
        }
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,
            ModelAndView modelAndView) throws Exception {
        String bookId = request.getParameter("bookId");
        if (bookId != null) {
            System.out.println("postHandle() method sending book access mail...");
            sendEmail(bookId, String.format("Book %s access complete", bookId));
            System.out.println("Done");
        }
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)
            throws Exception {
        String bookId = request.getParameter("bookId");
        if (bookId != null) {
            System.out.println("Sending book request and response completion mail...");
            sendEmail(bookId, "Request and response complete");
            System.out.println("Done");
        }
    }

    private void sendEmail(String bookId, String messageText) {
        SimpleMailMessage msg = new SimpleMailMessage();
        msg.setFrom("xxxxxx@yahoo.com");
        msg.setTo("[target email]@outlook.com");
        msg.setSubject(String.format("Book related activity for book : %s", bookId));
        msg.setText(messageText + " : " + dateFormat.format(new Date()));

        javaMailSender.send(msg);
    }
}
```

The BookHandlerInterceptor implements the **HandlerInterceptor interface**. The HandlerInterceptor comes with three methods that we have to implement. The first method is executed before a request is sent to a handler. The second method after the request is sent to the handler. And, the third method after the completion of the request when the view is rendered. I've used the **@Component** annotation on the BookHandlerInterceptor, indicating that it's a component managed by Spring.

Now, our BookHandlerInterceptor will be using the **JavaMailSender** to send email. I have used the **@Autowired** annotation here to have this injected into my class automatically. I've also instantiated a simple date formatter here to format dates in the form of month, day, year, hour, minute, and seconds.

- **preHandle Method**

The first method that we implement from the HandlerInterceptor interface is the **preHandle method**. This takes in three input arguments, the HTTP request, the response, and the object handler. As its name suggests, this preHandle method basically is invoked before the actual handler is executed. So, before the handler in your controller code is executed, preHandle will be called to process your request. When the preHandle method is invoked, the view for the corresponding response has not been generated yet. This is where you can pre-process the request that the actual handler will see. Add-in any additional parameters, if needed. The preHandler is also the method where you can inject bits of information from other components in your system that your handler might need.

Our preHandler will basically check to see if the bookId parameter is set on the incoming request. This is the check that we perform on line 28. So long as there is a bookId parameter, we'll print out a log message to screen and then send an email to a specific account indicating that this is the book that was accessed.

Our preHandler here thus logs all access to the individual books in our library via email. Once the email has been sent out, we'll print out, Done, and we return, true, from this preHandle method, indicating the request has been handled.

- **postHandle Method**

The next method that we implement from the HandlerInterceptor interface is the **postHandle method**. It takes as an input argument, the HTTP request, the response, the handler object, and the model and view. This is the model and view that will be rendered to the user.

The postHandle method is invoked after your handler code has been executed, but before the view is rendered to the user. This handler is a great place to add-in any additional model parameters that you want the view to understand and display.

We'll keep our code simple. We'll check whether the bookId parameter is present in the request, and we'll send a book access email indicating the book access is now complete.

- **afterCompletion Method**

And finally, if you scroll down, you'll see the third method that our interceptor implements, **afterCompletion**. This takes as an input argument, the request response, the handler object, as well as any exception that may have been thrown by our handler mapping.

This afterCompletion handler is invoked once the response is completely processed by our handler. So, after the handler code is executed, when the request is complete, and the view has been rendered, that's when afterCompletion is called. Within this handler, you have not just the request and response data, but also information about any exceptions that were thrown.

You can use this for logging or for timing purposes. What we'll do is just send another email saying Request and response complete.

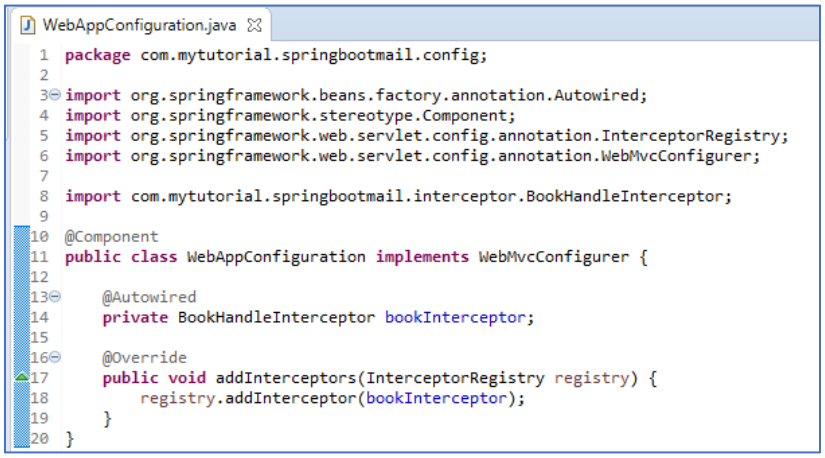
- **sendMail Method**

Let's take a look at the sendEmail function. This is familiar to us, we set up a simple mail message. We'll send email to *[target_email]@outlook.com* .

The subject will talk about book-related activity and we'll specify the bookId as well and the messageText will be whatever message text we've passed in as an input argument plus the current date and time. And, we use **javaMailSender.send** to send the book.

5. Prepare Web Application Configuration Class

Before this interceptor becomes part of your Spring application, it has to be explicitly registered as such. And, I'll do this within the **WebAppConfiguration.java** file.



```
1  package com.mytutorial.springbootmail.config;
2
3  import org.springframework.beans.factory.annotation.Autowired;
4  import org.springframework.stereotype.Component;
5  import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
6  import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
7
8  import com.mytutorial.springbootmail.interceptor.BookHandleInterceptor;
9
10 @Component
11 public class WebAppConfiguration implements WebMvcConfigurer {
12
13     @Autowired
14     private BookHandleInterceptor bookInterceptor;
15
16     @Override
17     public void addInterceptors(InterceptorRegistry registry) {
18         registry.addInterceptor(bookInterceptor);
19     }
20 }
```

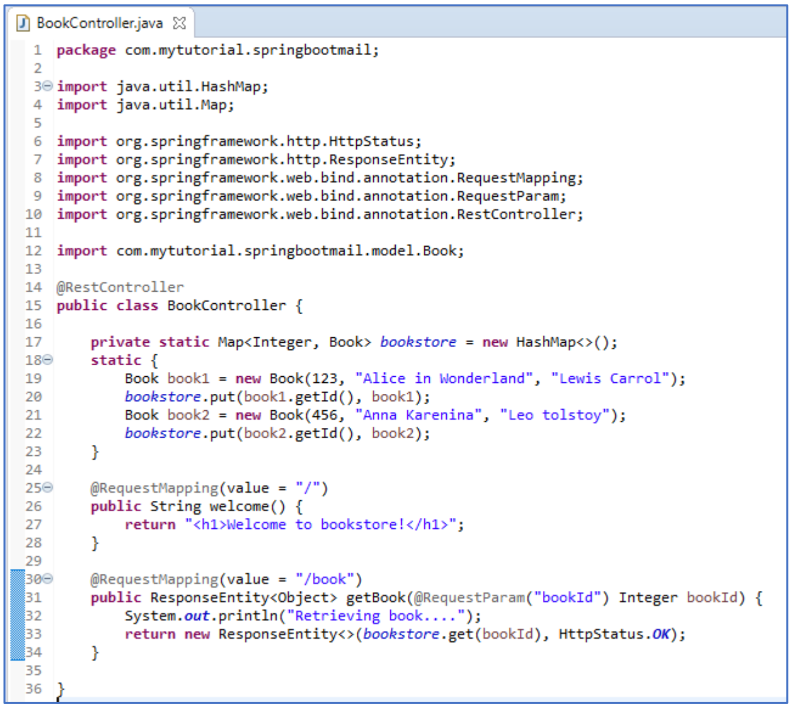
The WebAppConfiguration implements the **WebMvcConfigurer**, indicating that some external configuration properties are present in here.

We tag this class as **@Component** indicating it's managed by Spring. We inject the bookInterceptor here using **@Autowired**, and then we implement the **addInterceptors** method. Within addInterceptors you have access to the InterceptorRegistry which is where you register all of the interceptors that you have in your code.

You can of course configure more than one. We have exactly one, the bookInterceptor. I use **registry.addInterceptor** on line 18 to add this interceptor in.

6. Create Controller Class

Now, we just have one last file to look at, the **BookController.java**.



```java
1  package com.mytutorial.springbootmail;
2
3  import java.util.HashMap;
4  import java.util.Map;
5
6  import org.springframework.http.HttpStatus;
7  import org.springframework.http.ResponseEntity;
8  import org.springframework.web.bind.annotation.RequestMapping;
9  import org.springframework.web.bind.annotation.RequestParam;
10 import org.springframework.web.bind.annotation.RestController;
11
12 import com.mytutorial.springbootmail.model.Book;
13
14 @RestController
15 public class BookController {
16
17     private static Map<Integer, Book> bookstore = new HashMap<>();
18     static {
19         Book book1 = new Book(123, "Alice in Wonderland", "Lewis Carrol");
20         bookstore.put(book1.getId(), book1);
21         Book book2 = new Book(456, "Anna Karenina", "Leo tolstoy");
22         bookstore.put(book2.getId(), book2);
23     }
24
25     @RequestMapping(value = "/")
26     public String welcome() {
27         return "<h1>Welcome to bookstore!</h1>";
28     }
29
30     @RequestMapping(value = "/book")
31     public ResponseEntity<Object> getBook(@RequestParam("bookId") Integer bookId) {
32         System.out.println("Retrieving book....");
33         return new ResponseEntity<>(bookstore.get(bookId), HttpStatus.OK);
34     }
35
36 }
```

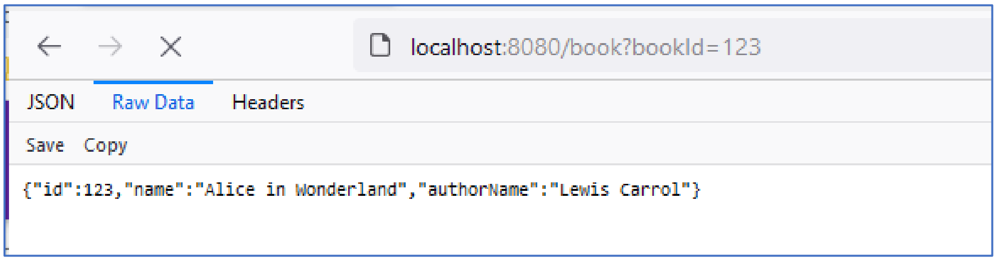
We use the **@RestController** annotation on line 14 indicating that all the return values from our handler methods are web responses to be directly rendered in the browser.

In order to keep things simple, I haven't really configured a database from where we'll access data. I have set-up a static in-memory map instead. We have a map of Integer to Book. This is our bookstore and we have a static initialization block here, which instantiates two book objects, Alice in Wonderland and Anna Karenina and adds them to our bookstore map.

Now, within this controller, I've set-up exactly two handler mappings. One is to the `"/"` path, which simply says, *Welcome. Welcome to the bookstore!*

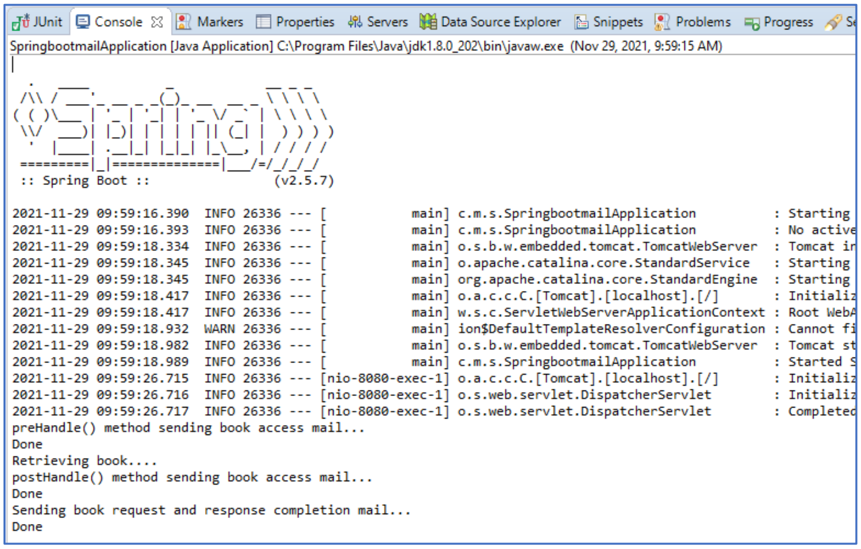
This is not the interesting mapping. The second one is to the path `"/book"`. This method getBook expects as an input argument the bookId, which is injected using **@RequestParam**. We'll print out a Retrieving book and we'll return a **new ResponseEntity**, which simply accesses the book from our bookstore map, and sends back **HTTPStatus, OK**. The book information in our map will be automatically rendered as JSON.

Let's run this code and see how this works and how our interceptors work. I'm going to head over to localhost 8080 and specify the bookId 123. This retrieves the book Alice in Wonderland with id, 123.



```
{"id":123,"name":"Alice in Wonderland","authorName":"Lewis Carrol"}
```

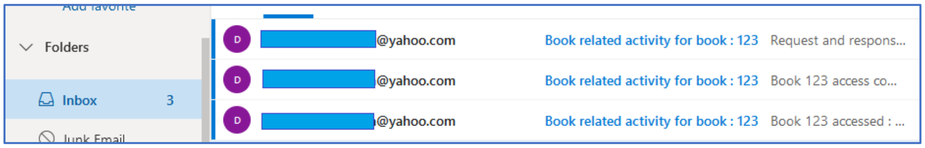
But what I'm really interested in is whether our interceptors were executed. If you take a look at the console logs, you can see that first the preHandle method was called, then the postHandle, and then, finally, the afterCompletion method.



SpringbootmailApplication [Java Application] C:\Program Files\Java\jdk1.8.0_202\bin\javaw.exe (Nov 29, 2021, 9:59:15 AM)

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.5.7)

2021-11-29 09:59:16.390  INFO 26336 --- [           main] c.m.s.SpringbootmailApplication          : Starting
2021-11-29 09:59:16.393  INFO 26336 --- [           main] c.m.s.SpringbootmailApplication          : No active
2021-11-29 09:59:18.334  INFO 26336 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat in
2021-11-29 09:59:18.345  INFO 26336 --- [           main] o.apache.catalina.core.StandardService   : Starting
2021-11-29 09:59:18.345  INFO 26336 --- [           main] org.apache.catalina.core.StandardEngine  : Starting
2021-11-29 09:59:18.417  INFO 26336 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializ
2021-11-29 09:59:18.417  INFO 26336 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebA
2021-11-29 09:59:18.932  WARN 26336 --- [           main] ion$DefaultTemplateResolverConfiguration : Cannot fi
2021-11-29 09:59:18.982  INFO 26336 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat st
2021-11-29 09:59:18.989  INFO 26336 --- [           main] c.m.s.SpringbootmailApplication          : Started S
2021-11-29 09:59:26.715  INFO 26336 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializ
2021-11-29 09:59:26.716  INFO 26336 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Initializ
2021-11-29 09:59:26.717  INFO 26336 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Completed
preHandle() method sending book access mail...
Done
Retrieving book....
postHandle() method sending book access mail...
Done
Sending book request and response completion mail...
Done
```

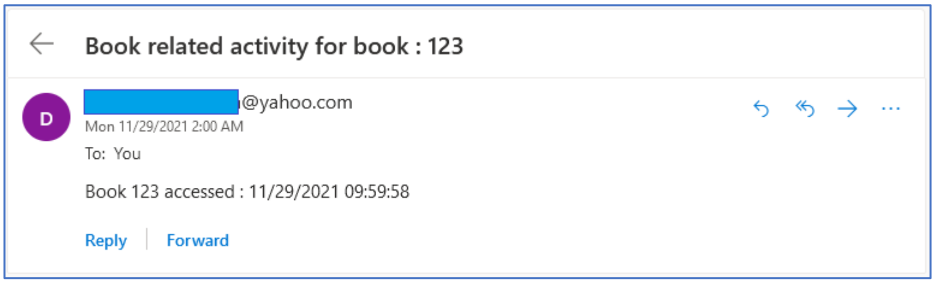
Let's head over to *[target email]@outlook.com* and take a look at the emails sent. There are three emails from alice for book-related activity. The first is Book accessed, then Book access complete, and then Request and response complete. You can see that preHandle was called first, then postPandle, and then afterCompletion.



[redacted]@yahoo.com Book related activity for book : 123 Request and respons...

[redacted]@yahoo.com Book related activity for book : 123 Book 123 access co...

[redacted]@yahoo.com Book related activity for book : 123 Book 123 accessed : ...



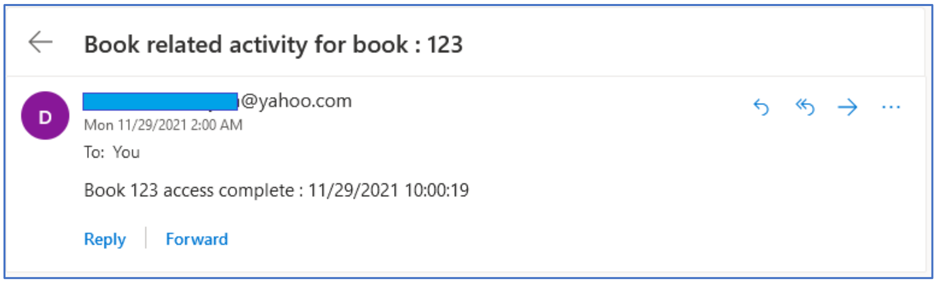
Book related activity for book : 123

[redacted]@yahoo.com

Mon 11/29/2021 2:00 AM

To: You

Book 123 accessed : 11/29/2021 09:59:58



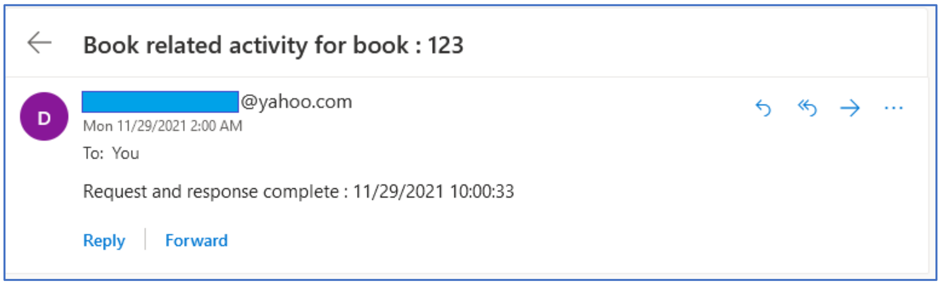
Book related activity for book : 123

D [redacted]@yahoo.com

Mon 11/29/2021 2:00 AM

To: You

Book 123 access complete : 11/29/2021 10:00:19



Book related activity for book : 123

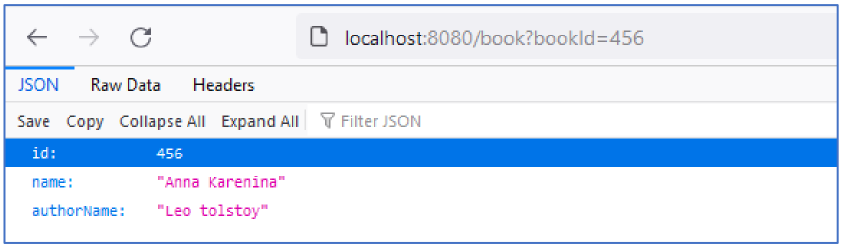
D [redacted]@yahoo.com

Mon 11/29/2021 2:00 AM

To: You

Request and response complete : 11/29/2021 10:00:33

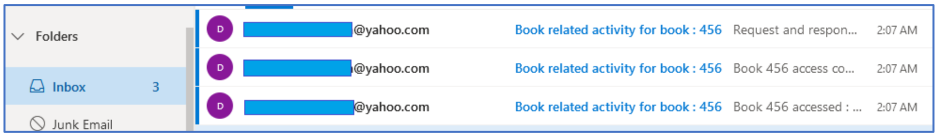
We can try this once again for the other book we have in our bookstore, the one with ID, 456. This is Anna Karenina.



localhost:8080/book?bookId=456

```
id:         456
name:       "Anna Karenina"
authorName: "Leo tolstoy"
```

Let's head over to *[target email]@outlook.com* to see whether we have received our emails. And you can see there are three emails from alice for book with id, 456.



D [redacted]@yahoo.com Book related activity for book : 456 Request and respon... 2:07 AM

D [redacted]@yahoo.com Book related activity for book : 456 Book 456 access co... 2:07 AM

D [redacted]@yahoo.com Book related activity for book : 456 Book 456 accessed : ... 2:07 AM

And if you open up these emails, you'll see the order in which the emails were sent. The preHandle was executed first, then the postHandle, and the afterCompletion at the very end.

Setting up the Zuul Edge Service

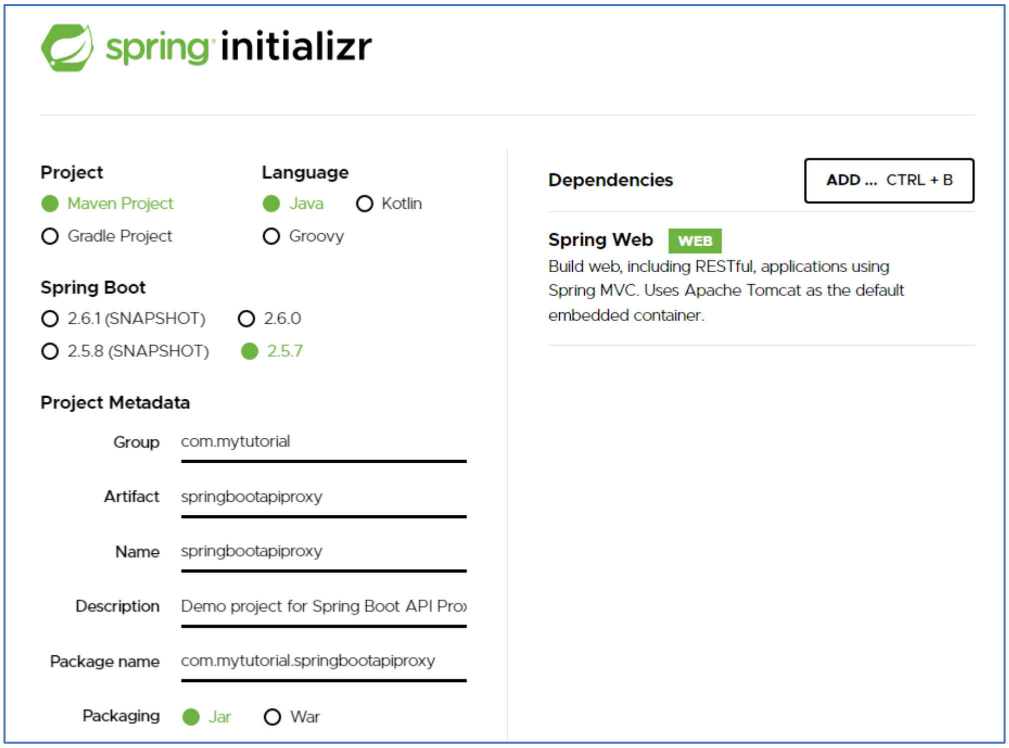
Within your organization, it's possible that you're working on an enterprise-grade web application that provides a RESTful API for its services.

Now, in that case, you might want some features which are beyond basic interceptors that we saw in the previous demo. You might want to set-up an **API proxy** for your services. A good question to ask right now would be: What exactly is an API proxy? *An API proxy is a service that sits in front of your APIs.* Rather than have your users consume your APIs directly, you'll have them send their request to this API proxy. This API proxy will thus control access to your APIs. We can use the API proxy then to enhance the security that we provide to our APIs. We can also use it for monetization by tracking how many times API requests are made by specific users. General access tracking is possible as well.

So, using an API proxy for your APIs is quite common in the real world. And, we'll see how we can do this in **Spring Boot using Zuul's routing and filtering**.

1. Prepare Spring Boot Maven Project

First, we'll set-up a very simple REST API service, which is essentially our actual API, which will live behind a proxy. The only artifact we need is the **spring-boot-starter-web** for the service.



Project
● Maven Project
○ Gradle Project

Language
● Java ○ Kotlin
○ Groovy

Spring Boot
○ 2.6.1 (SNAPSHOT) ○ 2.6.0
○ 2.5.8 (SNAPSHOT) ● 2.5.7

Project Metadata

Group com.mytutorial

Artifact springbootapiproxy

Name springbootapiproxy

Description Demo project for Spring Boot API Prox

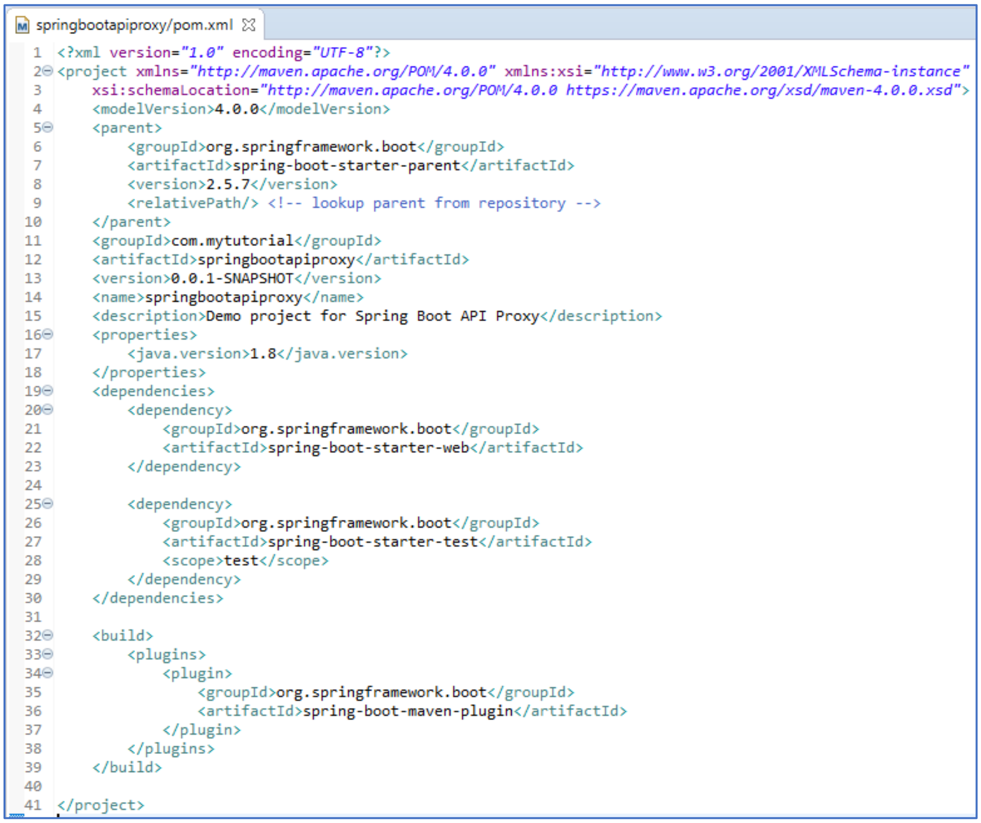
Package name com.mytutorial.springbootapiproxy

Packaging ● Jar ○ War

Dependencies

Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

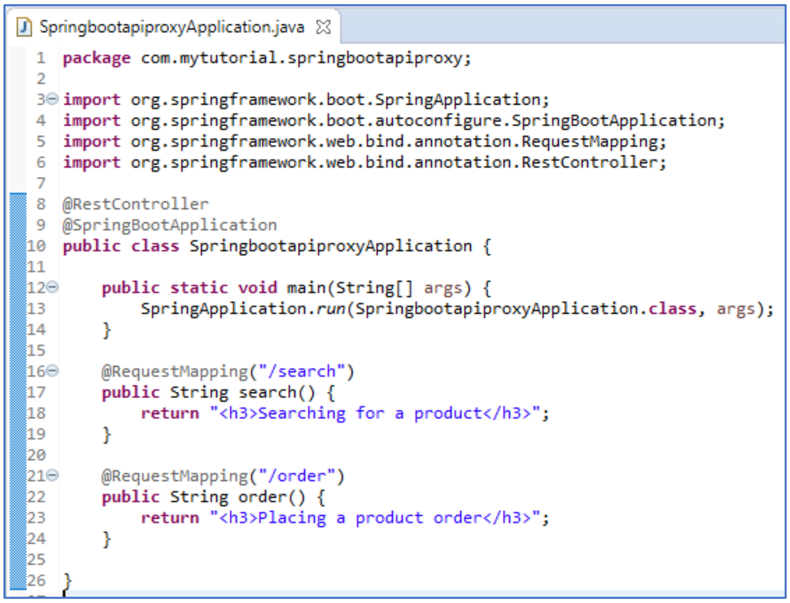
Generated **pom.xml**



```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.5.7</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.mytutorial</groupId>
    <artifactId>springbootapiproxy</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>springbootapiproxy</name>
    <description>Demo project for Spring Boot API Proxy</description>
    <properties>
        <java.version>1.8</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

2. Update the Entry Point Spring Boot Application

I'm going to keep this REST API as simple as possible. I have a single class here, the SpringbootApplication.java file. I've tagged it with the annotations **@SpringBootApplication** and **@RestController**. This is the main entry point of our application as you can see from the main method, where we call **SpringApplication.run**. This is also the controller where we define our handler mappings.



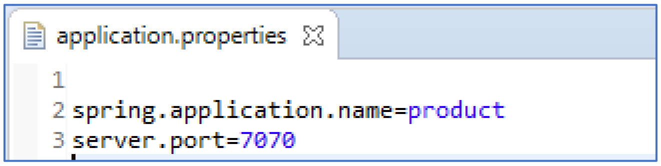
```java
package com.mytutorial.springbootapiproxy;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@SpringBootApplication
public class SpringbootapiproxyApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringbootapiproxyApplication.class, args);
    }

    @RequestMapping("/search")
    public String search() {
        return "<h3>Searching for a product</h3>";
    }

    @RequestMapping("/order")
    public String order() {
        return "<h3>Placing a product order</h3>";
    }

}
```

I have two methods mapped here. The first one is to the path `"/search"`. This method simply returns an h3 header, Searching for a product. I have another handler mapped to `"/order"`, and this returns an h3 header which says Placing a product order.

We have two methods responding to get requests in this API, which we will place behind an API proxy.

3. Configure Spring Boot Environment Properties

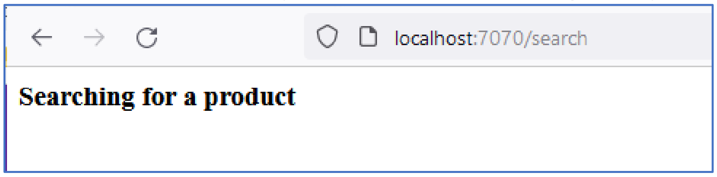
Now, if you're going to have an API proxy and an application run, we'll need to have them run on different ports. I'll configure the port on which my main application runs using **application.properties**.



```

spring.application.name=product
server.port=7070
```

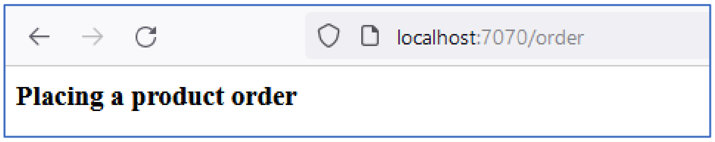
I've set the **server.port** property to *7070*, so that our main application runs on this port. I also set a name for this application just *product*.

First, let's run and test our standalone application and make sure it works. Hit localhost:7070/search, you should see Searching for a product.



Searching for a product

Let's change the URL we hit, localhost:7070/order, and you should see the message, Placing a product order.

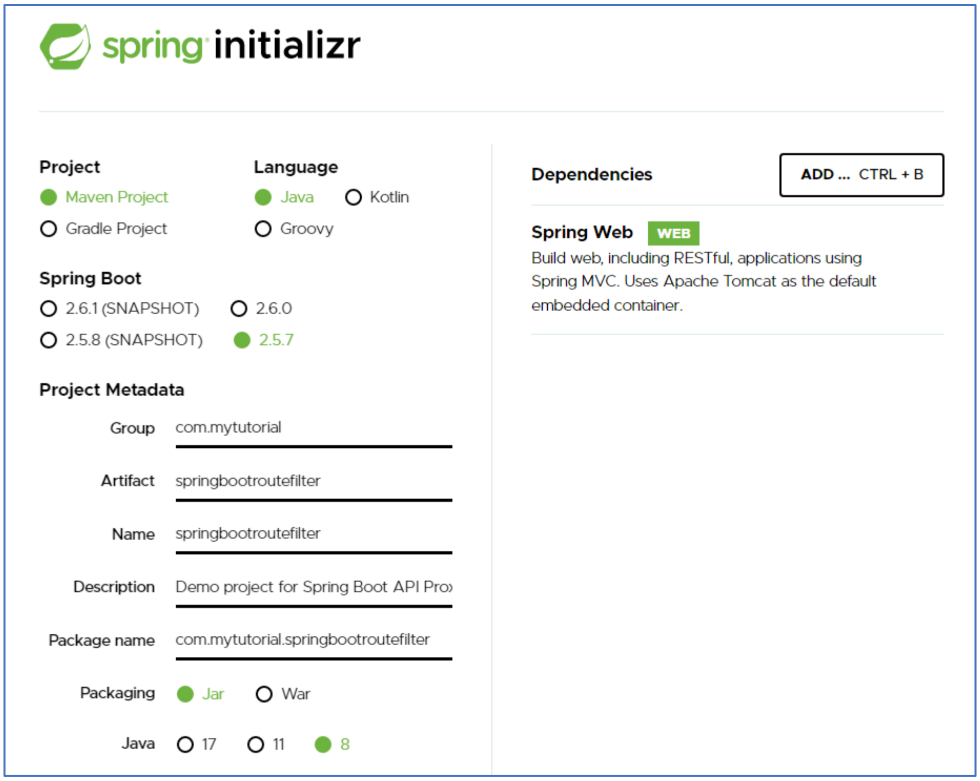


Placing a product order

Our standalone application works just fine.

4. Prepare 2nd Maven Project for API Proxy Service (Zuul Route & Filter)

So now, let's set up our second application that will serve as our API proxy. This will be a separate Spring Boot project. Remember, the API proxy is a completely different service that is in front of your real API. I'm going to use Spring Initializr, it's going to be a Maven Project written in Java, we'll use spring 2.3.1. I'll configure the Project Metadata first before I add-in the dependencies. The Group is com.skillsoft, the Artifact is routingandfiltering. Because this will be the API proxy that will route and filter the request made to our actual API. I'll add-in the dependencies for this routing and filtering operation. We'll of course need Spring Web because this API proxy needs to be able to respond to web request. The second dependency that we'll specify in order to allow our app to perform routing and filtering as API proxies is the **Zuul** dependency. ***Zuul** is a gateway service originally developed by Netflix*.



Project
● Maven Project
○ Gradle Project

Language
● Java ○ Kotlin
○ Groovy

Spring Boot
○ 2.6.1 (SNAPSHOT) ○ 2.6.0
○ 2.5.8 (SNAPSHOT) ● 2.5.7

Project Metadata

Group: com.mytutorial
Artifact: springbootroutefilter
Name: springbootroutefilter
Description: Demo project for Spring Boot API Pro›
Package name: com.mytutorial.springbootroutefilter
Packaging: ● Jar ○ War
Java: ○ 17 ○ 11 ● 8

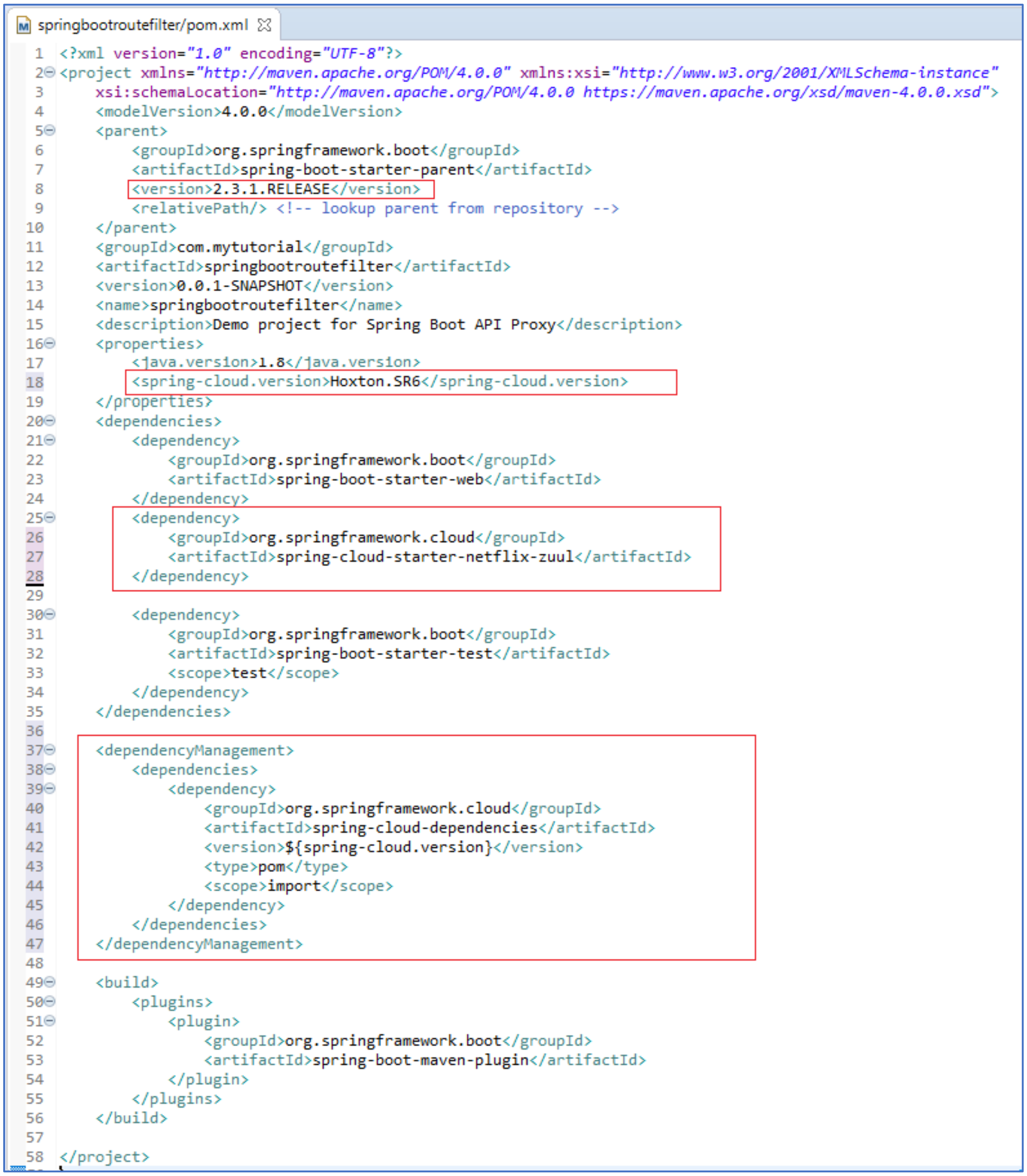
Dependencies

Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Zuul is described as an edge service application and is built to enable dynamic routing, monitoring, resiliency and security for your APIs. Zuul is a JVM-based router and server-side a load balancer. And Spring Cloud has a nice integration with the embedded Zuul proxy, and that's exactly what we'll use here.

Once you've added the Zuul dependency, all that's left to do in order to set-up this project is click on the GENERATE button. This will download a zip file onto our local machine. Unzip this file, this will contain the basic structure of our Maven project. Import this Maven project into our Eclipse IDE.

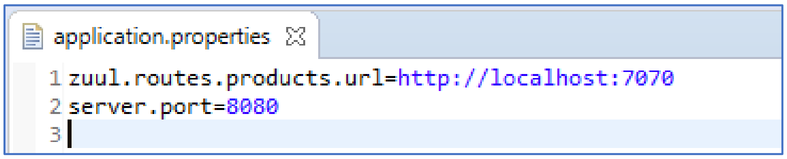
You can see here on the left Project Explorer pane, I have two projects setup, the springboot project that we looked at previously and a new *routingandfiltering* project. Let's take a look at the **pom.xml** of this *routingandfiltering* project to make sure that our dependencies have been set-up correctly. This project depends on the **spring-boot-starter-web** and the **spring-cloud-starter-netflix-zuul**.



```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.3.1.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.mytutorial</groupId>
    <artifactId>springbootroutefilter</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>springbootroutefilter</name>
    <description>Demo project for Spring Boot API Proxy</description>
    <properties>
        <java.version>1.8</java.version>
        <spring-cloud.version>Hoxton.SR6</spring-cloud.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-zuul</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>${spring-cloud.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

5. Setup Configuration Properties for The 2nd Application (Zuul Route & Filter)

Things look good here. Let's take a look at the application properties, which is where we set up our routing configuration.



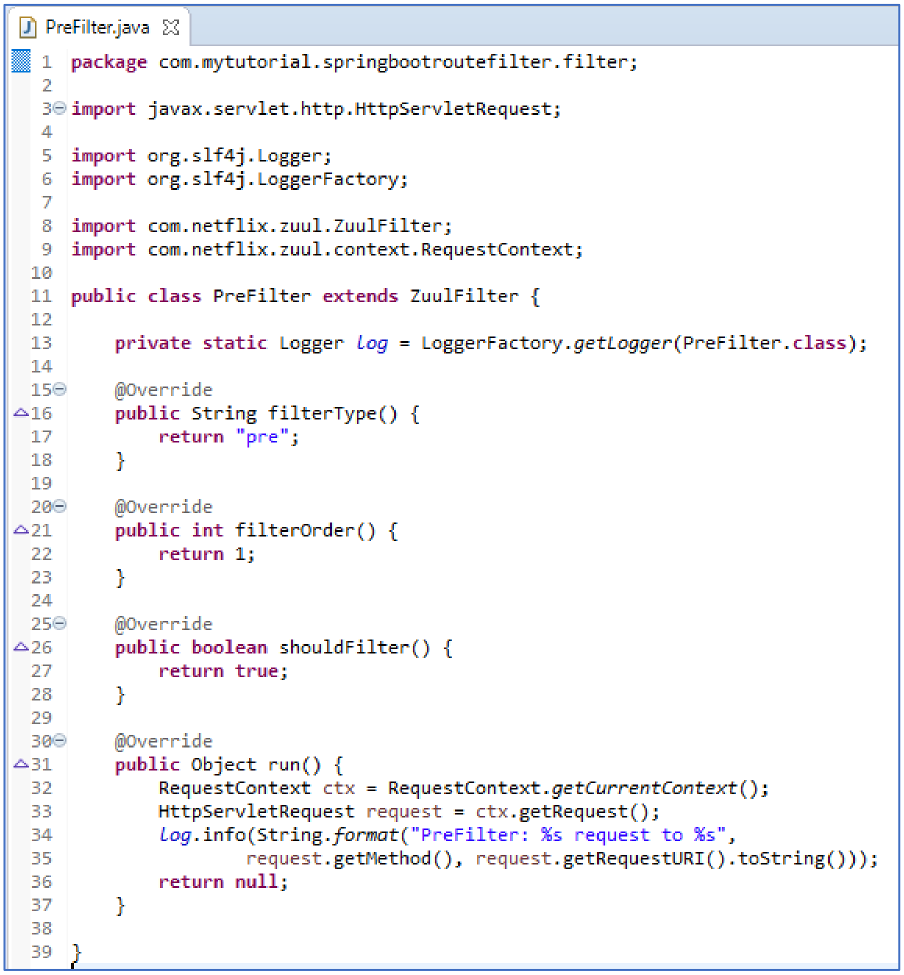
```
zuul.routes.products.url=http://localhost:7070
server.port=8080

```

The first detail to notice here is that we have set `zuul.routes.products.url=http://localhost:7070`, which is where our API service will be running. This basically tells our Zuul edge service application that any requests to the **/products** path in this application should be routed to localhost 7070, that is, our API service. This is the routing portion of our edge service where the actual request made to the API will be satisfied by the API service and not by this routing service.

In addition, we've also explicitly configured that our routing application will be on port 8080. Remember, the port of the routing application and our actual API needs to be different for it to run on the same machine.

6. Create Zuul's Pre Filter Class

In addition to routing, you can also filter out the requests that are made to your APIs using different kinds of filters. Let's see the four filters that can be configured using Zuul. The first is the PreFilter.



```
 1 package com.mytutorial.springbootroutefilter.filter;
 2 
 3 import javax.servlet.http.HttpServletRequest;
 4 
 5 import org.slf4j.Logger;
 6 import org.slf4j.LoggerFactory;
 7 
 8 import com.netflix.zuul.ZuulFilter;
 9 import com.netflix.zuul.context.RequestContext;
10 
11 public class PreFilter extends ZuulFilter {
12 
13     private static Logger log = LoggerFactory.getLogger(PreFilter.class);
14 
15     @Override
16     public String filterType() {
17         return "pre";
18     }
19 
20     @Override
21     public int filterOrder() {
22         return 1;
23     }
24 
25     @Override
26     public boolean shouldFilter() {
27         return true;
28     }
29 
30     @Override
31     public Object run() {
32         RequestContext ctx = RequestContext.getCurrentContext();
33         HttpServletRequest request = ctx.getRequest();
34         log.info(String.format("PreFilter: %s request to %s",
35                 request.getMethod(), request.getRequestURI().toString()));
36         return null;
37     }
38 
39 }
```

The PreFilter is run before the request is actually routed to your service. All of your filters have to **extend** the base class, **ZuulFilter**. As you can see, the PreFilter does here on line 11.

```
        public class PreFilter extends ZuulFilter {
```

How do we know that this filter is a PreFilter? Well, that's the overridden filterType method. Notice that the filterType returns the string, pre.

```
        @Override
        public String filterType() {
                return "pre";
        }
```

This indicates that this is a PreFilter executed before the request is routed to your service.
If you have multiple PreFilters, you might want to specify a certain ordering for your filters. And that you do by overriding the filterOrder method.

```
        @Override
        public int filterOrder() {
```

```
        return 1;
    }
```

Based on the incoming request, you may or may not want to apply this filter, you'll specify this as the return value for the shouldFilter overridden method. We return true, indicating that filtering is always enabled.

```
    @Override
    public boolean shouldFilter() {
        return true;
    }
```

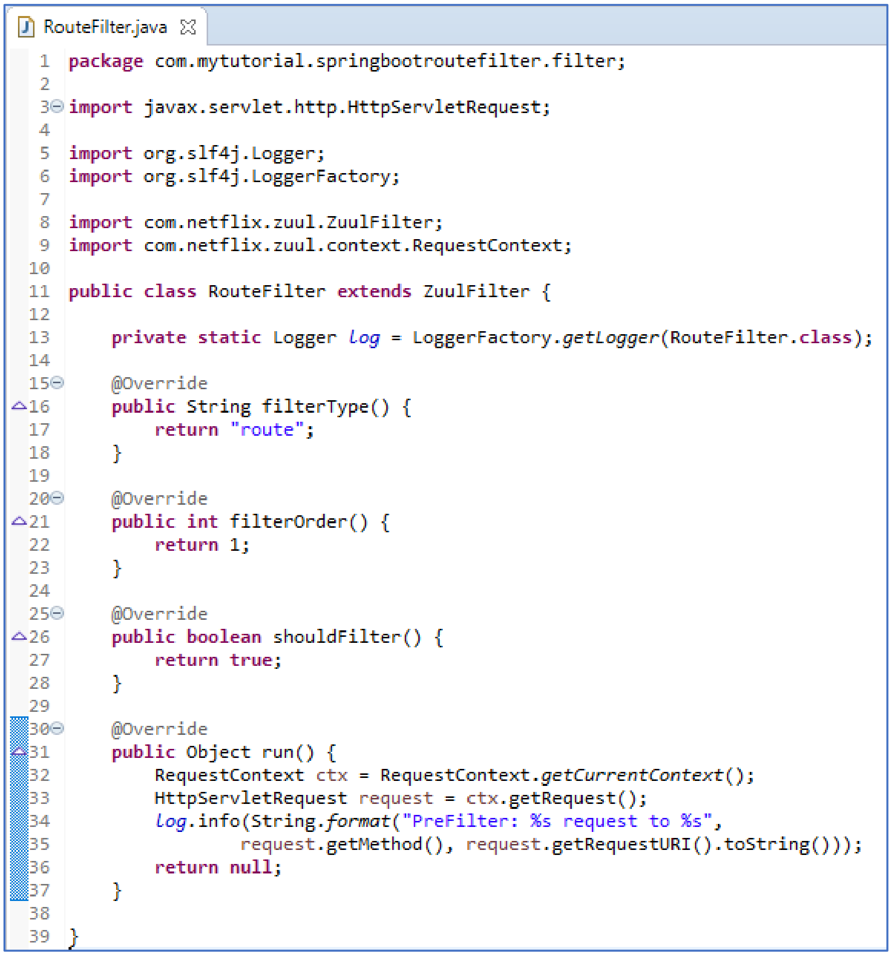
The actual filtering operation is performed by the overridden run method which throws a ZuulException. The current HTTP request can be accessed by other RequestContext. Now our PreFilter doesn't really do anything except log some info out to screen.

```
    @Override
    public Object run() {
        RequestContext ctx = RequestContext.getCurrentContext();
        HttpServletRequest request = ctx.getRequest();
        log.info(
            String.format("PreFilter: %s request to %s",
            request.getMethod(),
            request.getRequestURI().toString()));
        return null;
    }
```

But it's basically up to you how you want to PreFilter your request. Maybe you're tracking the number of request made by a certain client for the purposes of monetization. Maybe you'll check the origin of the request for the purposes of security. Anything is possible here within this PreFilter run code.

7. Create Zuul' Route Filter Class

Let's now look at some of the other types of filters that you can configure. For example, the RouteFilter handles the actual routing of the request. Once again, the *RouteFilter* **extends** the **ZuulFilter** base class.



RouteFilter.java

```
package com.mytutorial.springbootroutefilter.filter;

import javax.servlet.http.HttpServletRequest;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.netflix.zuul.ZuulFilter;
import com.netflix.zuul.context.RequestContext;

public class RouteFilter extends ZuulFilter {

    private static Logger log = LoggerFactory.getLogger(RouteFilter.class);

    @Override
    public String filterType() {
        return "route";
    }

    @Override
    public int filterOrder() {
        return 1;
    }

    @Override
    public boolean shouldFilter() {
        return true;
    }

    @Override
    public Object run() {
        RequestContext ctx = RequestContext.getCurrentContext();
        HttpServletRequest request = ctx.getRequest();
        log.info(String.format("PreFilter: %s request to %s",
                request.getMethod(), request.getRequestURI().toString()));
        return null;
    }

}
```

The filter type that you'll specify is route.

```
@Override
public String filterType() {
        return "route";
}
```

You can specify a filter order, that's optional.

```
@Override
public int filterOrder() {
        return 1;
}
```

You can also specify whether this filter should actually run or not in the shouldFilter method.

```
@Override
public boolean shouldFilter() {
        return true;
}
```

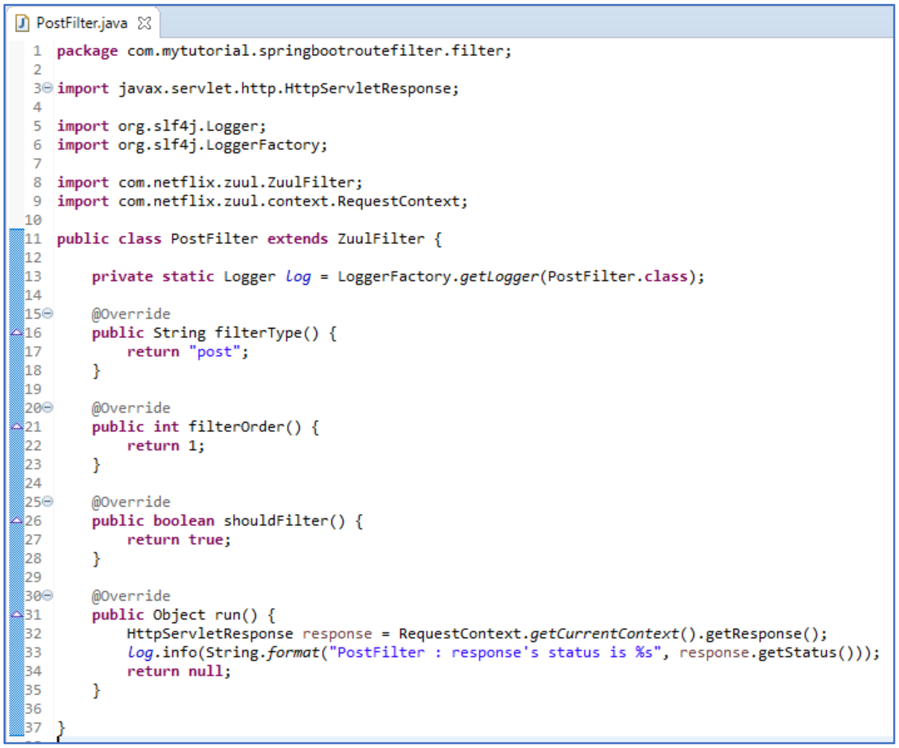
You'll override the run method to specify the code that you want executed when the request is actually routed.

```
@Override
public Object run() {
        RequestContext ctx = RequestContext.getCurrentContext();
        HttpServletRequest request = ctx.getRequest();
        log.info(String.format("PreFilter: %s request to %s",
                request.getMethod(),
                request.getRequestURI().toString()));
        return null;
}
```

So the RouteFilter is processed after the PreFilter. We've simply logged out information the RouteFilter out to screen.

8. Create Zuul' Post Filter Class

Another filter that we have set up here is the PostFilter. This is where the filterType returns post. This PostFilter extends the base ZuulFilter class. And this runs after the request has been routed to your API service.

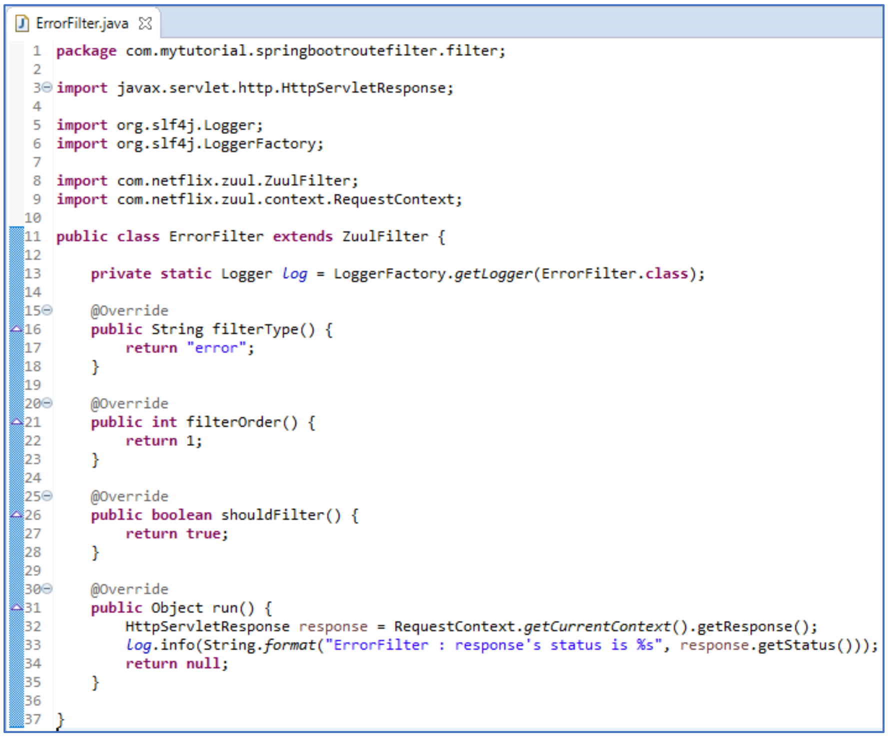


```java
package com.mytutorial.springbootroutefilter.filter;

import javax.servlet.http.HttpServletResponse;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.netflix.zuul.ZuulFilter;
import com.netflix.zuul.context.RequestContext;

public class PostFilter extends ZuulFilter {

    private static Logger log = LoggerFactory.getLogger(PostFilter.class);

    @Override
    public String filterType() {
        return "post";
    }

    @Override
    public int filterOrder() {
        return 1;
    }

    @Override
    public boolean shouldFilter() {
        return true;
    }

    @Override
    public Object run() {
        HttpServletResponse response = RequestContext.getCurrentContext().getResponse();
        log.info(String.format("PostFilter : response's status is %s", response.getStatus()));
        return null;
    }

}
```

The other methods that we've overridden here are all familiar to us. The actual code for this filter is in the run method. Notice that in the *PostFilter*, we have access to the **ServletResponse**. This is the HTTP response from our API service. We can post process this response as we want to.

9. Create Zuul’ Error Filter Class

Let's look at one last filter here, the ErrorFilter. The ErrorFilter extends the ZuulFilter base class.



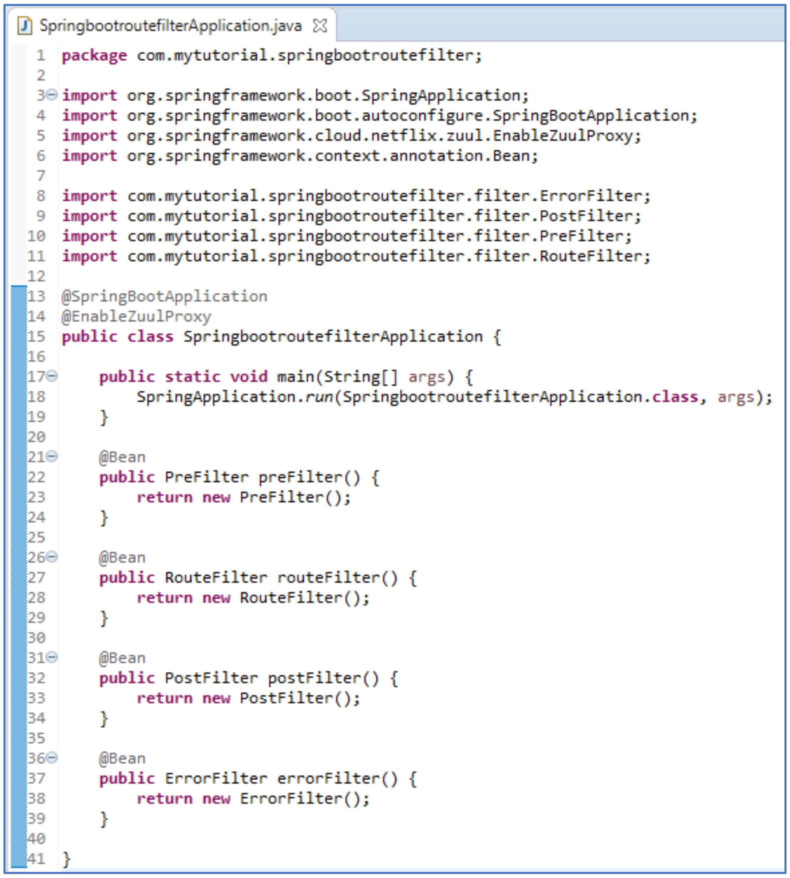
```java
package com.mytutorial.springbootroutefilter.filter;

import javax.servlet.http.HttpServletResponse;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.netflix.zuul.ZuulFilter;
import com.netflix.zuul.context.RequestContext;

public class ErrorFilter extends ZuulFilter {

    private static Logger log = LoggerFactory.getLogger(ErrorFilter.class);

    @Override
    public String filterType() {
        return "error";
    }

    @Override
    public int filterOrder() {
        return 1;
    }

    @Override
    public boolean shouldFilter() {
        return true;
    }

    @Override
    public Object run() {
        HttpServletResponse response = RequestContext.getCurrentContext().getResponse();
        log.info(String.format("ErrorFilter : response's status is %s", response.getStatus()));
        return null;
    }

}
```

The filterType is **error**. The ErrorFilter is not invoked for every request. It only runs if an error occurs in the handling of the request.

All of the overridden methods we are familiar with. The run method is where we access the response, and we can do additional stuff in case of an error.

10. Update Entry Point of Spring Boot Application (Zuul Route & Filter)

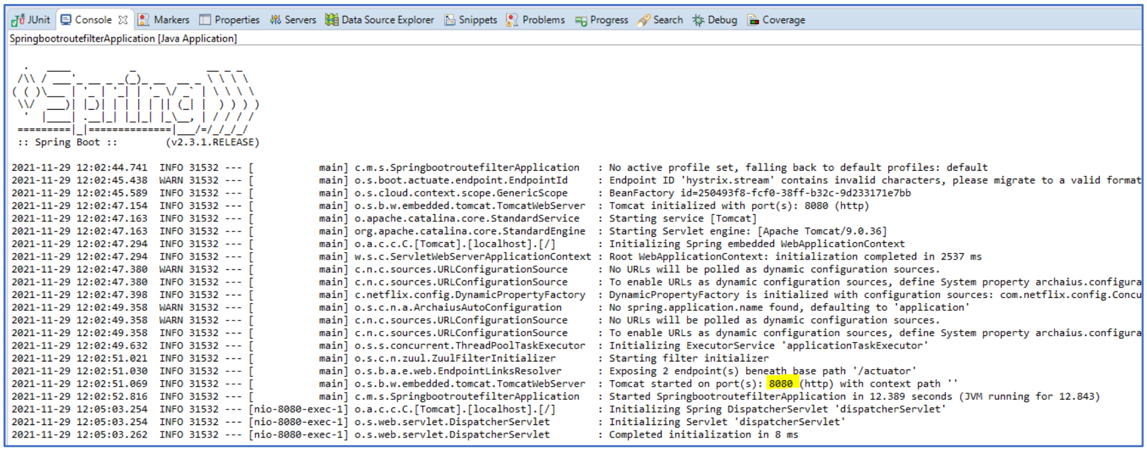
Let's take a look at the main entry point of this edge service application, the **SpringbootroutingfilterApplication.java** class.



```
1  package com.mytutorial.springbootroutefilter;
2
3  import org.springframework.boot.SpringApplication;
4  import org.springframework.boot.autoconfigure.SpringBootApplication;
5  import org.springframework.cloud.netflix.zuul.EnableZuulProxy;
6  import org.springframework.context.annotation.Bean;
7
8  import com.mytutorial.springbootroutefilter.filter.ErrorFilter;
9  import com.mytutorial.springbootroutefilter.filter.PostFilter;
10 import com.mytutorial.springbootroutefilter.filter.PreFilter;
11 import com.mytutorial.springbootroutefilter.filter.RouteFilter;
12
13 @SpringBootApplication
14 @EnableZuulProxy
15 public class SpringbootroutefilterApplication {
16
17     public static void main(String[] args) {
18         SpringApplication.run(SpringbootroutefilterApplication.class, args);
19     }
20
21     @Bean
22     public PreFilter preFilter() {
23         return new PreFilter();
24     }
25
26     @Bean
27     public RouteFilter routeFilter() {
28         return new RouteFilter();
29     }
30
31     @Bean
32     public PostFilter postFilter() {
33         return new PostFilter();
34     }
35
36     @Bean
37     public ErrorFilter errorFilter() {
38         return new ErrorFilter();
39     }
40
41 }
```

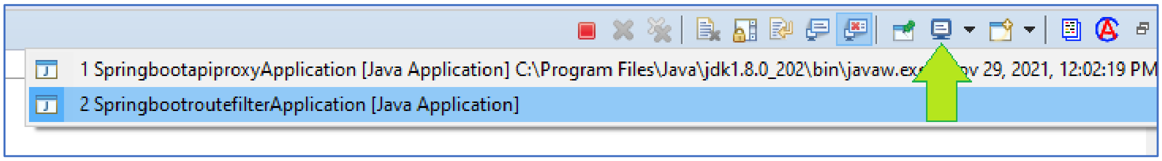
Notice that we have two annotations on lines 13 and 14 for this class. The **@SpringBootApplication** indicating this is the entry point, and the **@EnableZuulProxy** indicating that this is an edge service application for routing and filtering. In order to have your Zuul filters run, you need to explicitly instantiate these as Beans that Spring manages. I have a Bean specification for the **PreFilter**, **RouteFilter**, **PostFilter**, as well as the **ErrorFilter**.

We are now ready to run and test our code, which means we have to run each of our applications here. First, right click on the springboot application that is our API service. Select the app and click OK to run this app. Eclipse allows you to run multiple applications from within the IDE. Right-click on the *routingandfiltering* project, choose Run As and Java Application, and this will run your edge service proxy application as well.

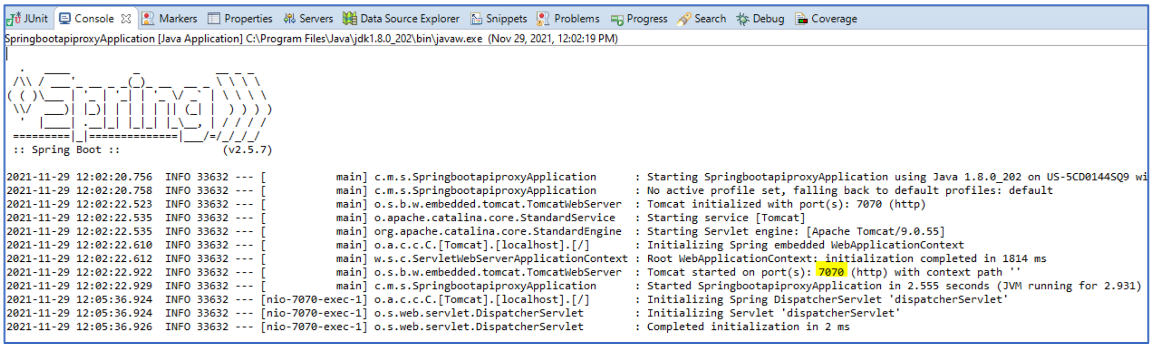


```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.3.1.RELEASE)

2021-11-29 12:02:44.741  INFO 31532 --- [           main] c.m.s.SpringbootroutefilterApplication   : No active profile set, falling back to default profiles: default
2021-11-29 12:02:45.438  WARN 31532 --- [           main] o.s.boot.actuate.endpoint.EndpointId     : Endpoint ID 'hystrix.stream' contains invalid characters, please migrate to a valid format
2021-11-29 12:02:45.589  INFO 31532 --- [           main] o.s.cloud.context.scope.GenericScope     : BeanFactory id=250493f8-fcf0-38ff-b32c-9d233171e7bb
2021-11-29 12:02:47.154  INFO 31532 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2021-11-29 12:02:47.163  INFO 31532 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2021-11-29 12:02:47.163  INFO 31532 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.36]
2021-11-29 12:02:47.294  INFO 31532 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2021-11-29 12:02:47.294  INFO 31532 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 2537 ms
2021-11-29 12:02:47.380  WARN 31532 --- [           main] c.n.c.sources.URLConfigurationSource     : No URLs will be polled as dynamic configuration sources.
2021-11-29 12:02:47.380  INFO 31532 --- [           main] c.n.c.sources.URLConfigurationSource     : To enable URLs as dynamic configuration sources, define System property archaius.configura
2021-11-29 12:02:47.398  INFO 31532 --- [           main] c.netflix.config.DynamicPropertyFactory  : DynamicPropertyFactory is initialized with configuration sources: com.netflix.config.Concu
2021-11-29 12:02:49.358  WARN 31532 --- [           main] o.s.c.n.a.ArchaiusAutoConfiguration      : No spring.application.name found, defaulting to 'application'
2021-11-29 12:02:49.358  WARN 31532 --- [           main] c.n.c.sources.URLConfigurationSource     : No URLs will be polled as dynamic configuration sources.
2021-11-29 12:02:49.358  INFO 31532 --- [           main] c.n.c.sources.URLConfigurationSource     : To enable URLs as dynamic configuration sources, define System property archaius.configura
2021-11-29 12:02:49.632  INFO 31532 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor  : Initializing ExecutorService 'applicationTaskExecutor'
2021-11-29 12:02:51.021  INFO 31532 --- [           main] o.s.c.n.zuul.ZuulFilterInitializer       : Starting filter initializer
2021-11-29 12:02:51.030  INFO 31532 --- [           main] o.s.b.a.e.web.EndpointLinksResolver      : Exposing 2 endpoint(s) beneath base path '/actuator'
2021-11-29 12:02:51.069  INFO 31532 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2021-11-29 12:02:52.816  INFO 31532 --- [           main] c.m.s.SpringbootroutefilterApplication   : Started SpringbootroutefilterApplication in 12.389 seconds (JVM running for 12.843)
2021-11-29 12:05:03.254  INFO 31532 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet 'dispatcherServlet'
2021-11-29 12:05:03.254  INFO 31532 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
2021-11-29 12:05:03.262  INFO 31532 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Completed initialization in 8 ms
```

You can view the console log messages for each of these applications individually by clicking on this little console icon, and selecting the application host console messages you want to view. If you want to view the console messages of the *SpringbootApplication* that is our API service, select that option here.


1 SpringbootapiproxyApplication [Java Application] C:\Program Files\Java\jdk1.8.0_202\bin\javaw.ex   ov 29, 2021, 12:02:19 PM
2 SpringbootroutefilterApplication [Java Application]

And these are the console messages for our original API, you can see that our API is running on port 7070. You can use the same console icon to switch to the RoutingandfilteringApplication. This is our edge service running the Zuul proxy. And if you scroll over to the right, you can see that our edge service is running on port 8080.

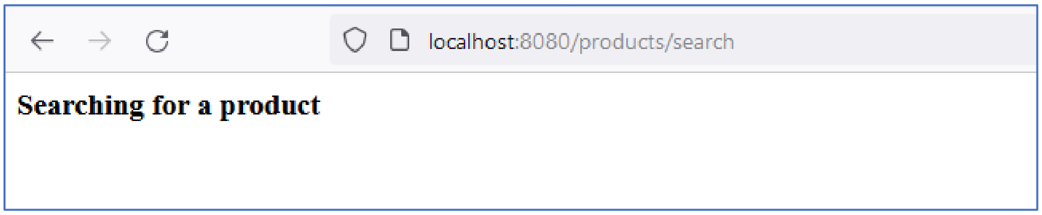

JUnit
Console
Markers
Properties
Servers
Data Source Explorer
Snippets
Problems
Progress
Search
Debug
Coverage
SpringbootapiproxyApplication [Java Application] C:\Program Files\Java\jdk1.8.0_202\bin\javaw.exe (Nov 29, 2021, 12:02:19 PM)

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.5.7)

2021-11-29 12:02:20.756  INFO 33632 --- [           main] c.m.s.SpringbootapiproxyApplication      : Starting SpringbootapiproxyApplication using Java 1.8.0_202 on US-5CD0144SQ9 wi
2021-11-29 12:02:20.758  INFO 33632 --- [           main] c.m.s.SpringbootapiproxyApplication      : No active profile set, falling back to default profiles: default
2021-11-29 12:02:22.523  INFO 33632 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 7070 (http)
2021-11-29 12:02:22.535  INFO 33632 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2021-11-29 12:02:22.535  INFO 33632 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.55]
2021-11-29 12:02:22.610  INFO 33632 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2021-11-29 12:02:22.612  INFO 33632 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1814 ms
2021-11-29 12:02:22.922  INFO 33632 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 7070 (http) with context path ''
2021-11-29 12:02:22.929  INFO 33632 --- [           main] c.m.s.SpringbootapiproxyApplication      : Started SpringbootapiproxyApplication in 2.555 seconds (JVM running for 2.931)
2021-11-29 12:05:36.924  INFO 33632 --- [nio-7070-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet 'dispatcherServlet'
2021-11-29 12:05:36.924  INFO 33632 --- [nio-7070-exec-1] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
2021-11-29 12:05:36.926  INFO 33632 --- [nio-7070-exec-1] o.s.web.servlet.DispatcherServlet        : Completed initialization in 2 ms
```

11. Routing and Filtering Using Zuul

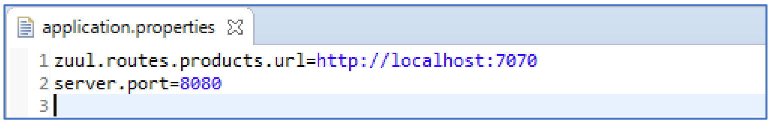
With our edge service as well as our API service up and running, let's see how everything works and let's see how our different filters are applied.

I'm first going to head over to *localhost:8080/products/search*.



Searching for a product

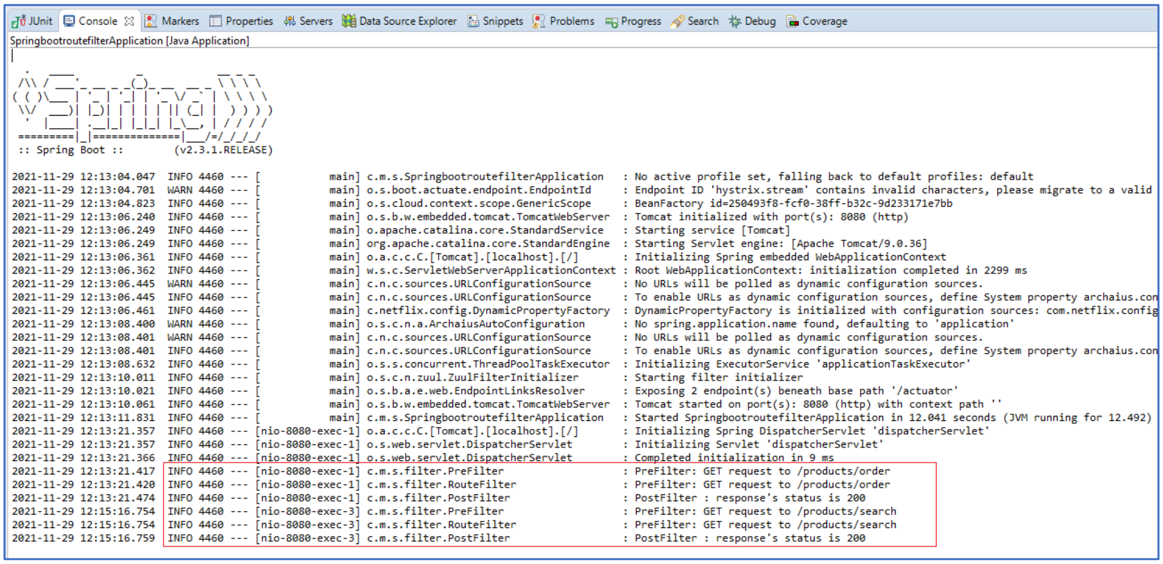
Why do we specify this particular path for our URL? This is because we had configured within our edge service application, that is, the Zuul application that **zuul.routes.products.url** should be routed to localhost 7070.



```
zuul.routes.products.url=http://localhost:7070
server.port=8080

```

The use of the term, products, here basically means that any request to **/products** which is what we have specified in our URL here will be routed to our API service running on *localhost 7070*. Let's see if that's true. Go ahead and hit Enter and you'll see the results from our API, Searching for a product. By hitting the URL of our edge service, we were routed to our original API. This would have resulted in the application of the different filters that we had configured.

Let's switch over to the Console window.

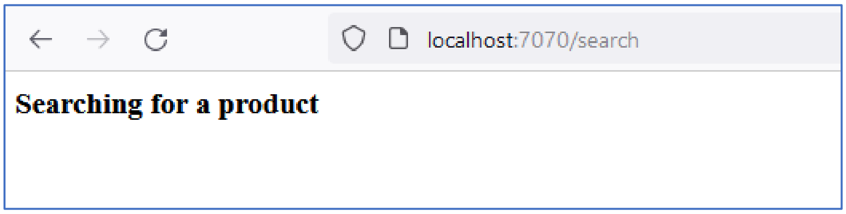


```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.3.1.RELEASE)

2021-11-29 12:13:04.047  INFO 4460 --- [           main] c.m.s.SpringbootroutefilterApplication   : No active profile set, falling back to default profiles: default
2021-11-29 12:13:04.701  WARN 4460 --- [           main] o.s.boot.actuate.endpoint.EndpointId     : Endpoint ID 'hystrix.stream' contains invalid characters, please migrate to a valid
2021-11-29 12:13:04.823  INFO 4460 --- [           main] o.s.cloud.context.scope.GenericScope     : BeanFactory id=250493f8-fcf0-38ff-b32c-9d233171e7bb
2021-11-29 12:13:06.240  INFO 4460 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2021-11-29 12:13:06.249  INFO 4460 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2021-11-29 12:13:06.249  INFO 4460 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.36]
2021-11-29 12:13:06.361  INFO 4460 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2021-11-29 12:13:06.362  INFO 4460 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 2299 ms
2021-11-29 12:13:06.445  WARN 4460 --- [           main] c.n.c.sources.URLConfigurationSource     : No URLs will be polled as dynamic configuration sources.
2021-11-29 12:13:06.445  INFO 4460 --- [           main] c.n.c.sources.URLConfigurationSource     : To enable URLs as dynamic configuration sources, define System property archaius.con
2021-11-29 12:13:06.461  INFO 4460 --- [           main] c.netflix.config.DynamicPropertyFactory  : DynamicPropertyFactory is initialized with configuration sources: com.netflix.config
2021-11-29 12:13:08.400  WARN 4460 --- [           main] o.s.c.n.a.ArchaiusAutoConfiguration      : No spring.application.name found, defaulting to 'application'
2021-11-29 12:13:08.401  WARN 4460 --- [           main] c.n.c.sources.URLConfigurationSource     : No URLs will be polled as dynamic configuration sources.
2021-11-29 12:13:08.401  INFO 4460 --- [           main] c.n.c.sources.URLConfigurationSource     : To enable URLs as dynamic configuration sources, define System property archaius.con
2021-11-29 12:13:08.632  INFO 4460 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor  : Initializing ExecutorService 'applicationTaskExecutor'
2021-11-29 12:13:10.011  INFO 4460 --- [           main] o.s.c.n.zuul.ZuulFilterInitializer       : Starting filter initializer
2021-11-29 12:13:10.021  INFO 4460 --- [           main] o.s.b.a.e.web.EndpointLinksResolver      : Exposing 2 endpoint(s) beneath base path '/actuator'
2021-11-29 12:13:10.061  INFO 4460 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2021-11-29 12:13:11.831  INFO 4460 --- [           main] c.m.s.SpringbootroutefilterApplication   : Started SpringbootroutefilterApplication in 12.041 seconds (JVM running for 12.492)
2021-11-29 12:13:21.357  INFO 4460 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet 'dispatcherServlet'
2021-11-29 12:13:21.357  INFO 4460 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
2021-11-29 12:13:21.366  INFO 4460 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Completed initialization in 9 ms
2021-11-29 12:13:21.417  INFO 4460 --- [nio-8080-exec-1] c.m.s.filter.PreFilter                   : PreFilter: GET request to /products/order
2021-11-29 12:13:21.420  INFO 4460 --- [nio-8080-exec-1] c.m.s.filter.RouteFilter                 : PreFilter: GET request to /products/order
2021-11-29 12:13:21.474  INFO 4460 --- [nio-8080-exec-1] c.m.s.filter.PostFilter                  : PostFilter : response's status is 200
2021-11-29 12:15:16.754  INFO 4460 --- [nio-8080-exec-3] c.m.s.filter.PreFilter                   : PreFilter: GET request to /products/search
2021-11-29 12:15:16.754  INFO 4460 --- [nio-8080-exec-3] c.m.s.filter.RouteFilter                 : PreFilter: GET request to /products/search
2021-11-29 12:15:16.759  INFO 4460 --- [nio-8080-exec-3] c.m.s.filter.PostFilter                  : PostFilter : response's status is 200
```

And in the console messages for this *routingandfiltering* edge service application, you can see that the PreFilter was applied first, then the RouteFilter, and then the PostFilter. We have logging messages from all three of these. The PostFilter tells us that the response status is 200.

Let's hit a slightly different URL, localhost:8080/products/order. This will route to the order web request of our original API where we place a product order. Let's take a look at the console messages. Notice that we made a GET request in the PreFilter to localhost:8080/products/order. Then, we have the RouteFilter run, and then the PostFilter is run. The PostFilter tells us that the status of the response is 200.

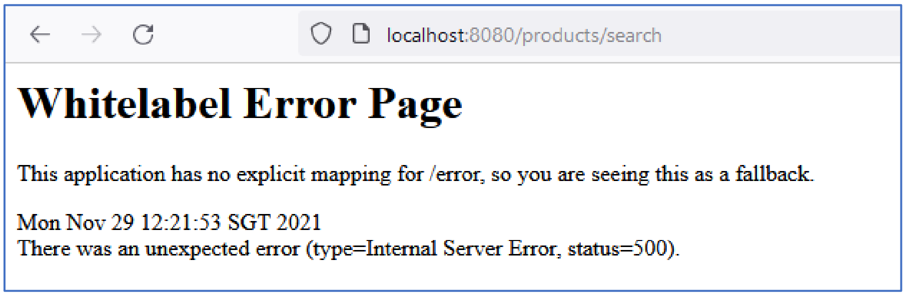
The API service will also be accessible by directly hitting its URL. Here we hit the URL *localhost:7070/search* and get a response. You can also hit the order URL on localhost:7070.



Searching for a product

This will give us a response directly from our API service.

Now, let's switch back to our Eclipse IDE and try out the ErrorFilter. I'm going to use the console icon to switch over to the console contents for the SpringbootApplication, our API service. This application as you can see is currently running. I'm going to use the stop button here within my Console tab in order to stop this API service. Our API service will no longer be up and running and this will cause the ErrorFilter to run. Switch back to RoutingandfilteringApplication. The routing application is still up, our API service is down. Back to our browser and let's hit *localhost:8080/products/search* and this gives us an error page.

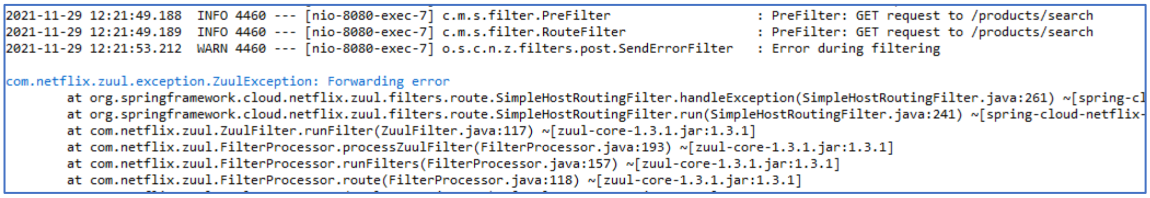


Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

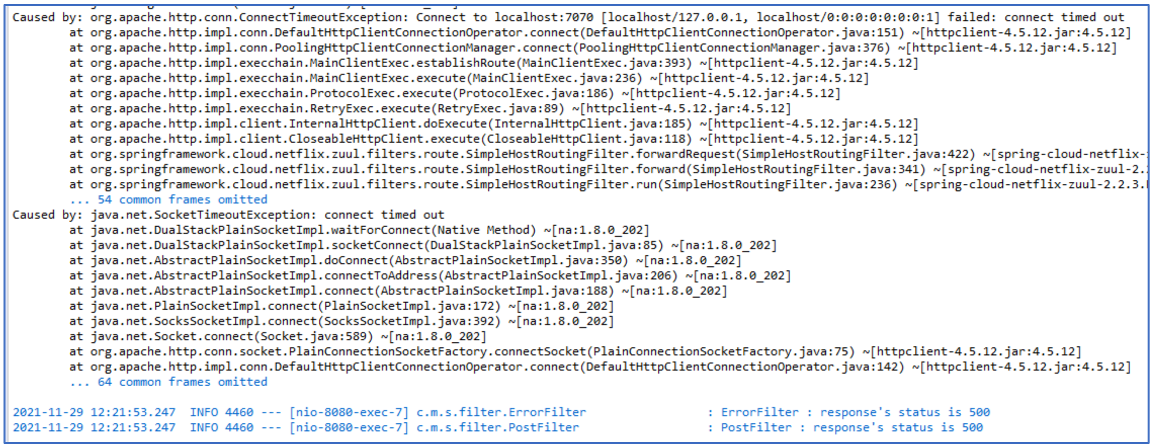
Mon Nov 29 12:21:53 SGT 2021
There was an unexpected error (type=Internal Server Error, status=500).

Clearly a server error with status 500 has occurred. If you switch back to your Eclipse IDE, you'll see something interesting, you can see that the PreFilter and the RouteFilter was run.



```
2021-11-29 12:21:49.188  INFO 4460 --- [nio-8080-exec-7] c.m.s.filter.PreFilter                   : PreFilter: GET request to /products/search
2021-11-29 12:21:49.189  INFO 4460 --- [nio-8080-exec-7] c.m.s.filter.RouteFilter                 : PreFilter: GET request to /products/search
2021-11-29 12:21:53.212  WARN 4460 --- [nio-8080-exec-7] o.s.c.n.z.filters.post.SendErrorFilter   : Error during filtering

com.netflix.zuul.exception.ZuulException: Forwarding error
	at org.springframework.cloud.netflix.zuul.filters.route.SimpleHostRoutingFilter.handleException(SimpleHostRoutingFilter.java:261) ~[spring-cl
	at org.springframework.cloud.netflix.zuul.filters.route.SimpleHostRoutingFilter.run(SimpleHostRoutingFilter.java:241) ~[spring-cloud-netflix-
	at com.netflix.zuul.ZuulFilter.runFilter(ZuulFilter.java:117) ~[zuul-core-1.3.1.jar:1.3.1]
	at com.netflix.zuul.FilterProcessor.processZuulFilter(FilterProcessor.java:193) ~[zuul-core-1.3.1.jar:1.3.1]
	at com.netflix.zuul.FilterProcessor.runFilters(FilterProcessor.java:157) ~[zuul-core-1.3.1.jar:1.3.1]
	at com.netflix.zuul.FilterProcessor.route(FilterProcessor.java:118) ~[zuul-core-1.3.1.jar:1.3.1]
```

After that, we had an ErrorFilter run. A Zuul exception with the message, Forwarding error, was encountered in our app. If you scroll down, you'll see additional details for this error.



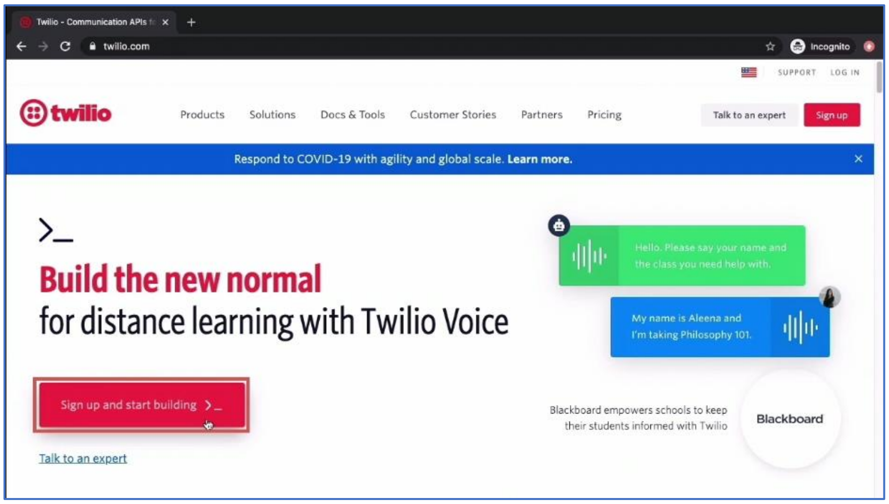
```
Caused by: org.apache.http.conn.ConnectTimeoutException: Connect to localhost:7070 [localhost/127.0.0.1, localhost/0:0:0:0:0:0:0:1] failed: connect timed out
	at org.apache.http.impl.conn.DefaultHttpClientConnectionOperator.connect(DefaultHttpClientConnectionOperator.java:151) ~[httpclient-4.5.12.jar:4.5.12]
	at org.apache.http.impl.conn.PoolingHttpClientConnectionManager.connect(PoolingHttpClientConnectionManager.java:376) ~[httpclient-4.5.12.jar:4.5.12]
	at org.apache.http.impl.execchain.MainClientExec.establishRoute(MainClientExec.java:393) ~[httpclient-4.5.12.jar:4.5.12]
	at org.apache.http.impl.execchain.MainClientExec.execute(MainClientExec.java:236) ~[httpclient-4.5.12.jar:4.5.12]
	at org.apache.http.impl.execchain.ProtocolExec.execute(ProtocolExec.java:186) ~[httpclient-4.5.12.jar:4.5.12]
	at org.apache.http.impl.execchain.RetryExec.execute(RetryExec.java:89) ~[httpclient-4.5.12.jar:4.5.12]
	at org.apache.http.impl.client.InternalHttpClient.doExecute(InternalHttpClient.java:185) ~[httpclient-4.5.12.jar:4.5.12]
	at org.apache.http.impl.client.CloseableHttpClient.execute(CloseableHttpClient.java:118) ~[httpclient-4.5.12.jar:4.5.12]
	at org.springframework.cloud.netflix.zuul.filters.route.SimpleHostRoutingFilter.forwardRequest(SimpleHostRoutingFilter.java:422) ~[spring-cloud-netflix-
	at org.springframework.cloud.netflix.zuul.filters.route.SimpleHostRoutingFilter.forward(SimpleHostRoutingFilter.java:341) ~[spring-cloud-netflix-zuul-2.
	at org.springframework.cloud.netflix.zuul.filters.route.SimpleHostRoutingFilter.run(SimpleHostRoutingFilter.java:236) ~[spring-cloud-netflix-zuul-2.2.3.
	... 54 common frames omitted
Caused by: java.net.SocketTimeoutException: connect timed out
	at java.net.DualStackPlainSocketImpl.waitForConnect(Native Method) ~[na:1.8.0_202]
	at java.net.DualStackPlainSocketImpl.socketConnect(DualStackPlainSocketImpl.java:85) ~[na:1.8.0_202]
	at java.net.AbstractPlainSocketImpl.doConnect(AbstractPlainSocketImpl.java:350) ~[na:1.8.0_202]
	at java.net.AbstractPlainSocketImpl.connectToAddress(AbstractPlainSocketImpl.java:206) ~[na:1.8.0_202]
	at java.net.AbstractPlainSocketImpl.connect(AbstractPlainSocketImpl.java:188) ~[na:1.8.0_202]
	at java.net.PlainSocketImpl.connect(PlainSocketImpl.java:172) ~[na:1.8.0_202]
	at java.net.SocksSocketImpl.connect(SocksSocketImpl.java:392) ~[na:1.8.0_202]
	at java.net.Socket.connect(Socket.java:589) ~[na:1.8.0_202]
	at org.apache.http.conn.socket.PlainConnectionSocketFactory.connectSocket(PlainConnectionSocketFactory.java:75) ~[httpclient-4.5.12.jar:4.5.12]
	at org.apache.http.impl.conn.DefaultHttpClientConnectionOperator.connect(DefaultHttpClientConnectionOperator.java:142) ~[httpclient-4.5.12.jar:4.5.12]
	... 64 common frames omitted

2021-11-29 12:21:53.247  INFO 4460 --- [nio-8080-exec-7] c.m.s.filter.ErrorFilter                 : ErrorFilter : response's status is 500
2021-11-29 12:21:53.247  INFO 4460 --- [nio-8080-exec-7] c.m.s.filter.PostFilter                  : PostFilter : response's status is 500
```

Connect to localhost:7070. Connection was refused. That's because our API service isn't really up. That's what caused this error. At the very bottom, you can see that after the Pre and the RouteFilter, the ErrorFilter was run. After the ErrorFilter was executed, then the PostFilter was executed. You can see the response status in the Error as well as PostFilter is 500.

Sending Messages and Making Phone Calls

In this demo, we'll do something that's rather cool. We'll use the Twilio cloud platform and the Twilio SDK to make and receive phone calls from our Spring Boot application.

Here we are on twilio.com.



Respond to COVID-19 with agility and global scale. **Learn more.**

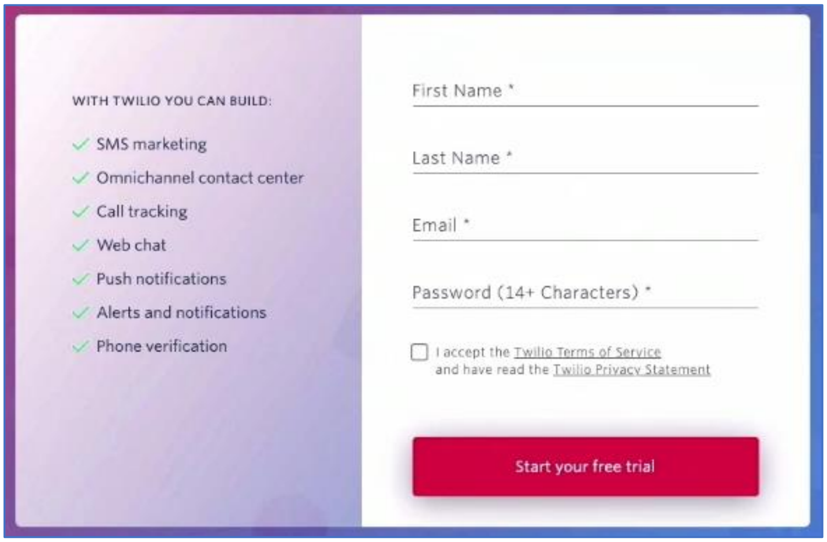
Build the new normal
for distance learning with Twilio Voice

Hello. Please say your name and the class you need help with.

My name is Aleena and I'm taking Philosophy 101.

Blackboard empowers schools to keep their students informed with Twilio

This is the website where we sign up for a Twilio account, click on sign up and start building. Twilio is a cloud communications platform as a service company that allows software developers to programmatically make and receive phone calls, send and receive text messages and perform other communication functions, using its web service APIs. I have filled up this form in order to create a new Twilio account using the email account bob@loonycorn.com. You need to verify that you're human before you can continue using Twilio.



WITH TWILIO YOU CAN BUILD:

- ✓ SMS marketing
- ✓ Omnichannel contact center
- ✓ Call tracking
- ✓ Web chat
- ✓ Push notifications
- ✓ Alerts and notifications
- ✓ Phone verification

First Name *

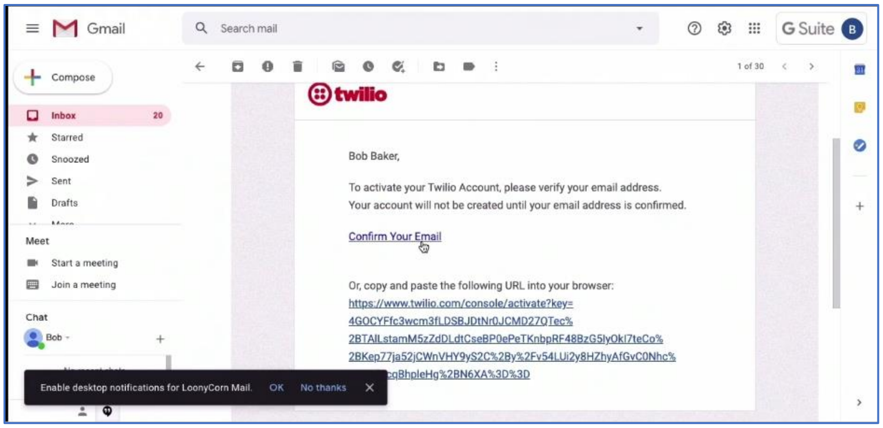
Last Name *

Email *

Password (14+ Characters) *

□ I accept the Twilio Terms of Service
and have read the Twilio Privacy Statement

So I'm going to head over to my bob@loonycorn.account. And here is my email from Twilio, click through. And you will find a link in there that allows you to confirm your email address click on that link.



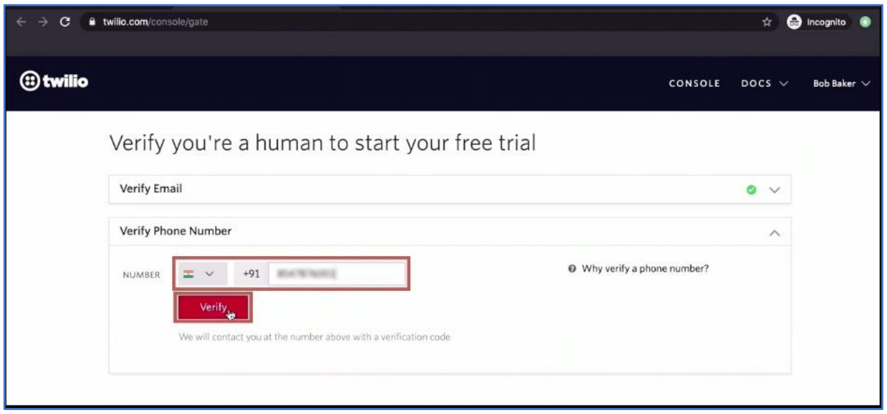
Bob Baker,

To activate your Twilio Account, please verify your email address.
Your account will not be created until your email address is confirmed.

Confirm Your Email

Or, copy and paste the following URL into your browser:
https://www.twilio.com/console/activate?key=
4GOCYFfc3wcm3fLDSBJDtNr0JCMD27QTec%
2BTAlLstamM5zZdDLdtCseBP0ePeTKnbpRF48BzG5lyOkl7teCo%
2BKep77ja52jCWnVHY9yS2C%2By%2Fv54LUi2y8HZhyAfGvC0Nhc%
cqBhpleHg%2BN6XA%3D%3D

This will open up a Twilio page where you can put in a phone number for verification.



Verify you're a human to start your free trial

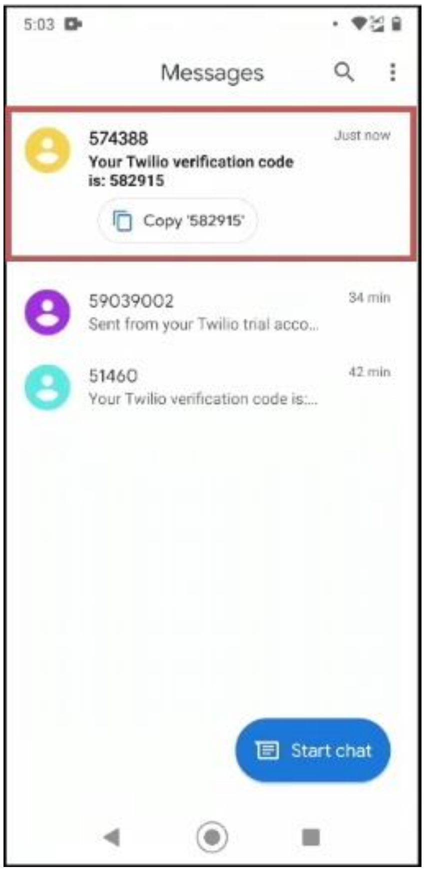
Verify Email ✓

Verify Phone Number

NUMBER +91 [redacted]

We will contact you at the number above with a verification code

Now make sure that you use a valid phone number on which you can receive a messages because when you hit verify. Twilio will send you a text message with a verification code. Now let's head over and look at our phone here. As I read on my phone, I get a text message from Twilio with a verification code. I'm going to click through and take a look at that verification code. You can see it here, it's 582915.



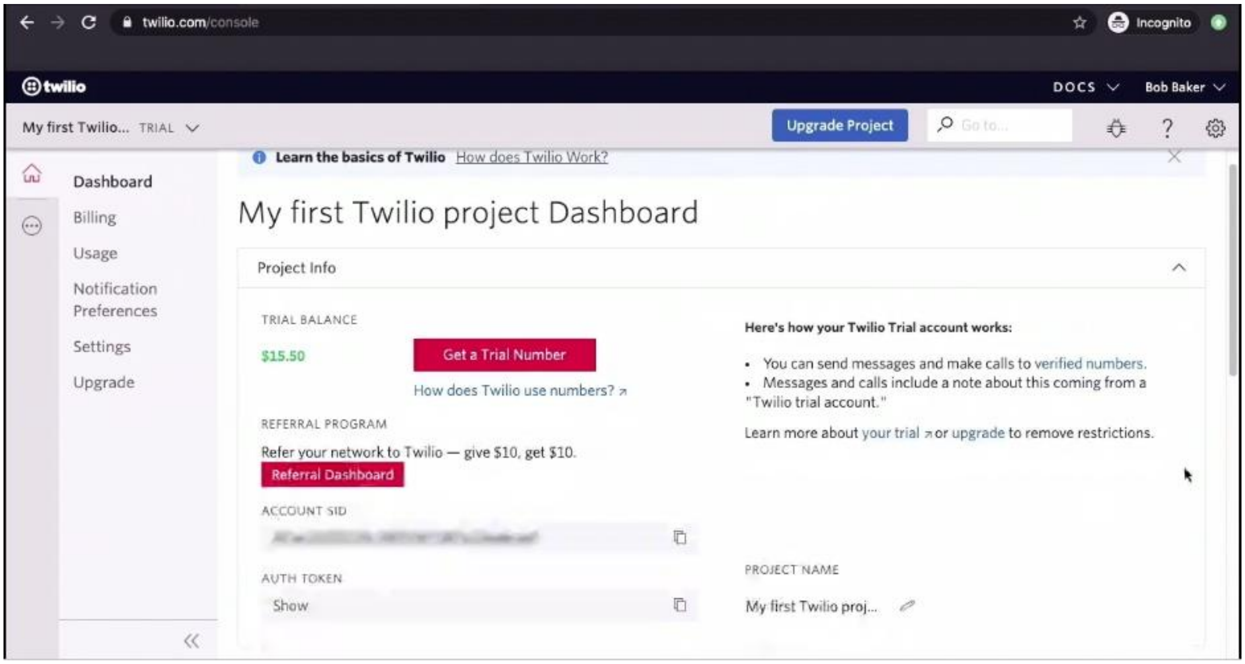
574388 Just now
Your Twilio verification code is: 582915
Copy '582915'

59039002 34 min
Sent from your Twilio trial acco...

51460 42 min
Your Twilio verification code is:...

This is the verification code that I need to enter on my Twilio account in order to verify myself. I hit submit, I'm all set up with a Twilio account now. Now there is a custom screen here asking you a bunch of questions. These questions have to do with how you plan to use Twilio. For our purposes, I'm simply going to say skip to dashboard because I want to get started. Using Twilio to send messages and make phone calls from within my spring boot application.

Skip to dashboard once again and here we are on the main Twilio dashboard. The first thing we need to do here is to get a trial number. This will be a US based number that Twilio will use to send SMS from your account.



Learn the basics of Twilio How does Twilio Work?

My first Twilio project Dashboard

Project Info

TRIAL BALANCE
$15.50

How does Twilio use numbers?

REFERRAL PROGRAM
Refer your network to Twilio — give $10, get $10.

ACCOUNT SID
[redacted]

AUTH TOKEN

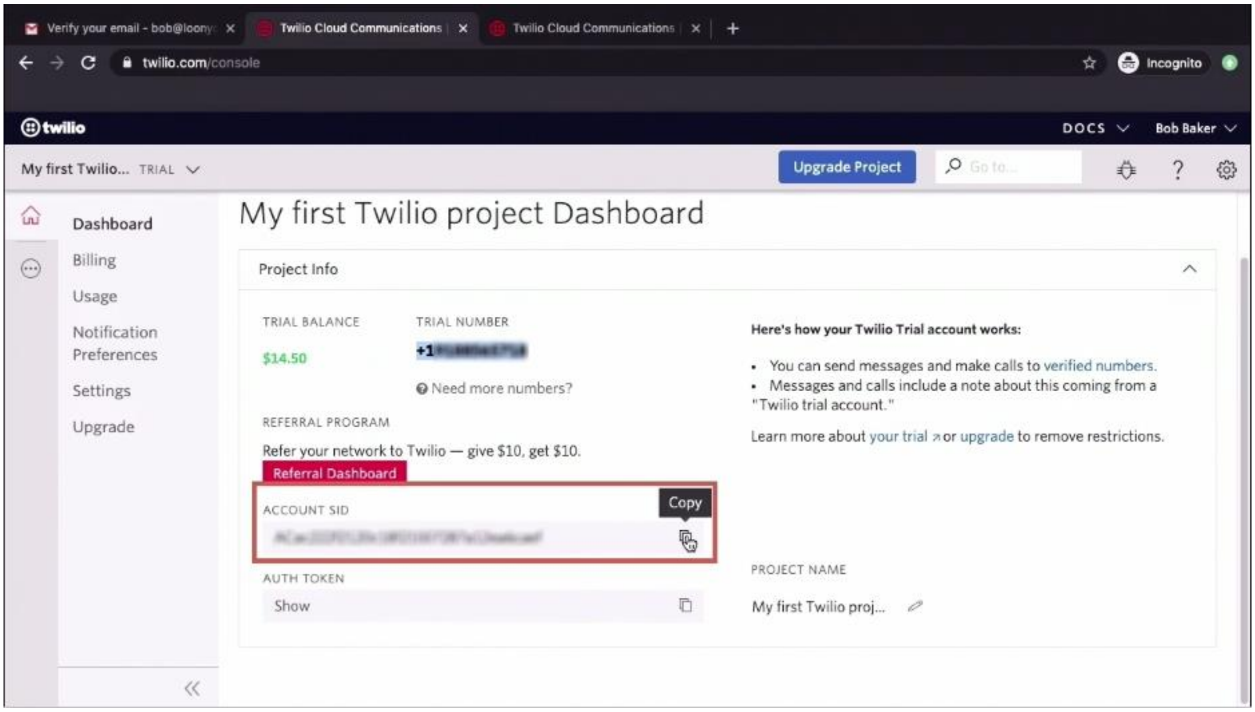
Here's how your Twilio Trial account works:

- You can send messages and make calls to verified numbers.
- Messages and calls include a note about this coming from a "Twilio trial account."

Learn more about your trial or upgrade to remove restrictions.

PROJECT NAME
My first Twilio proj...

Now you can choose whatever number has been displayed. So I'm going to simply click on choose this number and this randomly generated phone number will be associated with my Twilio account.



My first Twilio project Dashboard

Project Info

TRIAL BALANCE
$14.50

TRIAL NUMBER
+1[redacted]

Need more numbers?

REFERRAL PROGRAM
Refer your network to Twilio — give $10, get $10.

ACCOUNT SID
[redacted]

AUTH TOKEN

Here's how your Twilio Trial account works:

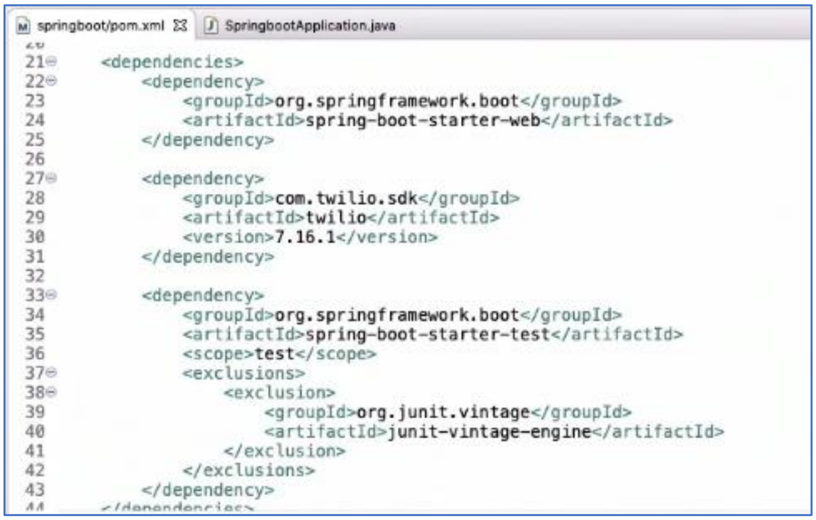
- You can send messages and make calls to verified numbers.
- Messages and calls include a note about this coming from a "Twilio trial account."

Learn more about your trial or upgrade to remove restrictions.

PROJECT NAME
My first Twilio proj...

We are now all set up with a Twilio phone number and ACCOUNT _SID and an AUTH token. Which means we're ready to head over to our Eclipse IDE and write code within our Spring Boot application.

The application itself is very straightforward. You can use any starter template I've chosen the Spring Boot starter web, make sure you depend on com.Twilio.sdk. The artifact ID is Twilio and the version is 7.16.1. Now let's take a look at our application code which you'll find this very simple.

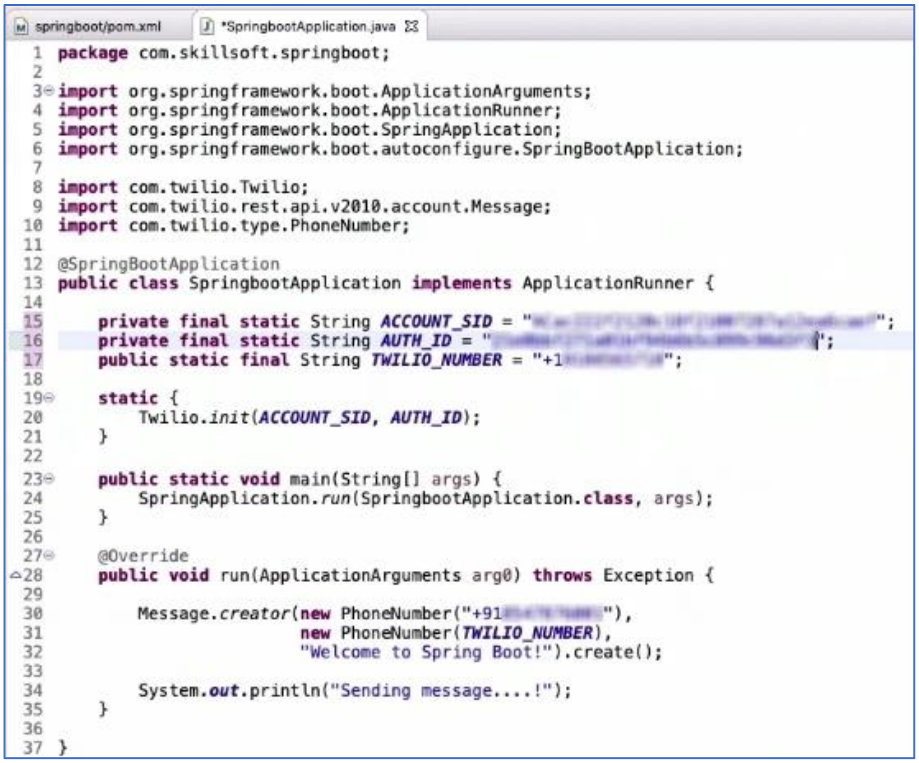


```xml
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>com.twilio.sdk</groupId>
            <artifactId>twilio</artifactId>
            <version>7.16.1</version>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
            <exclusions>
                <exclusion>
                    <groupId>org.junit.vintage</groupId>
                    <artifactId>junit-vintage-engine</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
    </dependencies>
```

Here is our basic Spring Boot application that implements the application runner interface. Just like the command line runner interface that we've seen before. The application runner essentially tells spring that this application is an executable, and this class should be executed or run as soon as it's instantiated.

Now in order to use Twilio we need three bits of information. The account set, the AUTH ID and the Twilio number. All of this is available on your Twilio dashboard, let's get it from there. Let's head over to the dashboard, copy over the US phone number here and paste it into the Twilio number field. The phone number that you have for your account will of course be different, just make sure you get the right number.

Now we need the account SID, which is available here in this account SID field. Click on the copy link that is present here and back in your Eclipse project, assign this account sid to the accounts sid static final string variable.



springboot/pom.xml | *SpringbootApplication.java

```
 1 package com.skillsoft.springboot;
 2 
 3 import org.springframework.boot.ApplicationArguments;
 4 import org.springframework.boot.ApplicationRunner;
 5 import org.springframework.boot.SpringApplication;
 6 import org.springframework.boot.autoconfigure.SpringBootApplication;
 7 
 8 import com.twilio.Twilio;
 9 import com.twilio.rest.api.v2010.account.Message;
10 import com.twilio.type.PhoneNumber;
11 
12 @SpringBootApplication
13 public class SpringbootApplication implements ApplicationRunner {
14 
15     private final static String ACCOUNT_SID = "[redacted]";
16     private final static String AUTH_ID = "[redacted]";
17     public static final String TWILIO_NUMBER = "+1[redacted]";
18 
19     static {
20         Twilio.init(ACCOUNT_SID, AUTH_ID);
21     }
22 
23     public static void main(String[] args) {
24         SpringApplication.run(SpringbootApplication.class, args);
25     }
26 
27     @Override
28     public void run(ApplicationArguments arg0) throws Exception {
29 
30         Message.creator(new PhoneNumber("+91[redacted]"),
31                 new PhoneNumber(TWILIO_NUMBER),
32                 "Welcome to Spring Boot!").create();
33 
34         System.out.println("Sending message....!");
35     }
36 
37 }
```

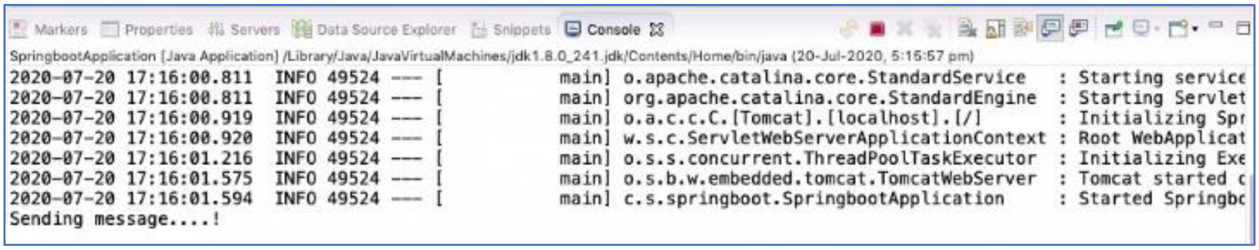
Now we need one more thing that is the AUTH token. Hit the Copy button here in your Twilio dashboard, back to Eclipse, assign this value to the AUTH_ID static variable. We're now all set ready to use Twilio. Now before you can use Twilio you need to call init on the Twilio SDK which we do within the static initialization block that you see on line 19. The main method for this application remains the same because spring application.run on the current class.

We also override the run method, which is part of the application runner interface. The input argument to this run method is of type application arguments. This is what is different between the application runner and the command line runner that we've seen earlier.

In the application runner, all of your input arguments are nicely wrapped up within this application arguments class. Now in order to send an SMS via Twilio, all you need to do is use **message.creator**, specify the phone number to which you want to send the message that is the first input argument.

Specify your Twilio number as well and specify the message our message will simply say welcome to Spring Boot. Invoke the create function on this message and Twilio will take care of the rest. It'll send the message to your phone.

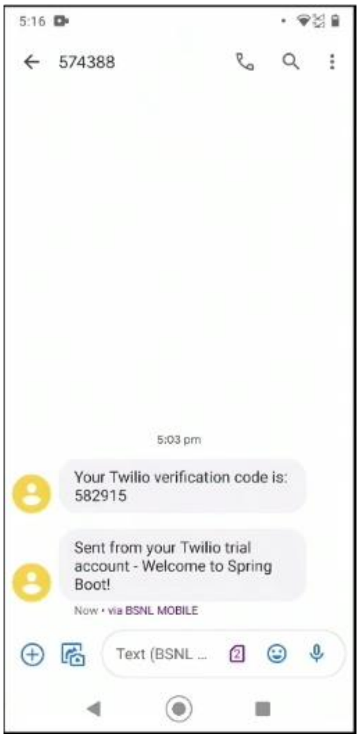
Go ahead and run this application. You can see the console message it says Sending message.



SpringbootApplication [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_241.jdk/Contents/Home/bin/java (20-Jul-2020, 5:15:57 pm)

```
2020-07-20 17:16:00.811  INFO 49524 --- [           main] o.apache.catalina.core.StandardService   : Starting service
2020-07-20 17:16:00.811  INFO 49524 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet
2020-07-20 17:16:00.919  INFO 49524 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spr
2020-07-20 17:16:00.920  INFO 49524 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicat
2020-07-20 17:16:01.216  INFO 49524 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor  : Initializing Exe
2020-07-20 17:16:01.575  INFO 49524 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started c
2020-07-20 17:16:01.594  INFO 49524 --- [           main] c.s.springboot.SpringbootApplication     : Started Springbo
Sending message....!
```

Let's take a look at our phone, and here on my phone I get the message from my Twilio trial account. The message says Welcome to Spring Boot!



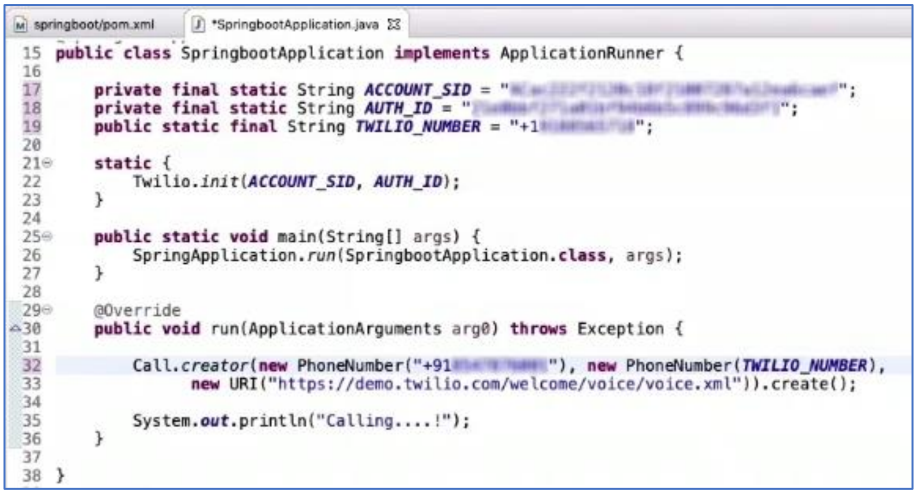
5:03 pm

Your Twilio verification code is: 582915

Sent from your Twilio trial account - Welcome to Spring Boot!

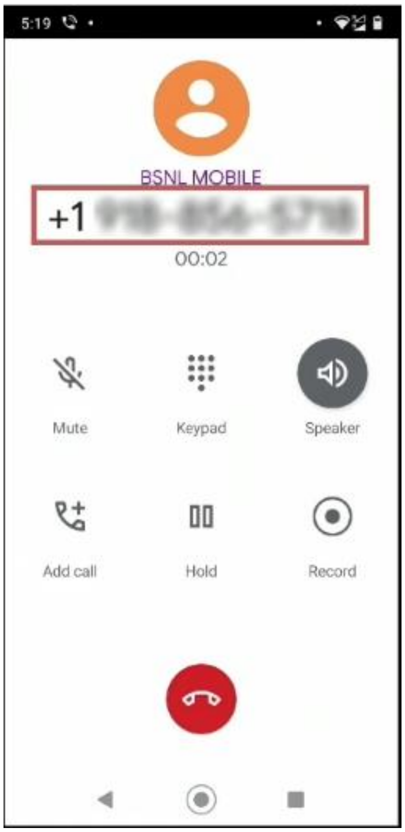
Now • via BSNL MOBILE

Well that was really easy. It turns out that making phone calls is also very simple in Twilio. Here's our application everything is exactly the same. The only change that we've made is in the code within the run method.



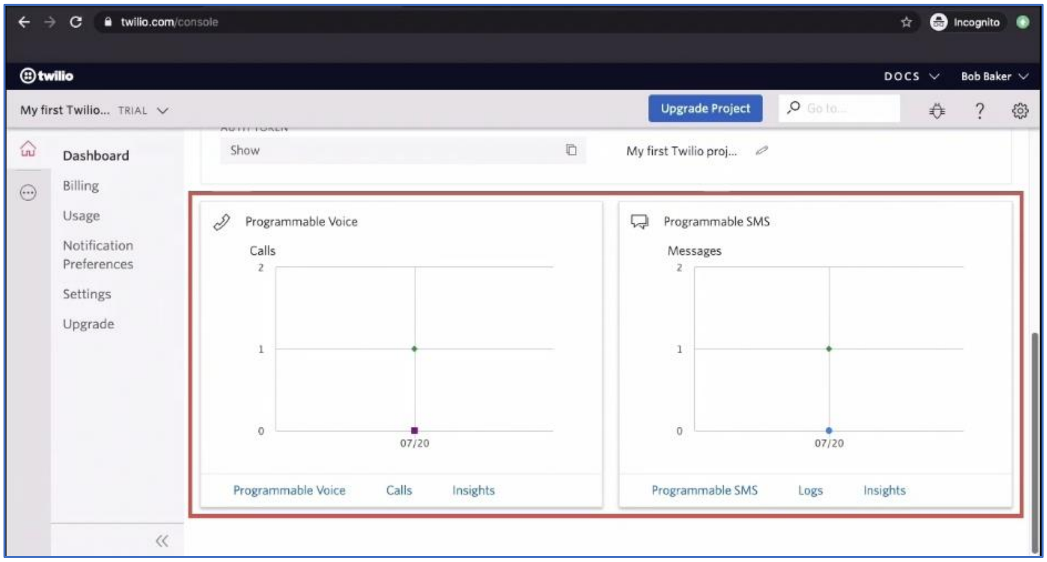
```
public class SpringbootApplication implements ApplicationRunner {

    private final static String ACCOUNT_SID = "[redacted]";
    private final static String AUTH_ID = "[redacted]";
    public static final String TWILIO_NUMBER = "+1[redacted]";

    static {
        Twilio.init(ACCOUNT_SID, AUTH_ID);
    }

    public static void main(String[] args) {
        SpringApplication.run(SpringbootApplication.class, args);
    }

    @Override
    public void run(ApplicationArguments arg0) throws Exception {

        Call.creator(new PhoneNumber("+91[redacted]"), new PhoneNumber(TWILIO_NUMBER),
                new URI("https://demo.twilio.com/welcome/voice/voice.xml")).create();

        System.out.println("Calling....!");
    }

}
```

Now inside of a message we use Call.creator, specify the number to which you want to make a call, specify your Twilio phone number, and then specify a URI. This is the standard Twilio URI to log the call that we've made the voice call, call create, and then print out calling. Run this code, and if you wait for about 15 seconds or so you'll receive a call on your phone number.



I'm going to head over to my phone recording. And as I wait, here is the incoming call from Twilio go ahead and answer this call. And you'll see that this number is your Twilio number and you'll hear an automated voice recording. When you're satisfied, you can go ahead and hang up. And that's it.

You've successfully use Twilio to send messages and make phone calls. Now in your Twilio dashboard, if you scroll down, you will see a graphical representation of the call that you made and the message that you sent. Twilio does automatically tracks all of your interactions with its SDK.

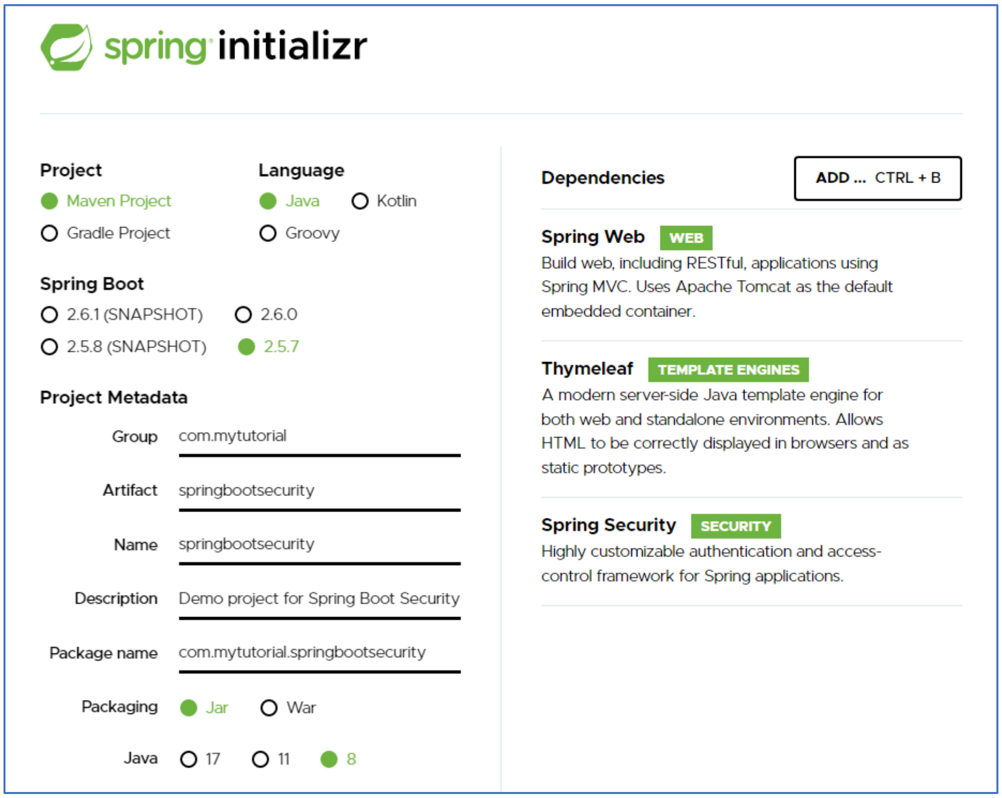


Create Spring Boot Web Application With Spring Security

What we've seen so far while working with Spring Boot is that with Spring Boot your spring applications just work, and the same is true for security as well.

1. Create Maven Project

Here I am on the spring initializr page where I'm going to set up a new project that uses Spring security. This is a Maven project, we'll write code in Java, we'll use Spring 2.3.1. We'll keep the same project metadata, the package team that will work with will be com.skillsoft.springboot. We'll configure dependencies for a simple web application that includes security, click on Add Dependencies.



Project
● Maven Project
○ Gradle Project

Language
● Java ○ Kotlin
○ Groovy

Spring Boot
○ 2.6.1 (SNAPSHOT) ○ 2.6.0
○ 2.5.8 (SNAPSHOT) ● 2.5.7

Project Metadata

Group com.mytutorial

Artifact springbootsecurity

Name springbootsecurity

Description Demo project for Spring Boot Security

Package name com.mytutorial.springbootsecurity

Packaging ● Jar ○ War

Java ○ 17 ○ 11 ● 8

Dependencies

Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

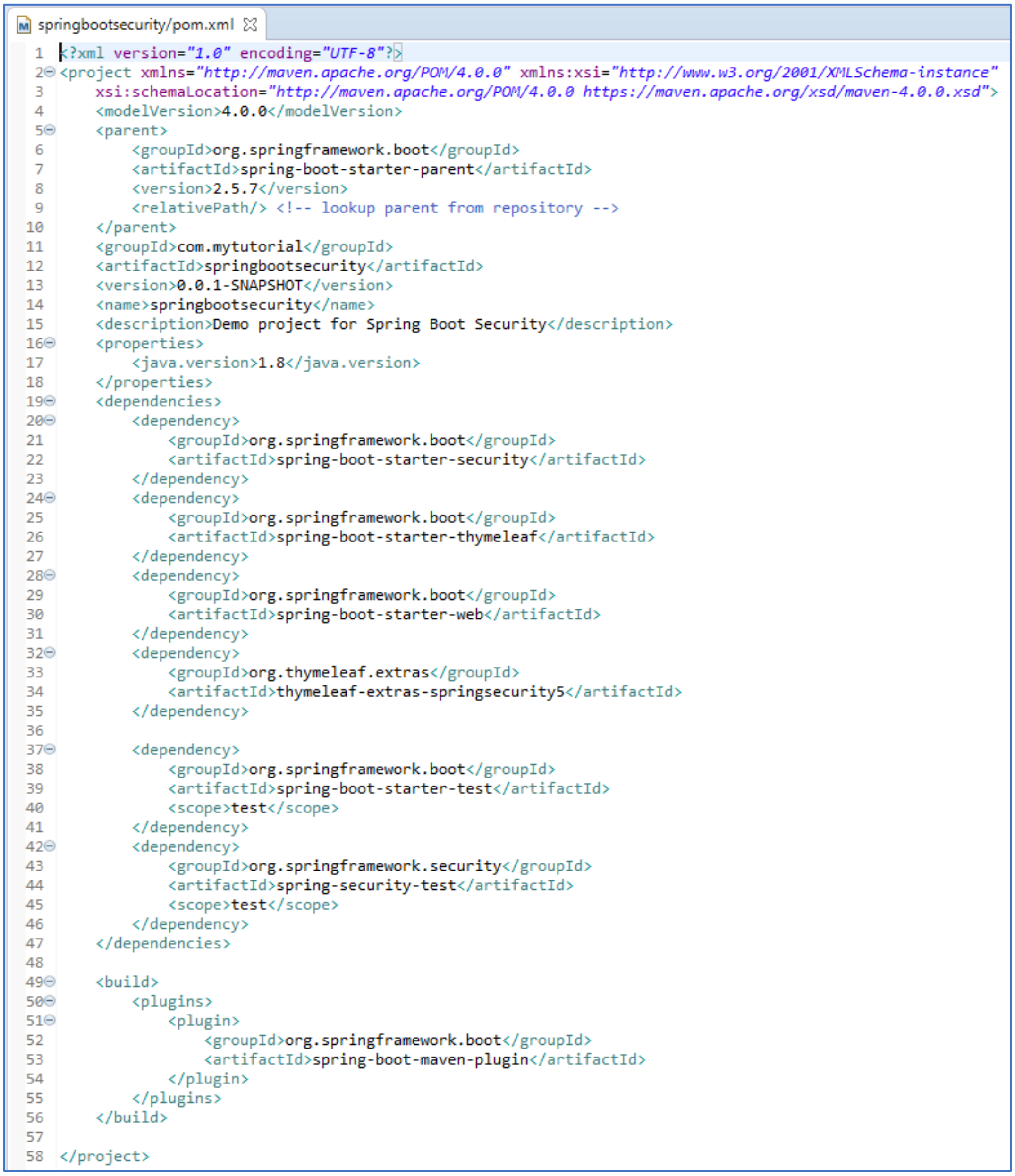
Thymeleaf TEMPLATE ENGINES
A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes.

Spring Security SECURITY
Highly customizable authentication and access-control framework for Spring applications.

We'll first select Spring Web, that's needed for any web application. In addition to Spring Web, we'll also use the Thymeleaf template engine, this is what will allow us to render our user interfaces. The new dependency that we'll configure to secure our web pages will be Spring Security.

The way Spring Security works is that if spring security is present on the class-path, Spring Boot automatically secures all HTTP endpoints with basic authentication. So make sure you add Spring Security as a dependency. Click on generate, that will generate and download a zip file onto your local machine containing the basic project structure.

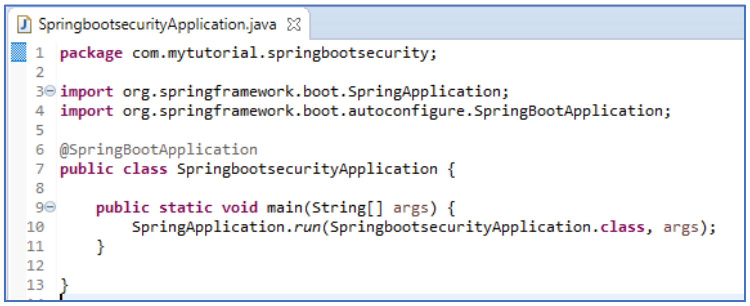
The **pom.xml** file for this project where we'll set up a simple login page includes the dependency Spring Boot starter security. Just by having the dependencies added by the starter template in your class path, all of your web pages will be automatically secured.



```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.5.7</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.mytutorial</groupId>
    <artifactId>springbootsecurity</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>springbootsecurity</name>
    <description>Demo project for Spring Boot Security</description>
    <properties>
        <java.version>1.8</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-security</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.thymeleaf.extras</groupId>
            <artifactId>thymeleaf-extras-springsecurity5</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

2. Inspect Spring Boot Main Entry Point Class

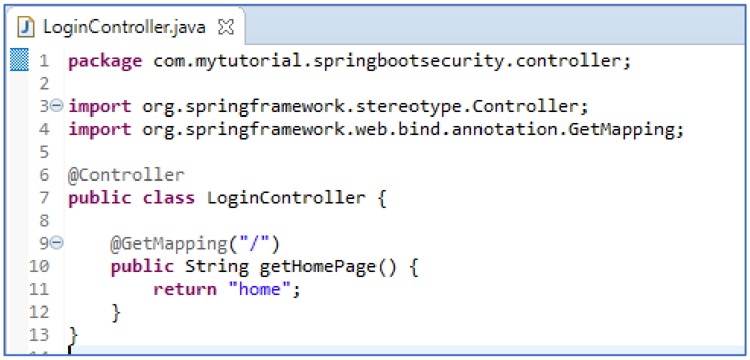
The main entry point to our Spring Boot application is the same as what we've seen before, very straightforward.



```java
package com.mytutorial.springbootsecurity;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringbootsecurityApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringbootsecurityApplication.class, args);
    }

}
```

3. Prepare Login Controller

Let's take a look at our login controller, which is annotated using the **@Controller** annotation, this is the controller that'll render are home page.

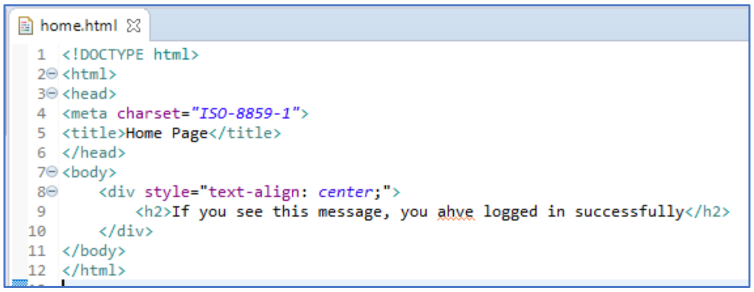


```java
package com.mytutorial.springbootsecurity.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class LoginController {

    @GetMapping("/")
    public String getHomePage() {
        return "home";
    }
}
```

We have a simple get mapper handler for the root part of our application which renders the home view.

4. Prepare View Layer Html Page

The home view is in the **home.html** file that simply says if you see this message you've logged in successfully.

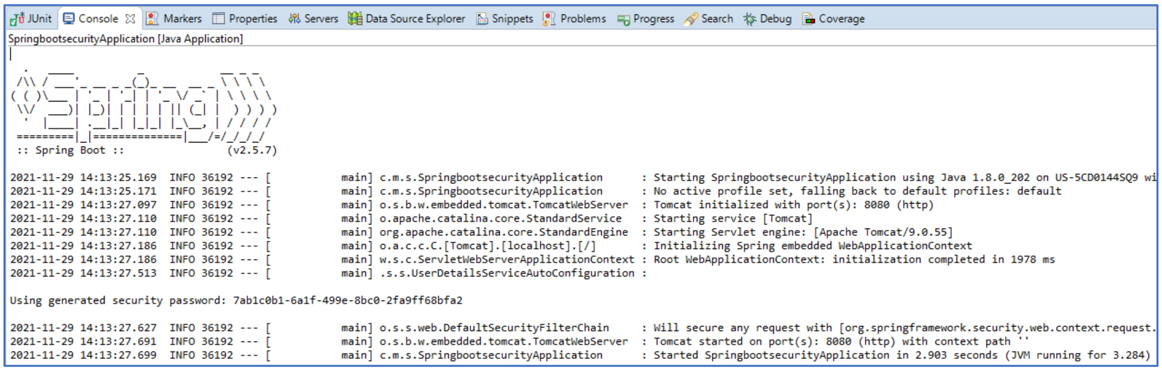


```html
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Home Page</title>
</head>
<body>
    <div style="text-align: center;">
        <h2>If you see this message, you ahve logged in successfully</h2>
    </div>
</body>
</html>
```

With spring security, all of our web pages will be behind a login screen automatically.

5. Run and Test Application

There's nothing else that you need to configure, simply run your application, and within your console window, you will find a user generated security password. When you haven't explicitly specified the user for our application, Spring Security will auto configure a user for you behind the scenes. This is the password that you will use to log in as that user.



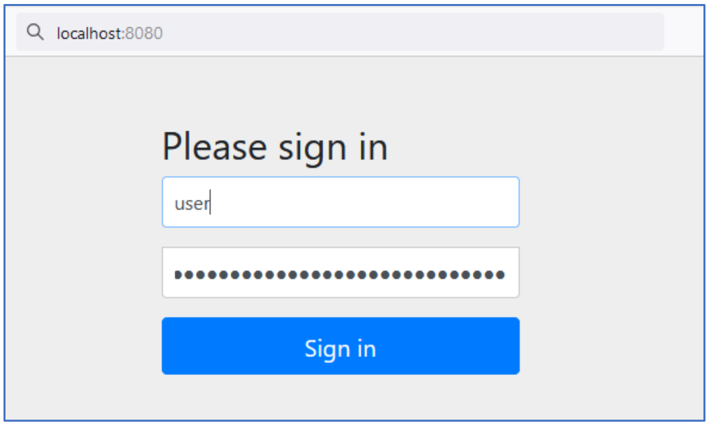
```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.5.7)

2021-11-29 14:13:25.169  INFO 36192 --- [           main] c.m.s.SpringbootsecurityApplication      : Starting SpringbootsecurityApplication using Java 1.8.0_202 on US-5CD0144SQ9 wi
2021-11-29 14:13:25.171  INFO 36192 --- [           main] c.m.s.SpringbootsecurityApplication      : No active profile set, falling back to default profiles: default
2021-11-29 14:13:27.097  INFO 36192 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2021-11-29 14:13:27.110  INFO 36192 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2021-11-29 14:13:27.110  INFO 36192 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.55]
2021-11-29 14:13:27.186  INFO 36192 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2021-11-29 14:13:27.186  INFO 36192 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1978 ms
2021-11-29 14:13:27.513  INFO 36192 --- [           main] .s.s.UserDetailsServiceAutoConfiguration : 

Using generated security password: 7ab1c0b1-6a1f-499e-8bc0-2fa9ff68bfa2

2021-11-29 14:13:27.627  INFO 36192 --- [           main] o.s.s.web.DefaultSecurityFilterChain     : Will secure any request with [org.springframework.security.web.context.request.
2021-11-29 14:13:27.691  INFO 36192 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2021-11-29 14:13:27.699  INFO 36192 --- [           main] c.m.s.SpringbootsecurityApplication      : Started SpringbootsecurityApplication in 2.903 seconds (JVM running for 3.284)
```

Now if you scroll over to the right, you will see that Spring Security has set up a filter security interceptor.

org.springframework.security.web.access.intercept.FilterSecurityInterceptor@238acd0b

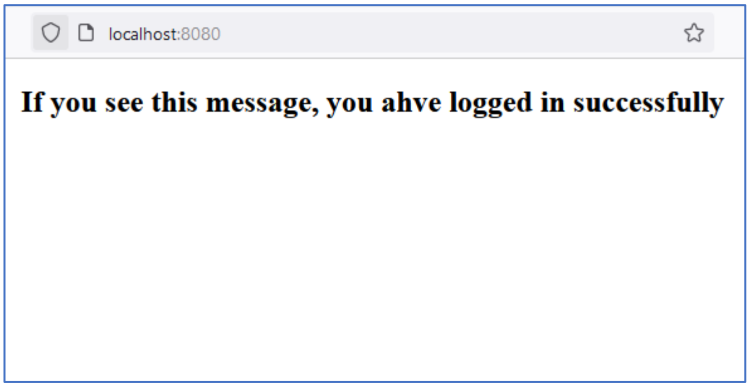
This interceptor essentially intercepts all requests that you make to this application and ensures that you're logged in before the request is satisfied.

Let's copy over this security password, and let's hit our app using our web browser. Let's go to *localhost:8080*, and immediately, you'll see a nice login page here, this login page is powered using Bootstrap UI. The auto configured default user that has been set up has the username, just user.



Go ahead and paste in the password that was generated within our console messages.

Now if you click on sign in, you will be logged in and you can see the message on the homepage.



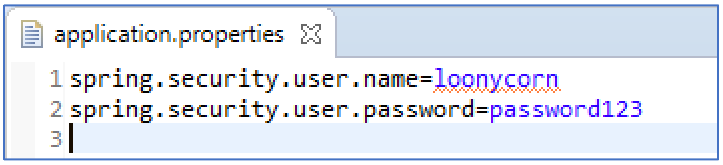
If you see this message, you ahve logged in successfully

Because Spring Security was on our class path, we can't access any of our web endpoints without logging in.

6. Configuring with fixed custom user and password

What if you don't want to use the default user that has been configured for you, what if you want to specify your own username and password?

The easiest way to do this is to use your application.properties file.



```
spring.security.user.name=loonycorn
spring.security.user.password=password123
```

You can configure spring.security.user.name, set it to whatever username you want, and set the corresponding password for this user name in spring.security.user.password. Go ahead and run this code, now the default auto generated user will no longer be created.

Instead, you can log into your app using *loonycorn* and *password123*, the username and password that we have specified. Let's head over to localhost:8080. Because we haven't logged in yet, we'll be directed to the login page, which is in the [url]/login, this path mapping is added automatically by Spring Security.

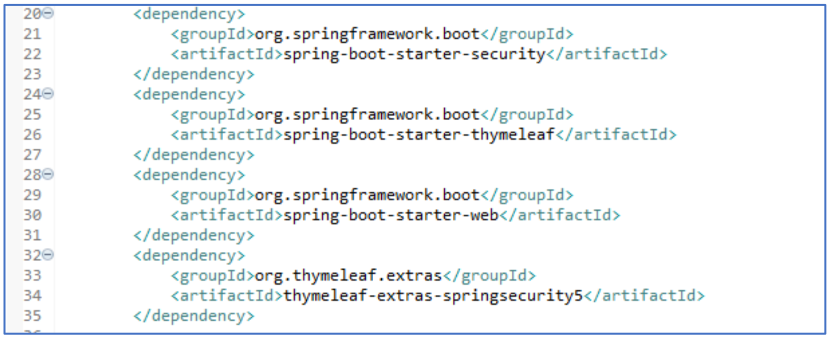
Now log in using the username and password that we have specified in the application.properties file. Spring security will automatically perform the lookup for you and log you in successfully if everything matches.

Configuring In-memory Users

Now it's quite possible that you want your web application to have multiple users and maybe these users belong to different roles, such as a regular user, an administrator, and so on. Maybe you also want to configure certain pages as being behind a login, and other pages are freely accessible.

With Spring Security all of this is possible.

Let's take a look at the **pom.xml** and notice that we have the spring-boot-starter-security template.



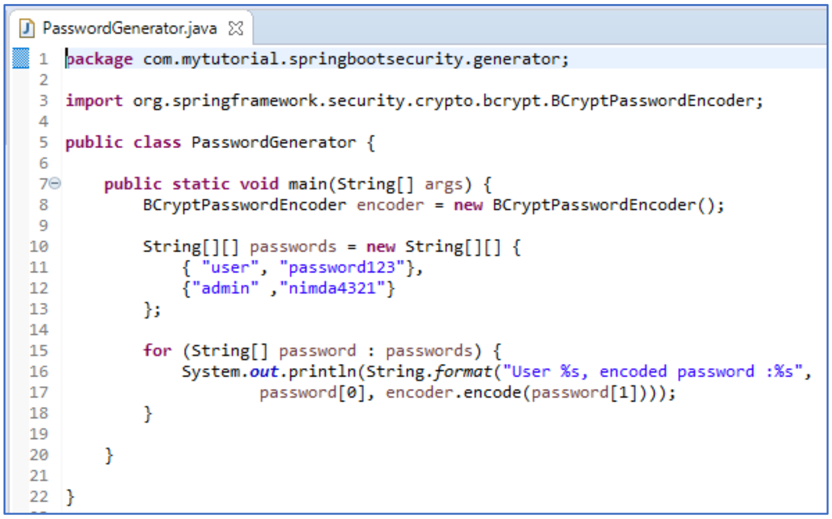
```xml
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-security</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.thymeleaf.extras</groupId>
            <artifactId>thymeleaf-extras-springsecurity5</artifactId>
        </dependency>
```

Since this is a web application, we'll use spring-boot-starter-web and spring-boot-starter-thymeleaf as well. In this application, we are going to configure some in memory users and authenticate against those users when we log in.

7. Create Password Generator Class

Now in order to generate a password for these in memory users, I'm going to use this executable class here, PasswordGenerator. As a best practice, you never store passwords in plain text either within your application or in your database.

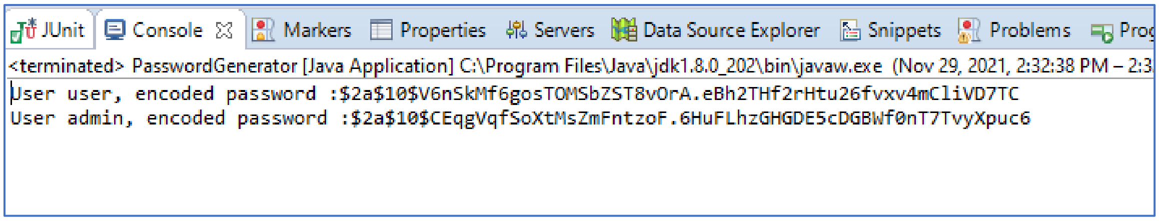
You always encrypt your passwords. When you're working with Spring Security, you can use one of those password encoders that it offers. The one that I have chosen here is the BCryptPasswordEncoder, which uses the BCrypt hashing algorithm to encode passwords.



```java
package com.mytutorial.springbootsecurity.generator;

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

public class PasswordGenerator {

    public static void main(String[] args) {
        BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();

        String[][] passwords = new String[][] {
            { "user", "password123"},
            {"admin" ,"nimda4321"}
        };

        for (String[] password : passwords) {
            System.out.println(String.format("User %s, encoded password :%s",
                    password[0], encoder.encode(password[1])));
        }

    }

}
```

I'm now going to encode two passwords corresponding to the two users that I'll set up for my app. The first password is for the user role. The password is *password123*. I'll generate an encoded password using **encoder.encode** and I'll print out this encoded password.

The second password is the admin Password. This will be for our admin user that is *nimda4321*. I'm going to select this password generator file within my project, right-click, choose Run As Java application, and generate the encoded passwords for the user as well as the admin.

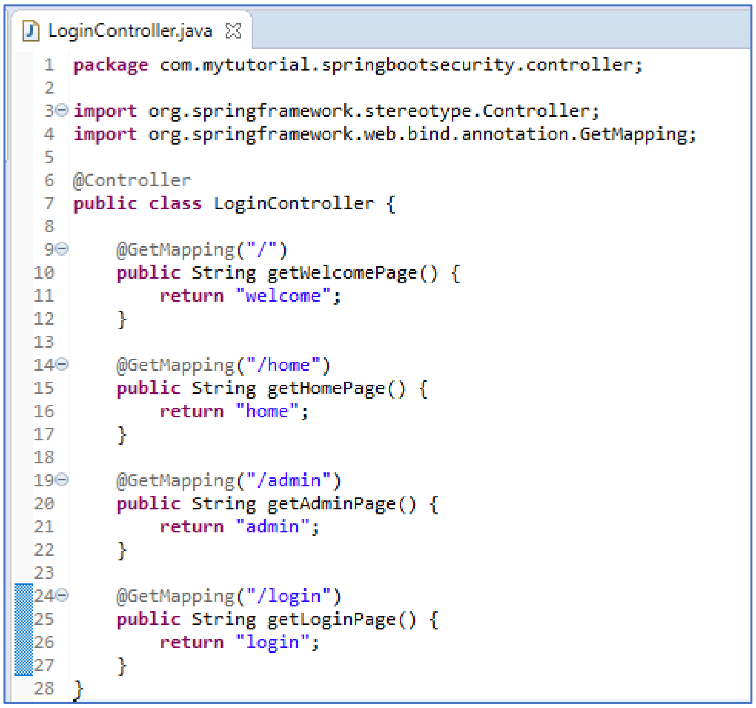


```
<terminated> PasswordGenerator [Java Application] C:\Program Files\Java\jdk1.8.0_202\bin\javaw.exe (Nov 29, 2021, 2:32:38 PM – 2:3
User user, encoded password :$2a$10$V6nSkMf6gosTOMSbZST8vOrA.eBh2THf2rHtu26fvxv4mCliVD7TC
User admin, encoded password :$2a$10$CEqgVqfSoXtMsZmFntzoF.6HuFLhzGHGDE5cDGBWf0nT7TvyXpuc6
```

I'm going to leave these encoded passwords within my console window. We'll use them in just a bit. Now the advantage of using a separate password generator file is that once you have the password, you can get rid of this file altogether. This means that the original plain text password need not be part of your code or your properties file.

8. Update Login controller

Let's take a look at the login controller first, so that we can see what web pages and what paths are mapped within this app.

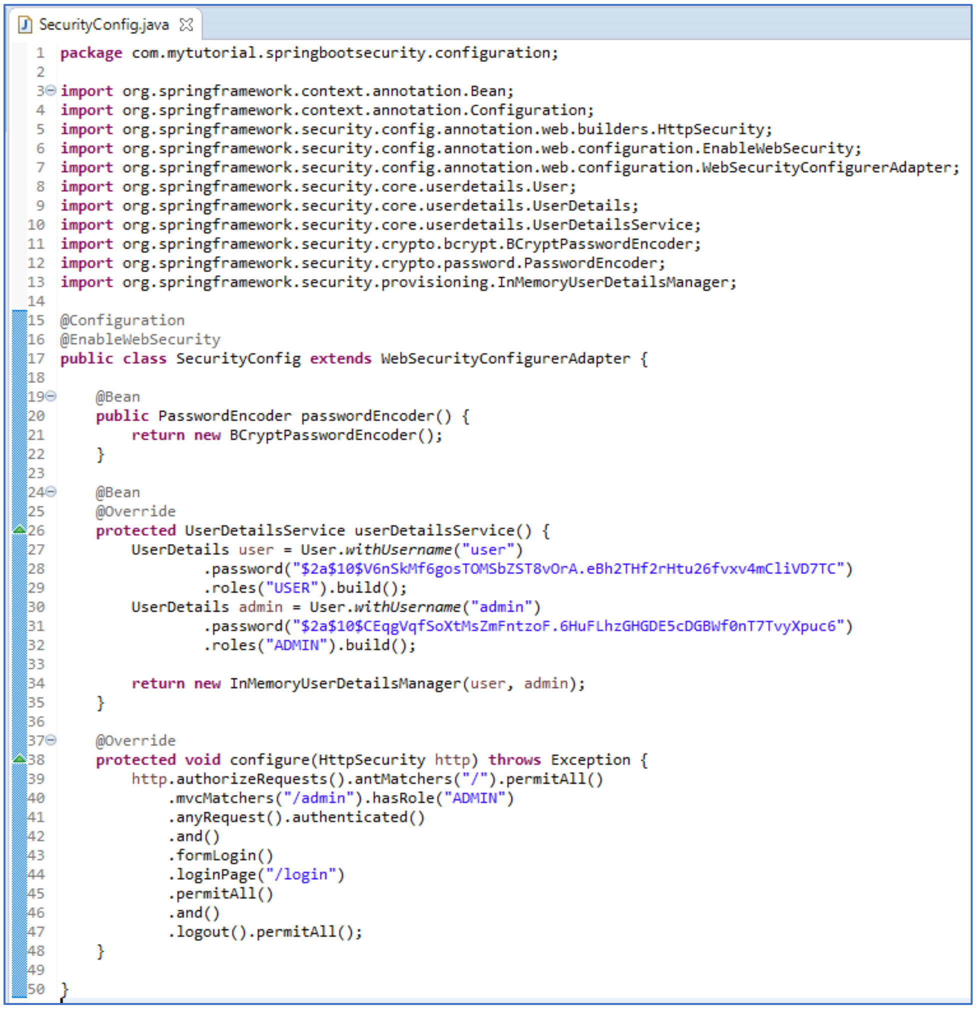


```java
package com.mytutorial.springbootsecurity.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class LoginController {

    @GetMapping("/")
    public String getWelcomePage() {
        return "welcome";
    }

    @GetMapping("/home")
    public String getHomePage() {
        return "home";
    }

    @GetMapping("/admin")
    public String getAdminPage() {
        return "admin";
    }

    @GetMapping("/login")
    public String getLoginPage() {
        return "login";
    }
}
```

We have a GetMapping for `"/"` which simply displays a *welcome page*. We have a GetMapping for `"/home"` that will render the *homepage*. We have a get mapping for `"/admin"` that will render an *administrator's page*. This should be accessible only to administrators. And finally, the login page is available at `"/login"`.

9. Create Security Configuration Class

I'll now head over to the SecurityConfig.java file where we configure the security settings for our web application. Notice that this class, SecurityConfig, extends the WebSecurityConfigurerAdapter.



```java
package com.mytutorial.springbootsecurity.configuration;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.provisioning.InMemoryUserDetailsManager;

@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    @Override
    protected UserDetailsService userDetailsService() {
        UserDetails user = User.withUsername("user")
                .password("$2a$10$V6nSkMf6gosTOMSbZST8vOrA.eBh2THf2rHtu26fvxv4mCliVD7TC")
                .roles("USER").build();
        UserDetails admin = User.withUsername("admin")
                .password("$2a$10$CEqgVqfSoXtMsZmFntzoF.6HuFLhzGHGDE5cDGBWf0nT7TvyXpuc6")
                .roles("ADMIN").build();

        return new InMemoryUserDetailsManager(user, admin);
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests().antMatchers("/").permitAll()
            .mvcMatchers("/admin").hasRole("ADMIN")
            .anyRequest().authenticated()
            .and()
            .formLogin()
            .loginPage("/login")
            .permitAll()
            .and()
            .logout().permitAll();
    }

}
```

This base class allows us to configure specific security settings for our application by overriding a few of its methods. Note the annotations on our security config class, **@Configuration** indicates that this is a class that contains configuration properties.

The **@EnableWebSecurity** is what enables Spring Security's web security support and also provides integration with the Spring MVC module that we're using to construct our web app.

Remember that passwords are not stored in plain text anywhere, not even in memory. Passwords are stored in an encrypted form. Which means when a user uses his password to log in, we need to encrypt that password and compare encrypted passwords.

That means our application needs to know the password encrypter that we've used.

The BCryptPasswordEncoder is what we have specified here as a bean. The base class, the WebSecurityConfigurerAdapter, has a method named userDetailsService. This is the method that you need to override to specify the store for the users who can access your application. The UserDetails service will access the user's store in order to look up a particular user.

Your users can be stored in any kind of data store. Maybe it's a MySQL database. Maybe it's some other external database on the cloud. Here in order to keep things simple, we are using an in memory store.

On line 28, we set up the first user here. Username is *user*. We'll still have to specify the password for this user and the role is that of a user.

The second user instantiated on line 31 is an administrator of our app. Username is admin. We haven't specified the password yet and the role for this user is ADMIN.

Now remember the passwords that we are generated for our user and administrator. Let's now copy these passwords over and paste them in. Here is the user password.

Next, I'll copy the admin password and specify that within our code as well. Notice that we're only dealing with the encrypted form of the passwords, not with the plain text password. Both of these users are in memory, they're not stored in an actual database. We use the **InMemoryUserDetailsManager** to indicate that these are the two users of our app.

The base class, **WebSecurityConfigurerAdapter**, also allows you to override the Configure method. Which is where you specify what URL paths in your application should be secured and which URLs are freely available without the user having to log in.

Let's take a look at our configuration.

```
	http.authorizeRequests().antMatchers("/").permitAll()
```

This line essentially means that any request to the root of our web application should be permitted to all users without log in.

The next line of our configuration says

```
		.mvcMatchers("/admin").hasRole("ADMIN")
```

Any request made to the /admin path requires the user to be logged in as an administrator of our website with the ADMIN role.

For all other requests, the requests have to be authenticated and have to be behind a login page.

```
		.anyRequest().authenticated()
		.and()
		.formLogin()
```

So **formLogin()** essentially allows us to configure the loginPage. Notice that we have set the loginPage at the path "/login".

```
		.loginPage("/login")
```

For any loginPage, the username and password will be assumed to be in the form fields called username and password by default.

The permitAll on line 45 applies to the login page.

```
		.permitAll()
```

All users can access the login page without logging in.

And the permitAll on line 47 applies to the logout page.

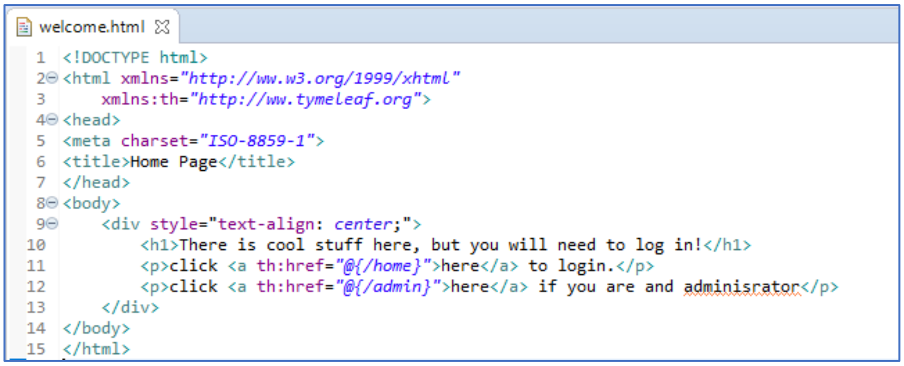
```
                .and()
                .logout().permitAll();
```

The logout page is accessible to all users without them having to be logged in.

This completes our server side configuration.

10. Prepare additional View Layer Page

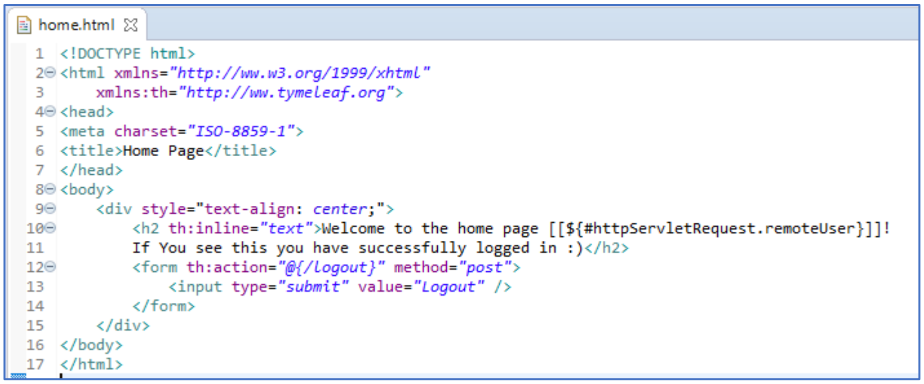
Let's take a look at some of the web pages that we'll display in our app. **welcome.html** is available to all users without them having to log in. There are two links on this page. The first is on line 11, which maps to the path */home*.



```
1  <!DOCTYPE html>
2  <html xmlns="http://ww.w3.org/1999/xhtml"
3      xmlns:th="http://ww.tymeleaf.org">
4  <head>
5  <meta charset="ISO-8859-1">
6  <title>Home Page</title>
7  </head>
8  <body>
9      <div style="text-align: center;">
10         <h1>There is cool stuff here, but you will need to log in!</h1>
11         <p>click <a th:href="@{/home}">here</a> to login.</p>
12         <p>click <a th:href="@{/admin}">here</a> if you are and adminisrator</p>
13     </div>
14 </body>
15 </html>
```

Any user can view this home page. The second link is on line 12, which maps to the path */admin*. Only administrators can view this page.

Let's now look at **home.html**. Here we have a header which says Welcome to the home page, an httpServletRequest.remoteUser will allow us to access the username from this web page.

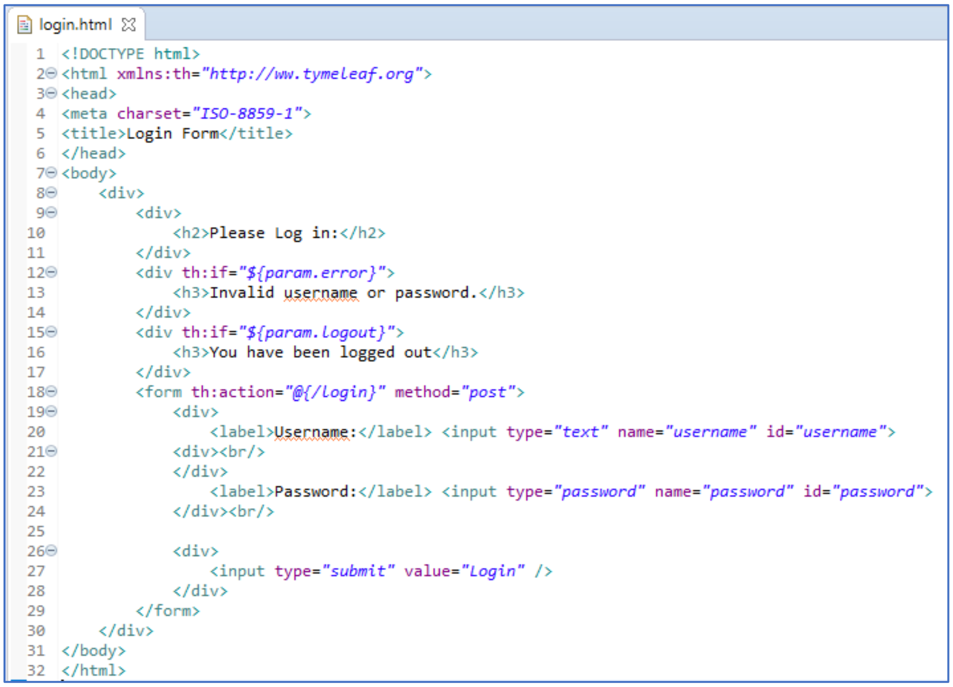


```
<!DOCTYPE html>
<html xmlns="http://ww.w3.org/1999/xhtml"
    xmlns:th="http://ww.tymeleaf.org">
<head>
<meta charset="ISO-8859-1">
<title>Home Page</title>
</head>
<body>
    <div style="text-align: center;">
        <h2 th:inline="text">Welcome to the home page [[${#httpServletRequest.remoteUser}]]!
        If You see this you have successfully logged in :)</h2>
        <form th:action="@{/logout}" method="post">
            <input type="submit" value="Logout" />
        </form>
    </div>
</body>
</html>
```

The message here on this page is visible only if you've successfully logged in as a user or an admin.

Notice that we have a form with the logout button here at the bottom of this page. The action corresponding to this form is simply **/logout**.

Next, let's take a look at the **login.html** page here.



```html
<!DOCTYPE html>
<html xmlns:th="http://ww.tymeleaf.org">
<head>
<meta charset="ISO-8859-1">
<title>Login Form</title>
</head>
<body>
    <div>
        <div>
            <h2>Please Log in:</h2>
        </div>
        <div th:if="${param.error}">
            <h3>Invalid username or password.</h3>
        </div>
        <div th:if="${param.logout}">
            <h3>You have been logged out</h3>
        </div>
        <form th:action="@{/login}" method="post">
            <div>
                <label>Username:</label> <input type="text" name="username" id="username">
            <div><br/>
            </div>
                <label>Password:</label> <input type="password" name="password" id="password">
            </div><br/>

            <div>
                <input type="submit" value="Login" />
            </div>
        </form>
    </div>
</body>
</html>
```

We display two messages here in this login page. If the error request param is set, this means this we've tried to log in with an invalid username, or password. Spring Security adds the error param automatically.

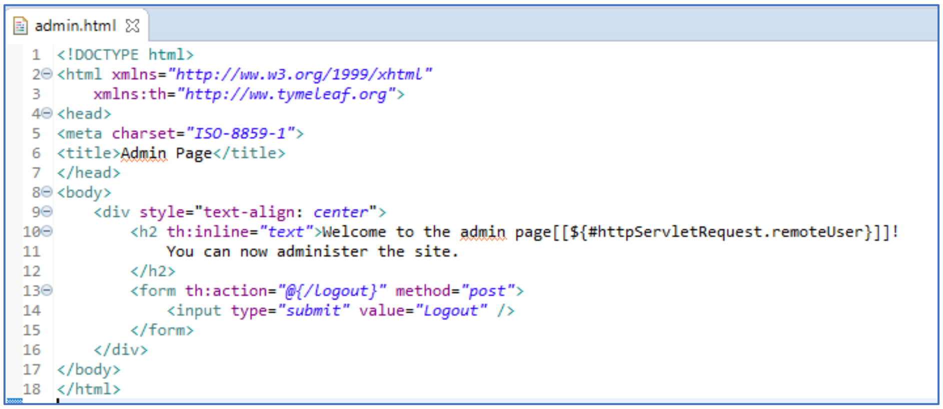
If the logout param is set, we've been logged out.

Once again the logout param is added by Spring Security automatically. The rest of the page contents include a simple form, which is mapped to the action /Login. This is the form that we'll use to log into our web app.

We accept a username and password. The name of the username input box should be username, and the name of the password input box should be password.

These are the defaults that Spring Security expects.

One last page to look at, **admin.html**. This is only accessible to the administrators of our application. This says you can now administer the site.



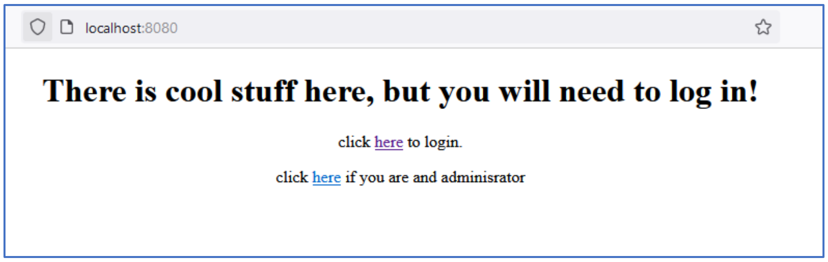
```html
<!DOCTYPE html>
<html xmlns="http://ww.w3.org/1999/xhtml"
    xmlns:th="http://ww.tymeleaf.org">
<head>
<meta charset="ISO-8859-1">
<title>Admin Page</title>
</head>
<body>
    <div style="text-align: center">
        <h2 th:inline="text">Welcome to the admin page[[${#httpServletRequest.remoteUser}]]!
            You can now administer the site.
        </h2>
        <form th:action="@{/logout}" method="post">
            <input type="submit" value="Logout" />
        </form>
    </div>
</body>
</html>
```

Once again at the bottom we have a form that allows us to log out from the app map to the action **/logout**.

All that's left is for us to run this code and see how login and logout works for the users and administrators in this app. Notice that the file security interceptor is automatically added to our app thanks to our use of Spring Security.

Configuring Login Roles

Now that we've configured the security settings for our application using in memory users, let's take our application for a test run. Hit localhost:8080, this will take you to the welcome page of the app.



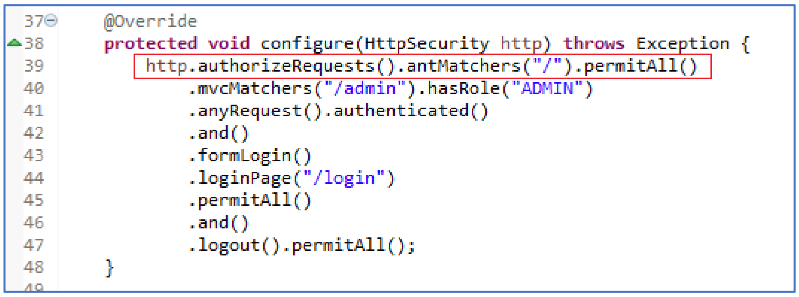
There is cool stuff here, but you will need to log in!

click here to login.

click here if you are and adminisrator

Remember, the welcome page can be displayed to the user without the user having to log in. This is because of the line in our configuration which says `http.authorizeRequests().antMatchers("/").permitAll()`

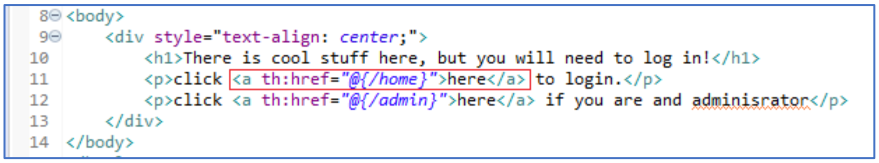
SecurityConfig.java



```
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests().antMatchers("/").permitAll()
            .mvcMatchers("/admin").hasRole("ADMIN")
            .anyRequest().authenticated()
            .and()
            .formLogin()
            .loginPage("/login")
            .permitAll()
            .and()
            .logout().permitAll();
    }
```

The first line here within this method, let's click on the first link here in this page, if you remember, this takes us to the path `"/home"`, which is a secured page.

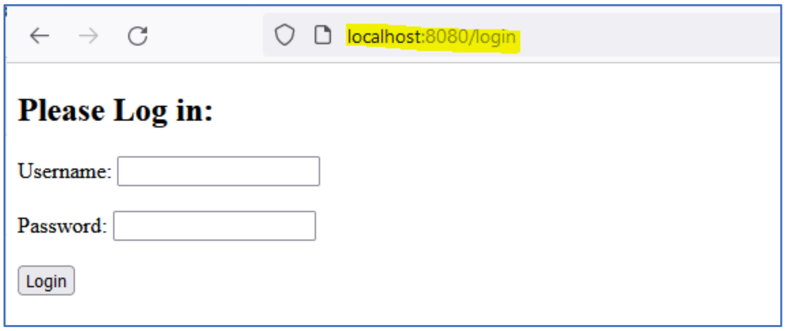
welcome.html



```
<body>
    <div style="text-align: center;">
        <h1>There is cool stuff here, but you will need to log in!</h1>
        <p>click <a th:href="@{/home}">here</a> to login.</p>
        <p>click <a th:href="@{/admin}">here</a> if you are and adminisrator</p>
    </div>
</body>
```

You can't access that page without logging in.

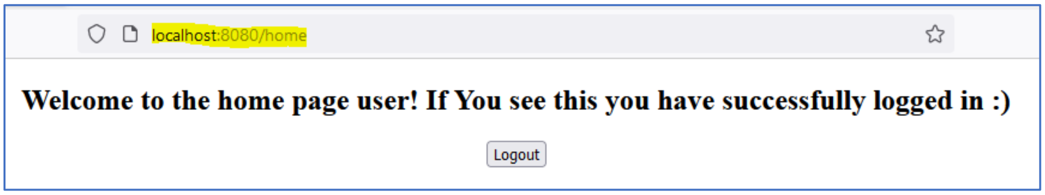
Once we click on this URL, we're taken to the login page `/login`.



Please Log in:

If you remember the settings that we have configured, we had said that the login page should be at the path `/login` and that's what the URL here is. Let's log in using the user account. The username was "*user*" and the password if you remember was "*password123*". This User is a plain user and not an admin, to be type in the password password123. This password will be encrypted and then compared against the password that we have specified for our in memory "user".

Once you click the Login button, you will be redirected to the original page that you had requested, the homepage `@{/home}`. Take a look at the URL we're `@{/home}`.

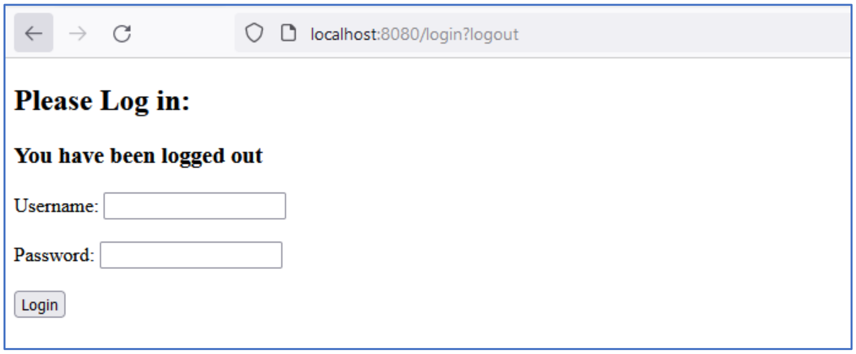


Welcome to the home page user! If You see this you have successfully logged in :)

We have successfully logged in, we see the success message, and you can see the button there which allows us to logout.

Once you've logged in, this information is stored within the string session till you explicitly log out. Let's say you go back to the Welcome page and click on Click here to log in. Once you've logged in, you'll be taken directly to the homepage. You don't need to log in again.

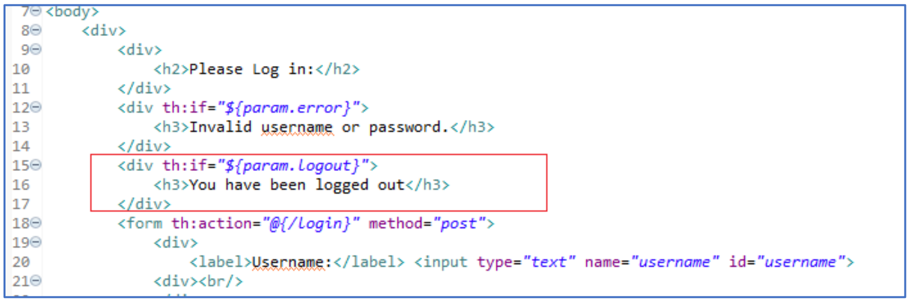
Let's log out as the user. And this time we'll try and access the page which requires an admin role.



Please Log in:

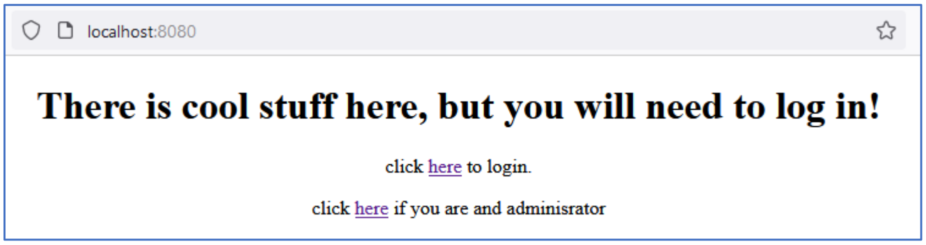
You have been logged out

Notice the message here, you have been logged out. That's because if you look at the URL, the logout request parameter has been set by spring. And we had a thyme leaf if **param.logout** will display the message you have been logged out.



```
<body>
    <div>
        <div>
            <h2>Please Log in:</h2>
        </div>
        <div th:if="${param.error}">
            <h3>Invalid username or password.</h3>
        </div>
        <div th:if="${param.logout}">
            <h3>You have been logged out</h3>
        </div>
        <form th:action="@{/login}" method="post">
            <div>
                <label>Username:</label> <input type="text" name="username" id="username">
            <div><br/>
```

Let's head back here to our main welcome page. Remember this Welcome page can be accessed without logging in.



localhost:8080

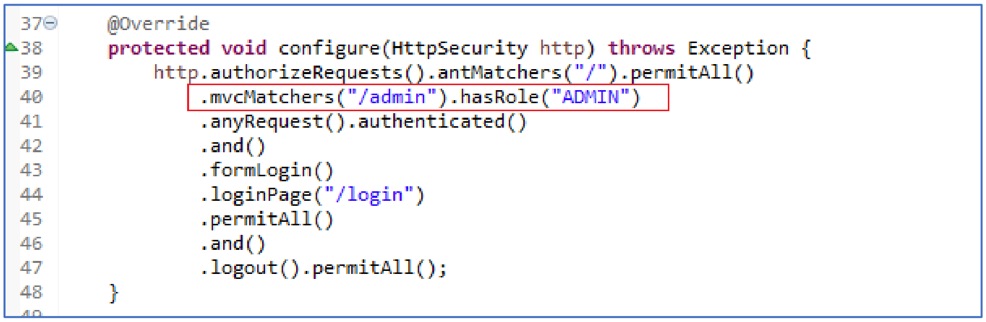
There is cool stuff here, but you will need to log in!

click here to login.

click here if you are and adminisrator

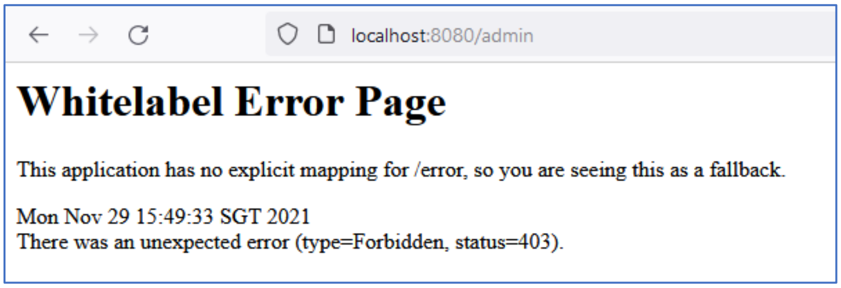
Let's click on the link, which can be accessed only if your administrators. This is to the URL `/admin`. But because this is a secure URL, we are redirected to the login page.

Let's try and log in as a user rather than an admin. So the page that we are trying to access is at the path `/admin`.



```
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests().antMatchers("/").permitAll()
            .mvcMatchers("/admin").hasRole("ADMIN")
            .anyRequest().authenticated()
            .and()
            .formLogin()
            .loginPage("/login")
            .permitAll()
            .and()
            .logout().permitAll();
    }
```

And we are trying to log in as *user*. When you hit the login button, you will see an error.



localhost:8080/admin

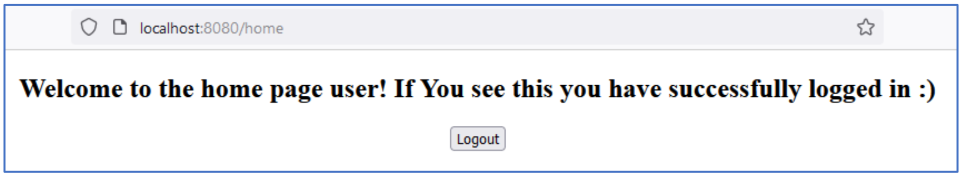
Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Mon Nov 29 15:49:33 SGT 2021
There was an unexpected error (type=Forbidden, status=403).

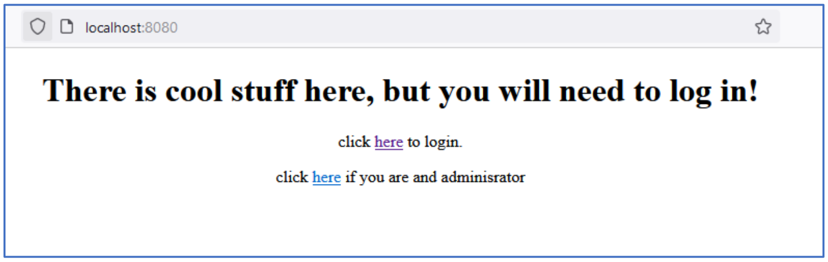
The user isn't the user role but the `/admin` path is only allowed for those who have the `ADMIN` role. This is thanks to our configuration with says

`.mvcMatchers("/admin").hasRole("ADMIN").` We are not allowed to access this page as a user. But if we go to the `/home` page, this page you can access as an ordinary user and you can see the Welcome to the home page message right here.



Welcome to the home page user! If You see this you have successfully logged in :)

Let's log out as the user, and this time we'll log in as an admin. Let's go back to our main welcome page where we have the two links to `/home` and `/admin`. Let's click on the link with says *Click here if you are an administrator*.

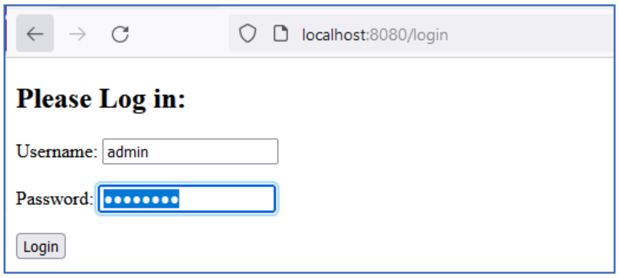


There is cool stuff here, but you will need to log in!

click here to login.

click here if you are and adminisrator

And this time around, we log in as an admin. In our in memory configuration of users we've specified with the role admin.

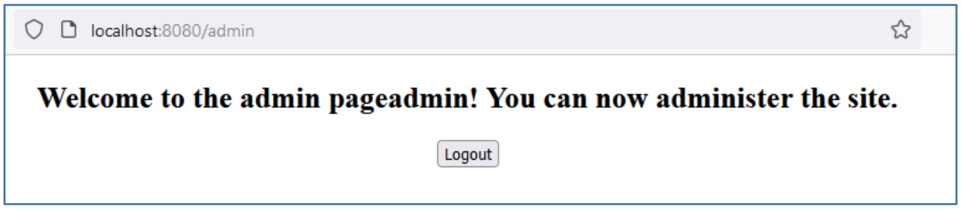


Please Log in:

Username: admin

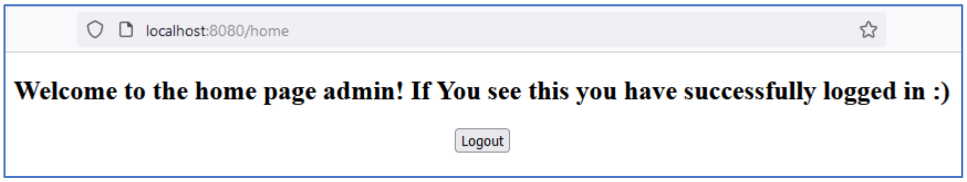
Password: ••••••••

Click on login, and you're immediately taken to the admin page.



Welcome to the admin pageadmin! You can now administer the site.

We've seen that only administrators are allowed access to this page. What about the page`/home` ?



Welcome to the home page admin! If You see this you have successfully logged in :)

both users and admins can access this page, you can see that admin has access to this page as well.

Setting up the User Entity and Repository

In this demo, we are going to bring into a single Spring Boot application, many of the concepts that we've studied in this learning path.

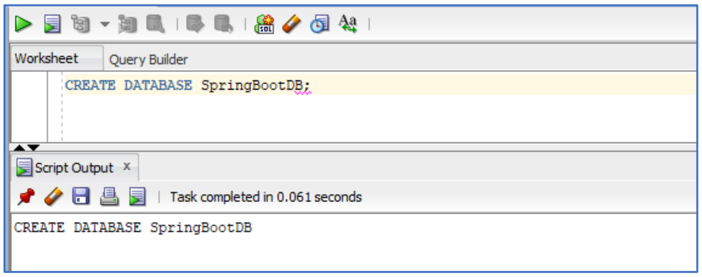
We'll secure the pages of our web app using Spring Security. But this time around, we'll allow any user with an email address to register on our site. Any user with a valid email address will be allowed to register on our site. We'll send that user a confirmation email, which the user can then use to set his or her password. That email will have a special link with a confirmation token. When the user clicks on that link, he or she will be allowed to set the password for their login. All of our registered users will be stored in a MySQL database.

This database will include registered users and users who have set a password as well. Users who have successfully set their password will be allowed to log in. This MySQL database will be accessed using JPA and Hibernate.

As you can see, our login process is fairly involved. This mimics the login process of several applications in the real world.

1. Prepare Database

Let's start off within our MySQL Workbench, where I'm going to create a new database called SpringBootDB, which will hold the user's table, the registered users for our app.

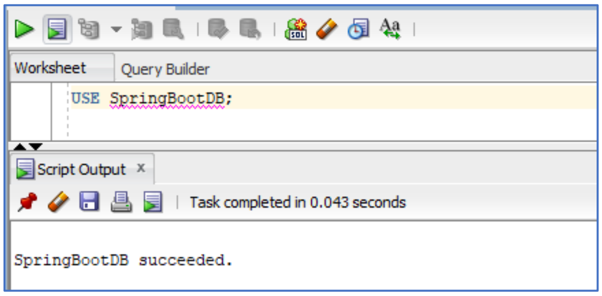


```
CREATE DATABASE SpringBootDB;
```

Task completed in 0.061 seconds

```
CREATE DATABASE SpringBootDB
```

I've used the this command to run for **CREATE DATABASE SpringBootDB**. A new database will be created. Once the database is created, I'm going to use this database. I'm going to create tables within this database, **USE SpringBootDB**.

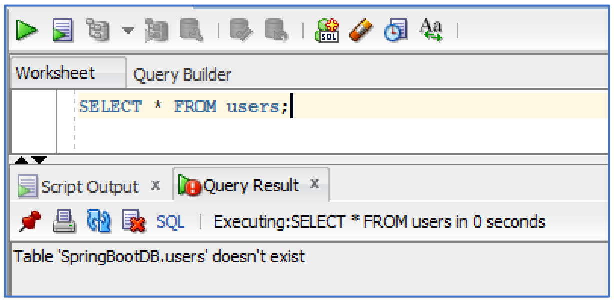


```
USE SpringBootDB;
```

Task completed in 0.043 seconds

```
SpringBootDB succeeded.
```

This SpringBootDB that we've just created is currently empty. There are no tables within it. We can run a simple SELECT * FROM users and you can see that the users table does not exist at this point in time.



```
SELECT * FROM users;
```

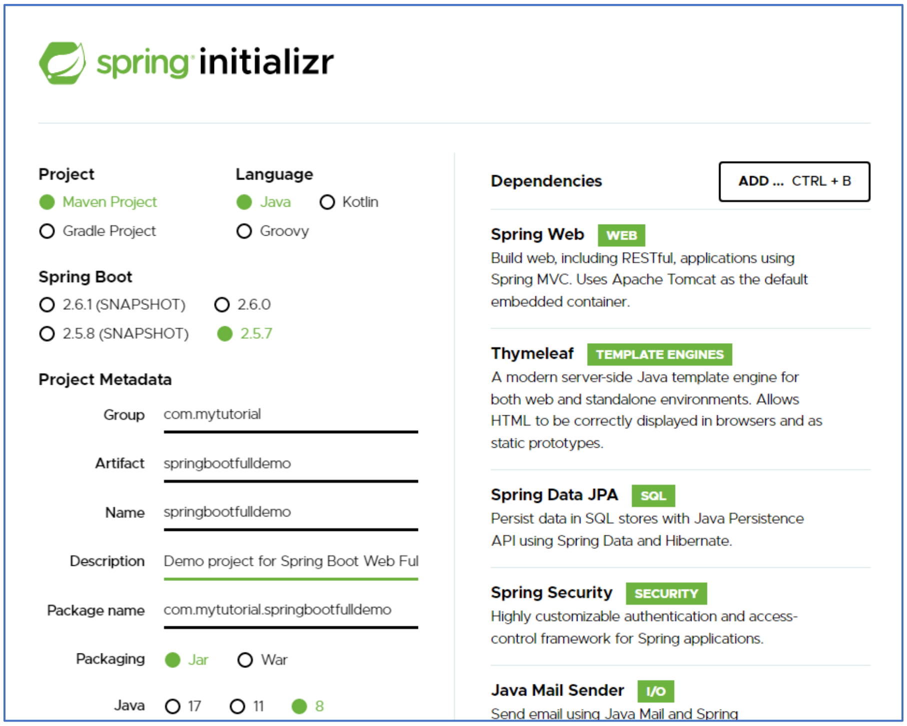
Executing:SELECT * FROM users in 0 seconds

Table 'SpringBootDB.users' doesn't exist

It'll be created when we run our application.

2. Prepare Maven Project

Create Template Maven project from start.spring.io, click generate and extract it into our Eclipse workspace



spring initializr

Project
● Maven Project
○ Gradle Project

Language
● Java ○ Kotlin
○ Groovy

Spring Boot
○ 2.6.1 (SNAPSHOT) ○ 2.6.0
○ 2.5.8 (SNAPSHOT) ● 2.5.7

Project Metadata

Group: com.mytutorial
Artifact: springbootfulldemo
Name: springbootfulldemo
Description: Demo project for Spring Boot Web Ful
Package name: com.mytutorial.springbootfulldemo
Packaging: ● Jar ○ War
Java: ○ 17 ○ 11 ● 8

Dependencies — ADD ... CTRL + B

Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

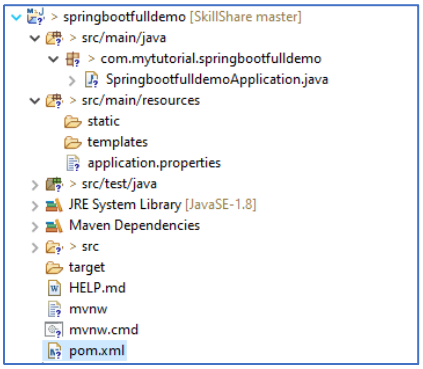
Thymeleaf TEMPLATE ENGINES
A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes.

Spring Data JPA SQL
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

Spring Security SECURITY
Highly customizable authentication and access-control framework for Spring applications.

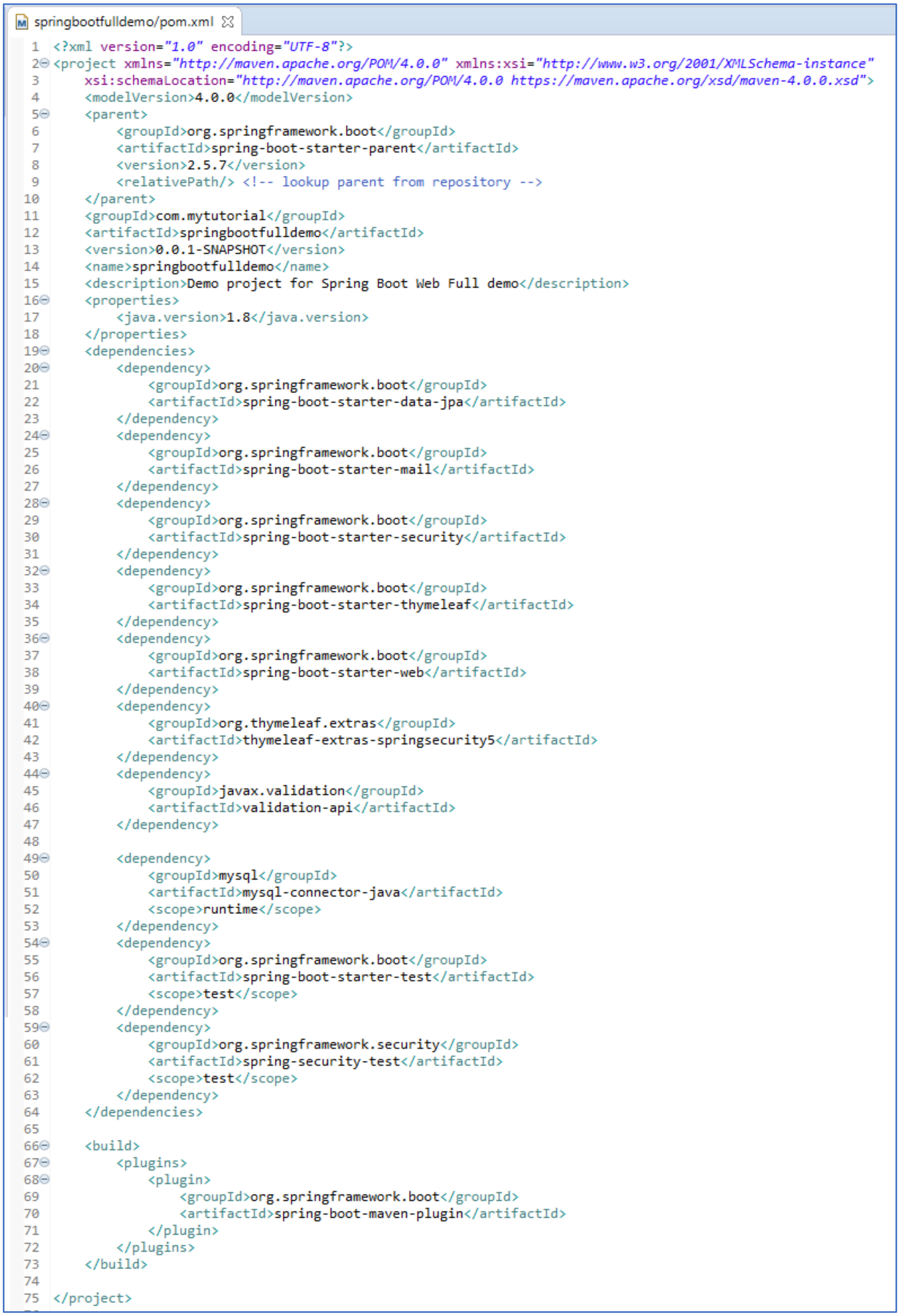
Java Mail Sender I/O
Send email using Java Mail and Spring

Let's now head over to our Eclipse IDE where I've set up the code for this project.



```
springbootfulldemo [SkillShare master]
  src/main/java
    com.mytutorial.springbootfulldemo
      SpringbootfulldemoApplication.java
  src/main/resources
    static
    templates
    application.properties
  src/test/java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
  target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
```

Here is my **pom.xml** file.



```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.5.7</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.mytutorial</groupId>
    <artifactId>springbootfulldemo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>springbootfulldemo</name>
    <description>Demo project for Spring Boot Web Full demo</description>
    <properties>
        <java.version>1.8</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-mail</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-security</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.thymeleaf.extras</groupId>
            <artifactId>thymeleaf-extras-springsecurity5</artifactId>
        </dependency>
        <dependency>
            <groupId>javax.validation</groupId>
            <artifactId>validation-api</artifactId>
        </dependency>

        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

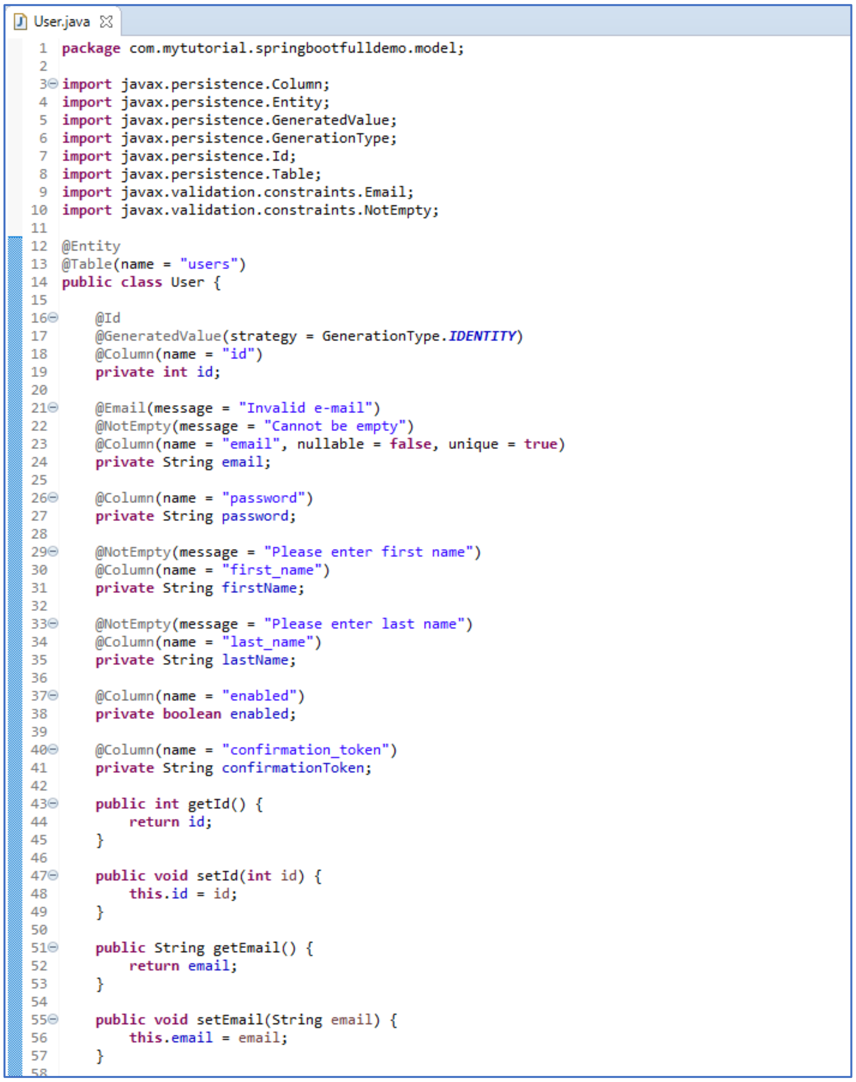
Let's take a look at the starter templates that we'll be using for this project.

- The **spring-boot-starter-data-jpa** template will allow us to use the Java Persistence API and the Hibernate ORM framework to access the underlying MySQL database.
- The **spring-boot-starter-security** dependency will allow us to secure the request made to our web application.
- The **validation-api** dependency will allow us to validate any forms that we use in our app. (*not included yet when generated from start.spring.io, manually add this dependency)
- The **spring-boot-starter-thymeleaf** will allow us to use the thymeleaf template engine.
- The **spring-boot-starter-web** ensures that we can build and run web applications in Spring.
- The **spring-boot-starter-mail** will allow us to use the JavaMailSender API in order to send emails.
- The **mysql-connector-java** dependency will enable the Java Persistence API and the Hibernate ORM framework will be working with an underlying MySQL database

3. Prepare Model Object

Let's explore the various components that we have within this application starting from the model namespace. Within *com.mytutorial.springbootfulldemo.model*, I have a class named User.

User.java



```java
package com.mytutorial.springbootfulldemo.model;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.validation.constraints.Email;
import javax.validation.constraints.NotEmpty;

@Entity
@Table(name = "users")
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private int id;

    @Email(message = "Invalid e-mail")
    @NotEmpty(message = "Cannot be empty")
    @Column(name = "email", nullable = false, unique = true)
    private String email;

    @Column(name = "password")
    private String password;

    @NotEmpty(message = "Please enter first name")
    @Column(name = "first_name")
    private String firstName;

    @NotEmpty(message = "Please enter last name")
    @Column(name = "last_name")
    private String lastName;

    @Column(name = "enabled")
    private boolean enabled;

    @Column(name = "confirmation_token")
    private String confirmationToken;

    public int getId() {
        return id;
    }

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

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
```

Now, this user class has been tagged using the **@Entity** annotation, indicating that this represents a table in our underlying MySQL database.

This class represents a JPA entity, which means objects of this class represent records or rows in the underlying users table. How do we know the table is named users? The @Table annotation has name equal to users. That's the name of our table. The member variables of this class correspond to columns in the underlying users table.

- The first member variable is the ID.

```java
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(name = "id")
```

```
private int id;
```

It corresponds to a column with name id. As you can see from the **@Column** annotation, the **@ID** annotation tells us that this is the primary key of a user entity, the primary key of the underlying users table.
The **@GeneratedValue(strategy = GenerationType.AUTO)** tells us that the primary key is automatically generated by the hibernate object relational mapping framework that connects us to the underlying database table.

- The next column in our database table is the email column.

```
@Email(message = "Invalid e-mail")
@NotEmpty(message = "Cannot be empty")
@Column(name = "email", nullable = false, unique = true)
private String email;
```

We will tag this as **@Column**. The name of the column is email. This email column cannot be null. And the email has to be **unique, nullable = false, unique = true**. These are the constraints on this column.
The **@Email** annotation and the **@NotEmpty** annotation are part of the **javax validation API**.
This basically says that this field should contain a valid email address, and this field cannot be empty. The next member variable in this class is for the password of this user.

- Now the password corresponds to the password column in the underlying table.

```
@Column(name = "password")
private String password;
```

We'll only store an encrypted form of the password in our table.

- The field after that is for the first name of the user.

```
@NotEmpty(message = "Please enter first name")
@Column(name = "first_name")
private String firstName;
```

The column that it corresponds to is called first_name. And we've tagged it using the **@NotEmpty** validation annotation indicating that the first name cannot be left empty. **@NotEmpty** is a part of the **javax validation API** for form validation.

- The last name member variable corresponds to the column last_name.

```
@NotEmpty(message = "Please enter last name")
@Column(name = "last_name")
private String lastName;
```

We have the **@NotEmpty** validation set on it indicating last name cannot be empty.

- We then have a column called enable, indicating whether the user has set a password and is active in our system.

  ```
  @Column(name = "enabled")
  private boolean enabled;
  ```

  This corresponds to the enabled column in the underlying table.

- And finally, we have a String confirmationToken.

  ```
  @Column(name = "confirmation_token")
  private String confirmationToken;
  ```

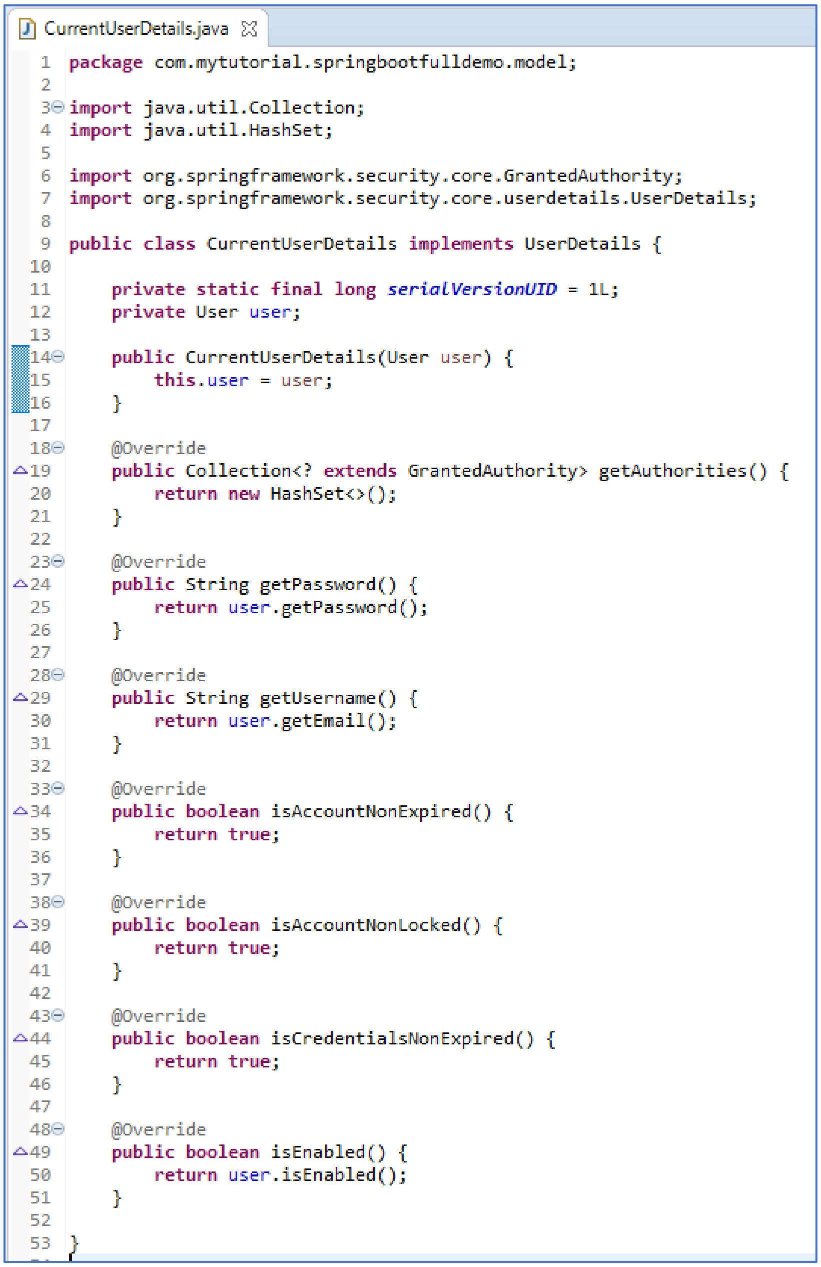
  For every user registered in our system, we'll generate a confirmation token that is unique to that user. This confirmation token will be part of the link sent to the user's email address, allowing the user to set the password. This confirmation token will be stored along with the user's record in the table, in a column called confirmation token.

If you look at the rest of the code in this user class, all you see are getters and setters for the individual member variables. These getters and setters are used by Spring in order to set values for the different fields, and access values from the different fields.

4. Create User Details Wrapper Class reference to Model Object

Every object of the user class corresponds to one row or one record in the underlying *users* table. This user object maps to the underlying database table where we store users. Spring Security cannot work directly with this user object. It needs a wrapper.

And the wrapper is **CurrentUserDetails**, which implements the UserDetails interface.



```java
package com.mytutorial.springbootfulldemo.model;

import java.util.Collection;
import java.util.HashSet;

import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;

public class CurrentUserDetails implements UserDetails {

    private static final long serialVersionUID = 1L;
    private User user;

    public CurrentUserDetails(User user) {
        this.user = user;
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return new HashSet<>();
    }

    @Override
    public String getPassword() {
        return user.getPassword();
    }

    @Override
    public String getUsername() {
        return user.getEmail();
    }

    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    @Override
    public boolean isAccountNonLocked() {
        return true;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    @Override
    public boolean isEnabled() {
        return user.isEnabled();
    }

}
```

Any class which implements the user details information is what Spring Security uses to get core user information.

- Now, this class is serializable. That's why we specify a ***serialVersionUID***.

- Within this class, we store a reference to the *user* object. You can see a member variable and the **constructor** for this class, that takes an input argument, a reference to the user object part of our model.

```
private User user;
public CurrentUserDetails(User user) {
	this.user = user;
}
```

- The rest of the code that you see here in this class are simply the implementations for the methods in the user details interface. Get authorities returns a list of authorities associated with this user. This cannot be null. We've just set it to an empty set.

```
@Override
public Collection<? extends GrantedAuthority> getAuthorities() {
	return new HashSet<>();
}
```

- We delegate getPassword and getUsername to the underlying user object.

```
@Override
public String getPassword() {
	return user.getPassword();
}
```

- The user name for every user is the email address of the user.

```
@Override
public String getUsername() {
	return user.getEmail();
}
```

- And we return *true* for **isAccountNonExpired** and **isAccountNonLocked**. You can of course implement these if you want to.

- IsCredentialsNonExpired, we return true there as well.

- And isEnabled, we delegate to the underlying user object. We return user.getEnabled.

```
@Override
public boolean isEnabled() {
	return user.isEnabled();
}
```

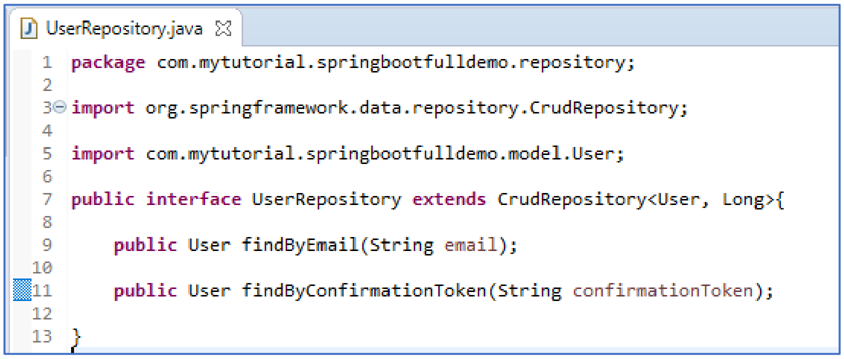
  A user is enabled when he or she has clicked on the confirmation link and set a password.

We've already set up the user class for the user objects that correspond to the user records in our underlying database table.

5. Prepare Entity Repository Interface For CRUD Operation

We now need a way to perform, create, read, update and delete operations on these users. We set up a user repository to perform these operations within the repository namespace. Now, you might look at this class and say, where are the CRUD operations implemented? Well, that's where the magic of Spring Boot and Spring Data comes in. This interface basically specifies our entire CRUD service. And Spring along with Spring Data does this magically.

Let's take things one step at a time. And notice that we have an interface here called **UserRepository**, which is tagged using the **@Repository** annotation.



```
package com.mytutorial.springbootfulldemo.repository;

import org.springframework.data.repository.CrudRepository;

import com.mytutorial.springbootfulldemo.model.User;

public interface UserRepository extends CrudRepository<User, Long>{

    public User findByEmail(String email);

    public User findByConfirmationToken(String confirmationToken);

}
```

The use of the **@Repository** annotation tells Spring that an instance of this interface is essentially a Spring component, which ***performs data access operations***. It's responsible for reading from the database that we have configured and maybe writing to the database as well.

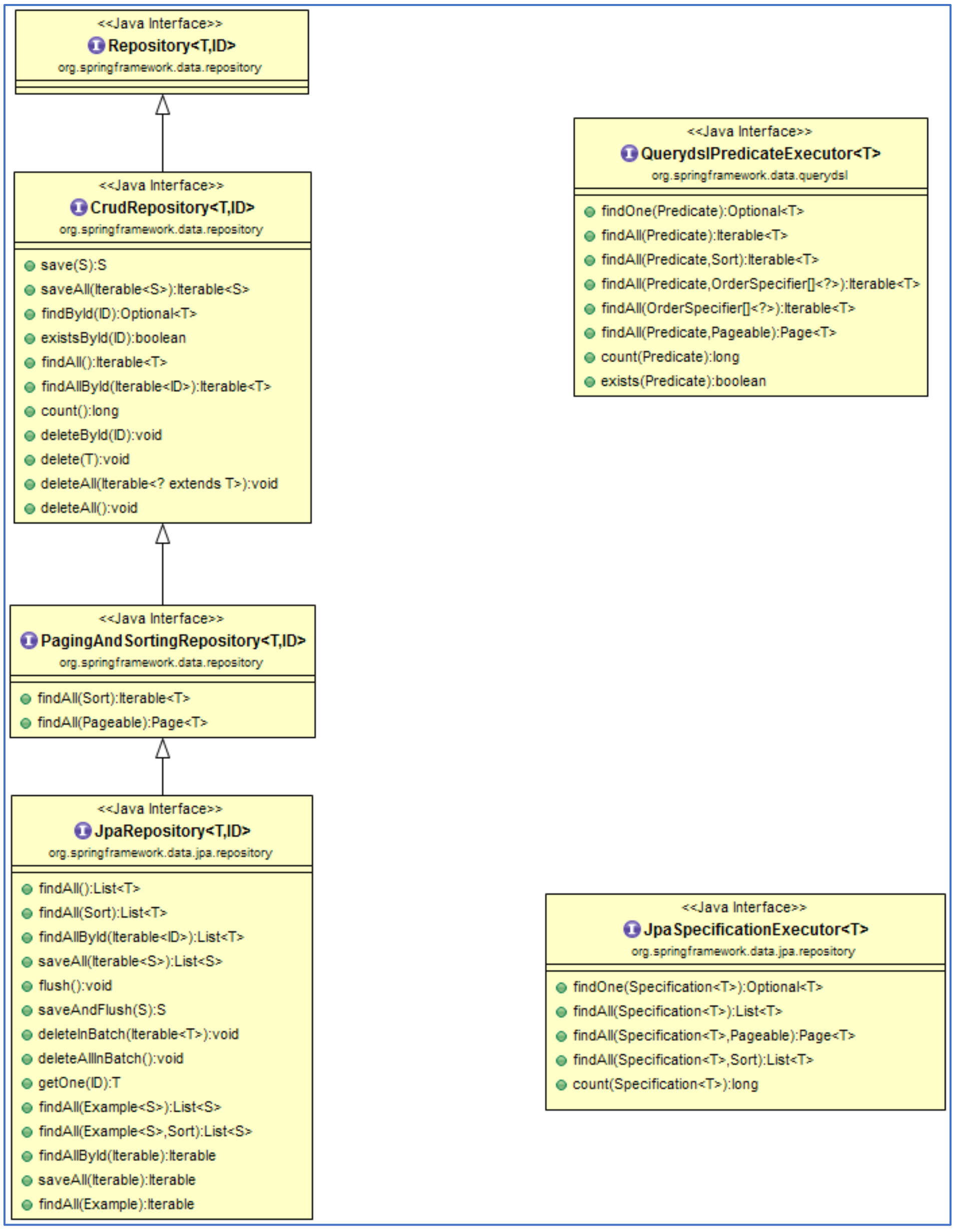
Any exceptions thrown by objects, which are annotated using **@Repository** are converted to data access exceptions in Spring.

The next thing to note here is that this UserRepository interface extends the interface **CrudRepository**. This is a generic interface. The generic types we've specified are *User*, **Long**. This basically tells Spring that our user repository performs, create, read, update, and delete operations on user entities. And the **primary key** for these user entities are of type **long**.

"This **CRUD** repository interface is a part of spring data. This is what allows us to perform CRUD or **create, read, update, and delete** operations on our underlying objects."

Now, you might say this is an interface. Where is the implementation? And this is where Spring Data's magic comes in. When you specify this interface, **Spring will automatically take care of instantiating a class that implements this interface** and allows you to perform operations on the underlying user objects within your users table.

Some examples of methods, which are part of the CRUD repository interface are these, ***findbyId***, **findAll**, **save**, **saveAll**, **delete**, **deleteAll**, **count**, and so on. Implementations for these methods will be automatically provided by Spring Data.


<<Java Interface>>
Repository<T,ID>
org.springframework.data.repository
<<Java Interface>>
QuerydslPredicateExecutor<T>
org.springframework.data.querydsl
findOne(Predicate):Optional<T>
findAll(Predicate):Iterable<T>
findAll(Predicate,Sort):Iterable<T>
findAll(Predicate,OrderSpecifier[]<?>):Iterable<T>
findAll(OrderSpecifier[]<?>):Iterable<T>
findAll(Predicate,Pageable):Page<T>
count(Predicate):long
exists(Predicate):boolean
<<Java Interface>>
CrudRepository<T,ID>
org.springframework.data.repository
save(S):S
saveAll(Iterable<S>):Iterable<S>
findById(ID):Optional<T>
existsById(ID):boolean
findAll():Iterable<T>
findAllById(Iterable<ID>):Iterable<T>
count():long
deleteById(ID):void
delete(T):void
deleteAll(Iterable<? extends T>):void
deleteAll():void
<<Java Interface>>
PagingAndSortingRepository<T,ID>
org.springframework.data.repository
findAll(Sort):Iterable<T>
findAll(Pageable):Page<T>
<<Java Interface>>
JpaRepository<T,ID>
org.springframework.data.jpa.repository
findAll():List<T>
findAll(Sort):List<T>
findAllById(Iterable<ID>):List<T>
saveAll(Iterable<S>):List<S>
flush():void
saveAndFlush(S):S
deleteInBatch(Iterable<T>):void
deleteAllInBatch():void
getOne(ID):T
findAll(Example<S>):List<S>
findAll(Example<S>,Sort):List<S>
findAllById(Iterable):Iterable
saveAll(Iterable):Iterable
findAll(Example):Iterable
<<Java Interface>>
JpaSpecificationExecutor<T>
org.springframework.data.jpa.repository
findOne(Specification<T>):Optional<T>
findAll(Specification<T>):List<T>
findAll(Specification<T>,Pageable):Page<T>
findAll(Specification<T>,Sort):List<T>
count(Specification<T>):long

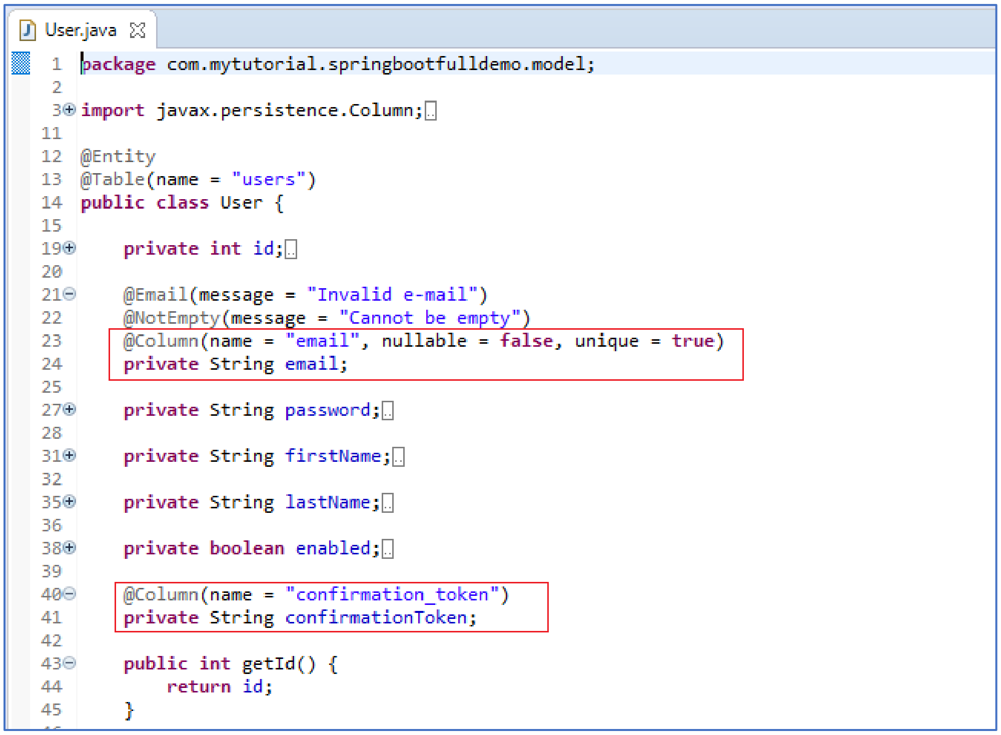
JpaRepository Interface

JpaRepository is JPA specific extension of Repository. It contains the full API of CrudRepository and PagingAndSortingRepository. So it contains API for basic CRUD operations and also API for pagination and sorting.

In addition to these CRUD methods, I want my user repository interface to be able to ***findByEmail***, that is retrieve users by email and retrieve users by confirmation token, ***findByConfirmationToken***.

I just need to specify these methods here. And Spring Data will automatically generate sensibly implementations for these. The names of these methods have been deliberately chosen. *Email* and *confirmationToken* must map to columns in the underlying database table.

User.java



```
package com.mytutorial.springbootfulldemo.model;

import javax.persistence.Column;

@Entity
@Table(name = "users")
public class User {

    private int id;

    @Email(message = "Invalid e-mail")
    @NotEmpty(message = "Cannot be empty")
    @Column(name = "email", nullable = false, unique = true)
    private String email;

    private String password;

    private String firstName;

    private String lastName;

    private boolean enabled;

    @Column(name = "confirmation_token")
    private String confirmationToken;

    public int getId() {
        return id;
    }
```

If you change the names of these interface methods, you will find that things won't work as you would expect.

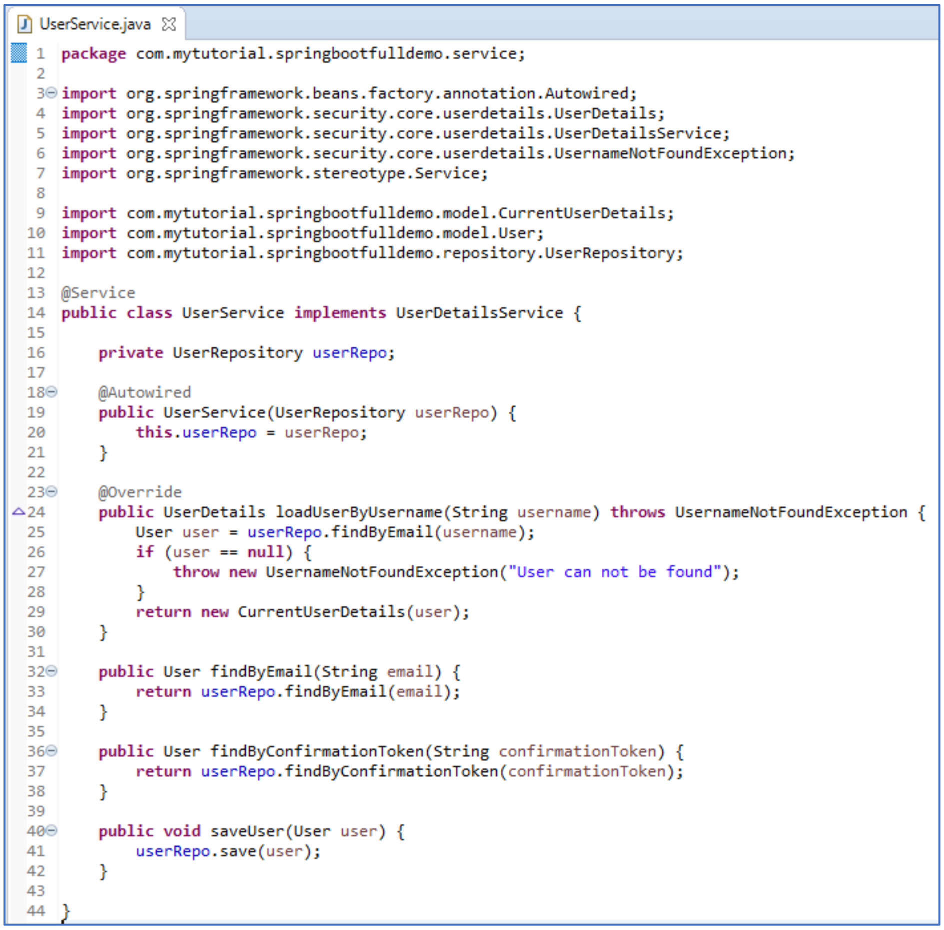
Setting up Services and Controllers

At this point we have the model and the basic data access layer of our app setup. We still need to set up the service layer where the business logic for our application will reside. The service layer will expose methods allowing us to register a new user, send email to a user and authenticate an already registered user.

6. Prepare the Service Class

Let's start off by looking at the user service class which implements the user details service. We need this class to implement the user detail service so that spring security can then use an instance of this class to authenticate logged-in users.

UserService.java



```java
package com.mytutorial.springbootfulldemo.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

import com.mytutorial.springbootfulldemo.model.CurrentUserDetails;
import com.mytutorial.springbootfulldemo.model.User;
import com.mytutorial.springbootfulldemo.repository.UserRepository;

@Service
public class UserService implements UserDetailsService {

    private UserRepository userRepo;

    @Autowired
    public UserService(UserRepository userRepo) {
        this.userRepo = userRepo;
    }

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        User user = userRepo.findByEmail(username);
        if (user == null) {
            throw new UsernameNotFoundException("User can not be found");
        }
        return new CurrentUserDetails(user);
    }

    public User findByEmail(String email) {
        return userRepo.findByEmail(email);
    }

    public User findByConfirmationToken(String confirmationToken) {
        return userRepo.findByConfirmationToken(confirmationToken);
    }

    public void saveUser(User user) {
        userRepo.save(user);
    }

}
```

Spring security will use an implementation of the user detail service interface to retrieve user related data in order to authenticate users.

- Notice that this class is tagged using the **@Service** annotation. This basically tells spring that this is a spring managed component. **@Service** annotation also signifies intent, indicating that this code will contain business logic and as a part of the service layer.

- The UserService has a reference to the *UserRepository* This UserRepository will be injected directly by spring into this user service object because you specified the **@Autowired** annotation on the **constructor** for the *UserService*.

  ```
  private UserRepository userRepo;
  @Autowired
  public UserService(UserRepository userRepo) {
          this.userRepo = userRepo;
  }
  ```

  And the constructor accepts the UserRepository as an input argument.

- Within this user service class, the first is *findByEmail*, which accepts as an input argument, the email address of the user, and then looks up the user using the user repository.

  ```
  public User findByEmail(String email) {
          return userRepo.findByEmail(email);
  }
  ```

  We simply delegate to *userRepo.findByEmail*.

- Similarly, we have *findByConfirmationToken*. The input argument here is the confirmation token associated with a user.

  ```
  public User findByConfirmationToken(String confirmationToken) {
          return userRepo.findByConfirmationToken(confirmationToken);
  }
  ```

  We delegate findByConfirmationToken once again to the userRepository.

- We expose a saveUser method here.

  ```
  public void saveUser(User user) {
          userRepo.save(user);
  }
  ```

  This simply delegates to *userRepository.save*.

- The user details service interface used by *Spring Security* requires that we implement this method here ***loadUserByUsername***.

  ```
  @Override
  public UserDetails loadUserByUsername(String username)
  throws UsernameNotFoundException {
          User user = userRepo.findByEmail(username);
          if (user == null) {
                  throw new UsernameNotFoundException
                              ("User can not be found");
          }
          return new CurrentUserDetails(user);
  }
  ```

  The input argument here is the *username* for a particular user. The user name in our case we know is the email address. We simply use *userRepo.findByEmail* to get the user corresponding to the email address.

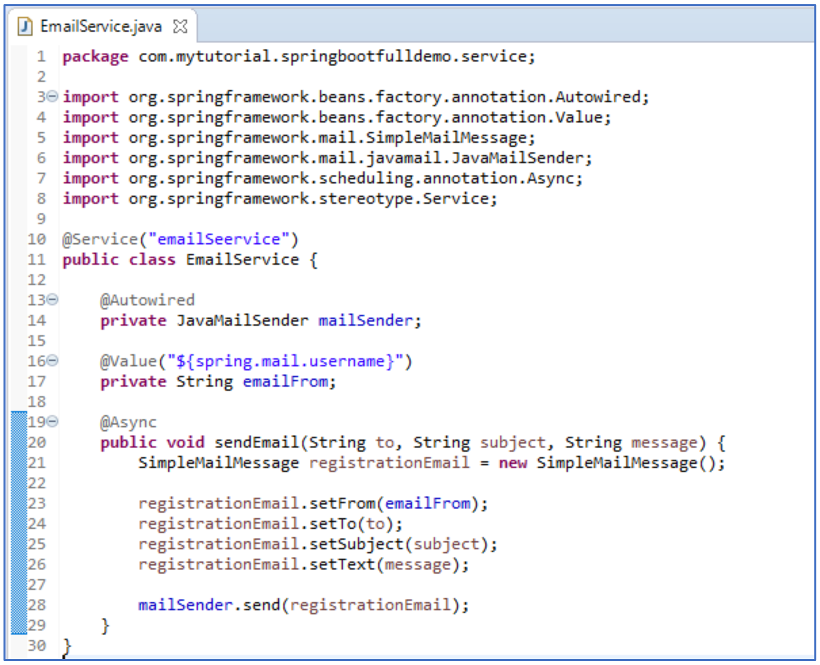
  If user is equal to null, we throw an exception. There is a specific exception available ***UsernameNotFoundException***, which is what we throw.

  If this is a valid user, we wrapped the user in the ***CurrentUserDetails*** class, which implements the user details interface that is the return type of this method.

The user service thus allows us to retrieve and work with the users in our system.

Let's take a look at another service class that I have implemented here.

EmailService.java



```java
package com.mytutorial.springbootfulldemo.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.mail.SimpleMailMessage;
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.scheduling.annotation.Async;
import org.springframework.stereotype.Service;

@Service("emailSeervice")
public class EmailService {

    @Autowired
    private JavaMailSender mailSender;

    @Value("${spring.mail.username}")
    private String emailFrom;

    @Async
    public void sendEmail(String to, String subject, String message) {
        SimpleMailMessage registrationEmail = new SimpleMailMessage();

        registrationEmail.setFrom(emailFrom);
        registrationEmail.setTo(to);
        registrationEmail.setSubject(subject);
        registrationEmail.setText(message);

        mailSender.send(registrationEmail);
    }
}
```

This is the EmailService. This is what we'll use to send an email to the user. This email will contain a link that will allow that user to set a password for his account.

- The EmailService is tagged using the **@Service** annotation indicating that it contains business logic.

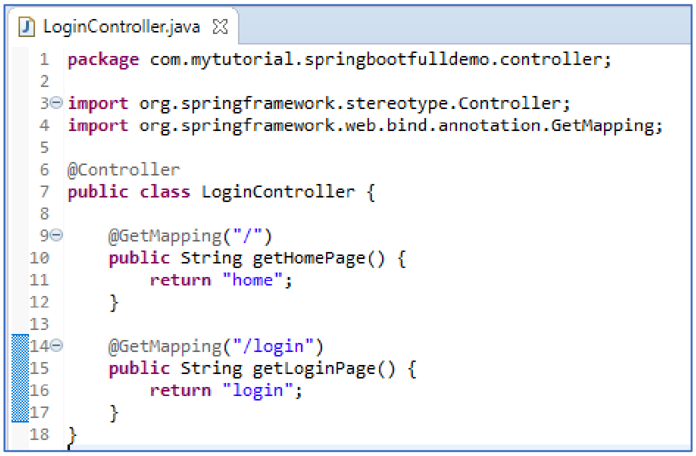
- We inject the JavaMailSender into this email service using **@Autowire** on the mailSender member variable.

- This exposes a single method called *sendEmail*. The input arguments are the two field, the subject and the message.
  We instantiate a SimpleMailMessage, set all of these fields, and then use mailSender.send to send this registration email.

7. Preparing the Controller Class

Next, let's take a look at our *Login* controller where we specify the handler mappings for the incoming web request.

LoginController.java



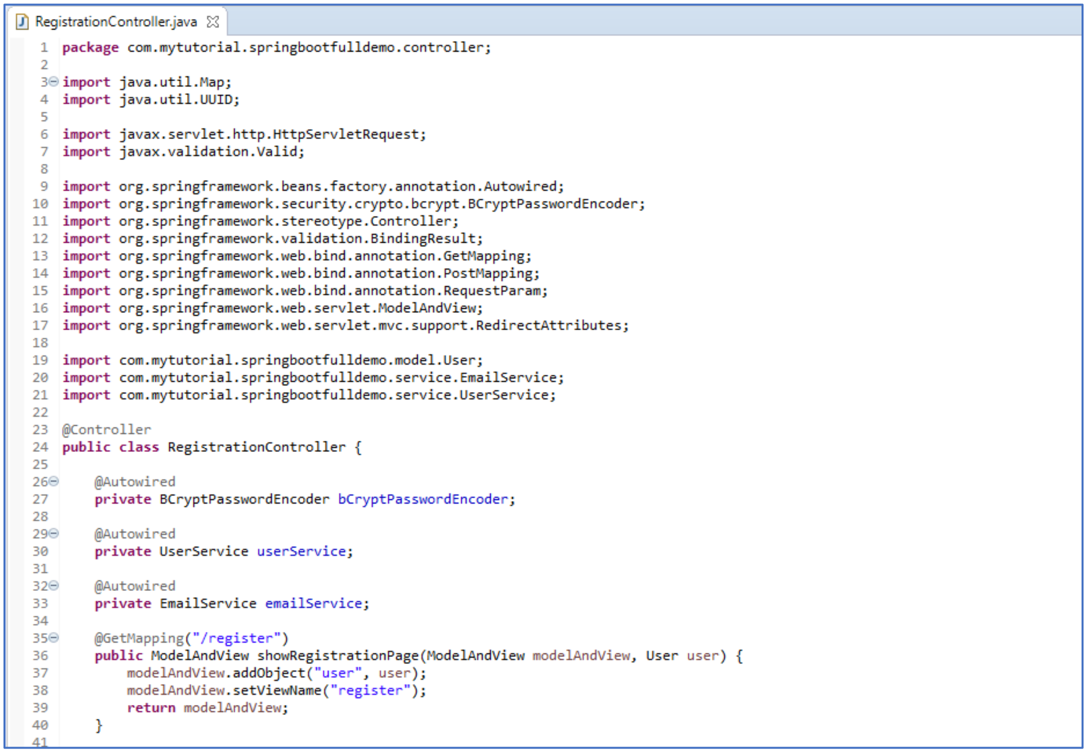
LoginController.java

```java
package com.mytutorial.springbootfulldemo.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class LoginController {

    @GetMapping("/")
    public String getHomePage() {
        return "home";
    }

    @GetMapping("/login")
    public String getLoginPage() {
        return "login";
    }
}
```

- We have **@GetMapping** for the root of our application that is a `"/"`, which renders the homepage.
- We have a **@GetMapping** for the login page at `"/login"`, which renders the login page.

These are the only two mappings that we configure in the LoginController.
The remaining part mappings will configure in the RegistrationController. Just a heads up that the *RegistrationController* contains some fairly complex code so make sure you follow along closely.

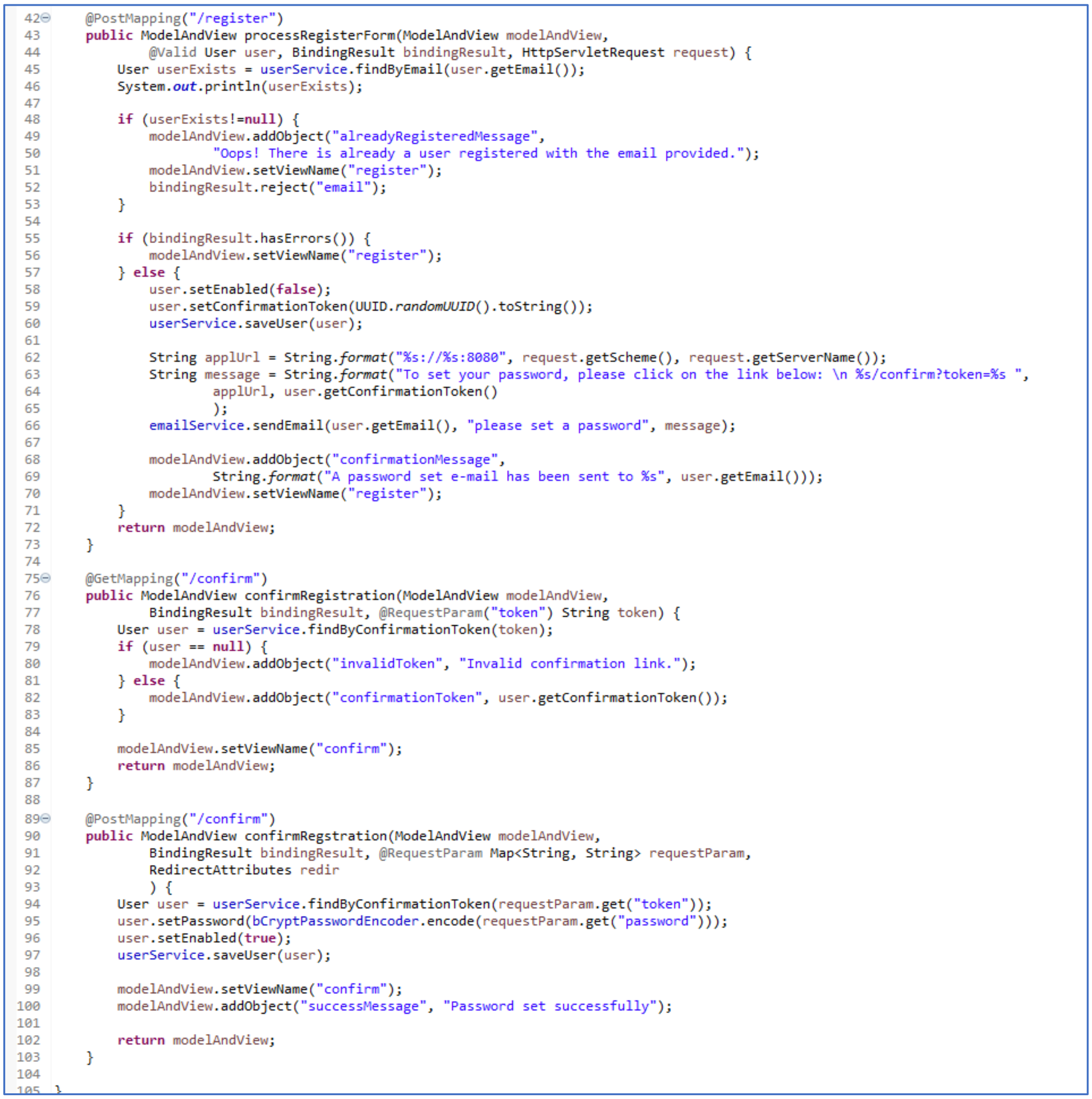
RegistrationController.java



RegistrationController.java

```java
package com.mytutorial.springbootfulldemo.controller;

import java.util.Map;
import java.util.UUID;

import javax.servlet.http.HttpServletRequest;
import javax.validation.Valid;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.stereotype.Controller;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;

import com.mytutorial.springbootfulldemo.model.User;
import com.mytutorial.springbootfulldemo.service.EmailService;
import com.mytutorial.springbootfulldemo.service.UserService;

@Controller
public class RegistrationController {

    @Autowired
    private BCryptPasswordEncoder bCryptPasswordEncoder;

    @Autowired
    private UserService userService;

    @Autowired
    private EmailService emailService;

    @GetMapping("/register")
    public ModelAndView showRegistrationPage(ModelAndView modelAndView, User user) {
        modelAndView.addObject("user", user);
        modelAndView.setViewName("register");
        return modelAndView;
    }

```

RegistrationController.java (continue…)



```java
    @PostMapping("/register")
    public ModelAndView processRegisterForm(ModelAndView modelAndView,
            @Valid User user, BindingResult bindingResult, HttpServletRequest request) {
        User userExists = userService.findByEmail(user.getEmail());
        System.out.println(userExists);

        if (userExists!=null) {
            modelAndView.addObject("alreadyRegisteredMessage",
                    "Oops! There is already a user registered with the email provided.");
            modelAndView.setViewName("register");
            bindingResult.reject("email");
        }

        if (bindingResult.hasErrors()) {
            modelAndView.setViewName("register");
        } else {
            user.setEnabled(false);
            user.setConfirmationToken(UUID.randomUUID().toString());
            userService.saveUser(user);

            String applUrl = String.format("%s://%s:8080", request.getScheme(), request.getServerName());
            String message = String.format("To set your password, please click on the link below: \n %s/confirm?token=%s ",
                    applUrl, user.getConfirmationToken()
                    );
            emailService.sendEmail(user.getEmail(), "please set a password", message);

            modelAndView.addObject("confirmationMessage",
                    String.format("A password set e-mail has been sent to %s", user.getEmail()));
            modelAndView.setViewName("register");
        }
        return modelAndView;
    }

    @GetMapping("/confirm")
    public ModelAndView confirmRegistration(ModelAndView modelAndView,
            BindingResult bindingResult, @RequestParam("token") String token) {
        User user = userService.findByConfirmationToken(token);
        if (user == null) {
            modelAndView.addObject("invalidToken", "Invalid confirmation link.");
        } else {
            modelAndView.addObject("confirmationToken", user.getConfirmationToken());
        }

        modelAndView.setViewName("confirm");
        return modelAndView;
    }

    @PostMapping("/confirm")
    public ModelAndView confirmRegstration(ModelAndView modelAndView,
            BindingResult bindingResult, @RequestParam Map<String, String> requestParam,
            RedirectAttributes redir
            ) {
        User user = userService.findByConfirmationToken(requestParam.get("token"));
        user.setPassword(bCryptPasswordEncoder.encode(requestParam.get("password")));
        user.setEnabled(true);
        userService.saveUser(user);

        modelAndView.setViewName("confirm");
        modelAndView.addObject("successMessage", "Password set successfully");

        return modelAndView;
    }

}
```

- Notice the services and the objects that we inject into the RegistrationController. We have the **BCryptPasswordEncoder** to encode passwords that we received. Remember, passwords should never be stored in plain text in our database. They should always be in an encrypted form.

```java
@Autowired
private BCryptPasswordEncoder bCryptPasswordEncoder;
```

- We also inject the userService as well as the emailService.

```java
@Autowired
private UserService userService;
@Autowired
private EmailService emailService;
```

- The first method is a handler mapping for the registration page for our application. It has an **@GetMapping** annotation for the path `"/register"`.

  ```
  @GetMapping("/register")
  public ModelAndView showRegistrationPage(
  ModelAndView modelAndView, User user) {
  ```

  We specify two input arguments here, the `ModelAndView` and a *`User`* object. A new *`User`* object will be constructed and injected into this method. This is the *`User`* object that will be bound to the form that we display to the *`User`* to register that user. Within this method, we add this user object using the user parameter on the **modelAndView**.

  We set the view name to `"register"` and return this modelAndView. This will render the *register.html* page and the form within the registration page will be bound to our user object.

- The next method called *processRegistrationForm* has been annotated using an **@PostMapping**. This method responds to post requests made to the `"/register"` path.

  ```
  @PostMapping("/register")
  public ModelAndView processRegisterForm(ModelAndView modelAndView,
              @Valid User user, BindingResult bindingResult,
              HttpServletRequest request) {
  ```

  The input argument here is a **modelAndView**, a **Valid** User object, the **BindingResult** and the **HttpServletRequest**. This is the method that is invoked when the user hits Submit on the registration form.
  Notice that we have the **@Valid** annotation on the `user` object that is input. This ensures that the `user` Object will be validated based on the validation API annotations that we've used on this object. If there are any errors in the validation of this `user` Object that is passed from the client form to our handler method here, that will be available in the binding result.

  Remember, the **BindingResult** input argument should always be after the **ModelUserObject**.

  Let's take a look at the steps involved in registering a new user. We'll use the `userService`, to see if another `user` with the same email address already exists. If a `userExists`, that is if `userExists` is not equal to **`null`**, in that case we need to return an *error message* to the client.

  ```
  User userExists = userService.findByEmail(user.getEmail());
  if (userExists!=null) {
        modelAndView.addObject("alreadyRegisteredMessage",
        "Oops! There is already a user registered
              with the email provided.");
        modelAndView.setViewName("register");
        bindingResult.reject("email");
  }
  ```

We can't have two users with the same email address to the **modelAndView** add a value for the `alreadyRegisteredMessage` will say `"Oops! There is already a user registered with the email provided."`. And we render the *register.html* page. Once again, we'll set the view name to `"register"` will reject the email that the `user` has specified using `bindingResult.reject`. This Code within the if block here ensures that every user in our system, is registered using a unique email address.

We then check to see if the `user` Object contains valid fields. If binding result has errors, we send the user back to the registration page and display those errors.

```
if (bindingResult.hasErrors()) {
        modelAndView.setViewName("register");
```

Our execution will get to the else block here.

```
} else {
        user.setEnabled(false);
        user.setConfirmationToken(UUID.randomUUID().toString());
```

If you have a unique email address and our submitted form has no errors. We'll register a new `user`, but we'll set the *enabled* field for the `user` to **`false`**. That's because the `user` hasn't specified a password yet.

We'll also generate a new confirmation token for the user using **UUID.randomUUID.toString**. *UUID is simply a utility available in Java that allows the generation of tokens*. UUID stands for Universally Unique Identifier.

```
        userService.saveUser(user);
```

Once we set the enable field and the confirmation token for this user, we'll call the userService and save this user in the underlying database.

We then generate a URL that the user can click on to set his or her password. This is our `applUrl`. We access the current request get the scheme whether it's `"http://"` or `"https://"`, we then get the current server name add the `":8080"` suffix that is the port for our current application.

```
        String applUrl = String.format("%s://%s:8080",
        request.getScheme(), request.getServerName());
```

We then construct the message for the email address. The `message` will say, `"To set your password, please click on the link below:"`. The link below will include the `applUrl`, `"/confirm"` and a request parameter, that is the *ConfirmationToken*.

```
        String message = String.format("To set your password, please
        click on the link below: \n %s/confirm?token=%s ",
        applUrl, user.getConfirmationToken());
```

Will then use the emailService to send an email to the email address specified by the user to register himself or herself.

```
            emailService.sendEmail(user.getEmail(),
            "please set a password", message);

            modelAndView.addObject("confirmationMessage",
            String.format("A password set e-mail has been sent to %s",
            user.getEmail()));

            modelAndView.setViewName("register");
    }
```

So user.getEmail() will give us the email address of the user. The subject will be please "please set a password". And we'll pass in the message that we constructed which contains a link along with a confirmation token.

```
message = "To set your password, please click on the link below: \n applUrl
/confirm?token=user.getConfirmationToken() "
```

notice that the link maps to the **/confirm** page. Now to the modelAndView, we'll add this confirmation message "A password set e-mail has been sent to [email-address-of-the-user]". We'll then render the *register.html* page once again.

- When the user opens up the email address used to register on our site and then clicks on that link, he'll be mapped to this path /confirm. This here is the handler method for confirming a user's registration.

```
    @GetMapping("/confirm")
    public ModelAndView confirmRegistration(
            ModelAndView modelAndView,
            BindingResult bindingResult,
            @RequestParam("token") String token) {
```

Notice that the input arguments here are the ModelAndView and the confirmation token that is injected as a request parameter. Remember the link contains the confirmation token as **@RequestParam** that is what is injected in here.

```
    User user = userService.findByConfirmationToken(token);
```

We'll then use the userService to look up the Registered User by confirmation token. We call userService.findByConfirmationToken(token), if the user is null that is there is no user with this confirmation token. We'll essentially set a message for invalid token saying "Invalid confirmation link.".

```
    if (user == null) {
            modelAndView.addObject("invalidToken",
            "Invalid confirmation link.");
```

Otherwise, we'll add an object for the confirmationToken key and pass back the confirmation token and we'll render the confirm view.

```
} else {
      modelAndView.addObject("confirmationToken",
      user.getConfirmationToken());
}

modelAndView.setViewName("confirm");
```

This *confirm.html* view that we render will allow the user to set a password for himself or herself.

- And once the user hits submit on the newly set password, that will be a post request to this "/confirm" path handled by the confirm registration method annotated using @PostMapping("/confirm").

```
@PostMapping("/confirm")
public ModelAndView confirmRegstration(
             ModelAndView modelAndView,
             BindingResult bindingResult,
             @RequestParam Map<String, String> requestParam,
             RedirectAttributes redir
) {
```

We'll use the @RequestParam Map<String, String> requestParam map to look up the confirmation token using requestParam.get("token"). And we then invoke userService.findByConfirmationToken() to retrieve the user corresponding to this confirmation token.

```
userService.findByConfirmationToken(requestParam.get("token"));
```

At this point, we know that a user corresponding to this confirmation token exists. We call user.setPassword(). Use the bCryptPasswordEncoder.encode() to encode the password first. The password is available in the requestParam.

```
user.setPassword(
      bCryptPasswordEncoder.encode(
             requestParam.get("password")
  ));
```

We also set the user as enable setEnabled equal to true and then call userService.saveUser to update and save this enabled user along with a password. And finally, having set the password for the user successfully,

```
user.setEnabled(true);
userService.saveUser(user);
modelAndView.setViewName("confirm")
modelAndView.addObject("successMessage", "Password set successfully");
```

we'll render the Confirm page once again, but the successMessage : "Password set successfully".

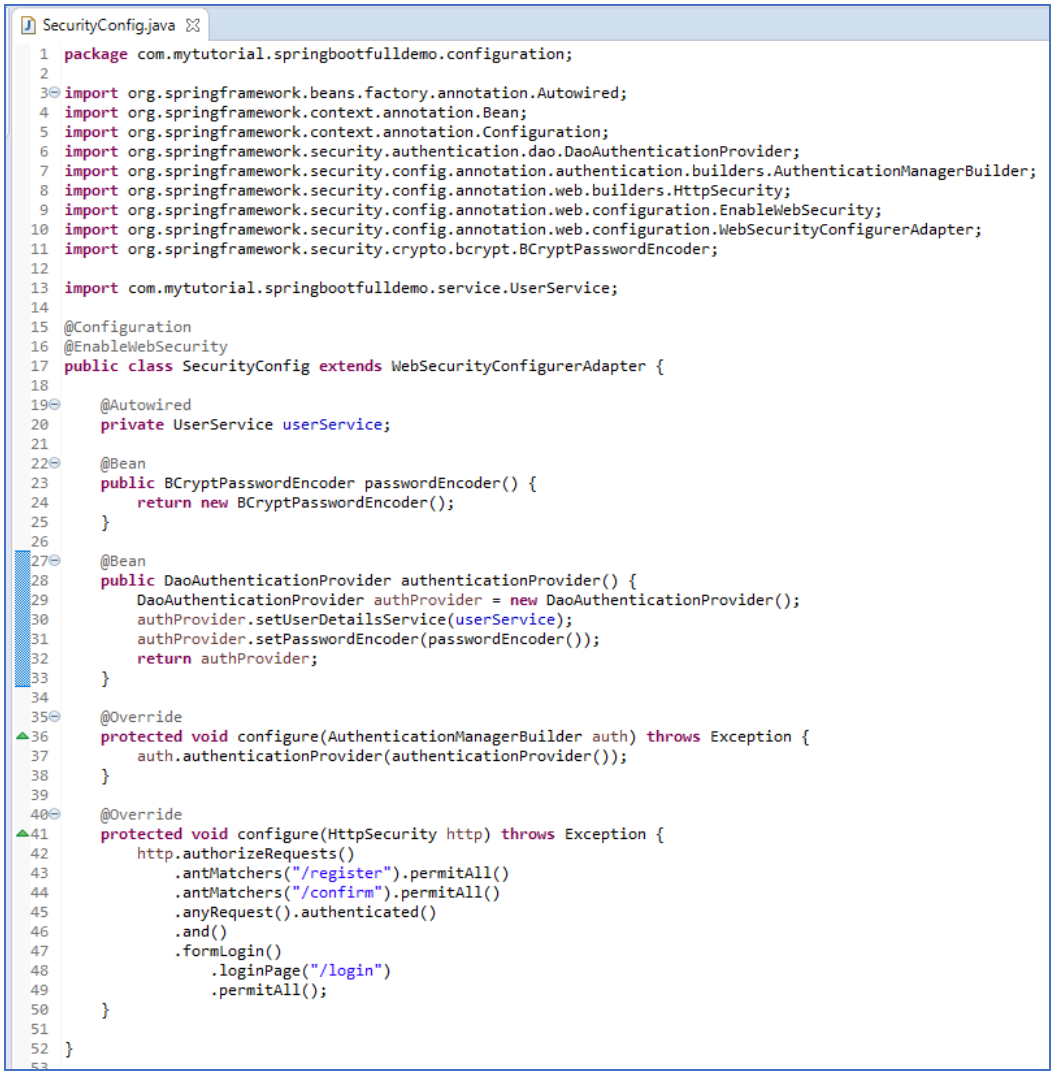
Specifying Security Settings

We've now seen the code for the model, the service layer, as well as the controllers. We'll now look at how we can set up the security configuration settings for our application, so that Spring Security will authenticate users from our underlying database table.

8. Setup Security Configuration Adapter

We'll specify these settings in the *SecurityConfig* class which extends the **WebSecurityConfigurerAdapter** as before.

SecurityConfig.java



```
1  package com.mytutorial.springbootfulldemo.configuration;
2
3  import org.springframework.beans.factory.annotation.Autowired;
4  import org.springframework.context.annotation.Bean;
5  import org.springframework.context.annotation.Configuration;
6  import org.springframework.security.authentication.dao.DaoAuthenticationProvider;
7  import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
8  import org.springframework.security.config.annotation.web.builders.HttpSecurity;
9  import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
10 import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
11 import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
12
13 import com.mytutorial.springbootfulldemo.service.UserService;
14
15 @Configuration
16 @EnableWebSecurity
17 public class SecurityConfig extends WebSecurityConfigurerAdapter {
18
19     @Autowired
20     private UserService userService;
21
22     @Bean
23     public BCryptPasswordEncoder passwordEncoder() {
24         return new BCryptPasswordEncoder();
25     }
26
27     @Bean
28     public DaoAuthenticationProvider authenticationProvider() {
29         DaoAuthenticationProvider authProvider = new DaoAuthenticationProvider();
30         authProvider.setUserDetailsService(userService);
31         authProvider.setPasswordEncoder(passwordEncoder());
32         return authProvider;
33     }
34
35     @Override
36     protected void configure(AuthenticationManagerBuilder auth) throws Exception {
37         auth.authenticationProvider(authenticationProvider());
38     }
39
40     @Override
41     protected void configure(HttpSecurity http) throws Exception {
42         http.authorizeRequests()
43             .antMatchers("/register").permitAll()
44             .antMatchers("/confirm").permitAll()
45             .anyRequest().authenticated()
46             .and()
47             .formLogin()
48                 .loginPage("/login")
49                 .permitAll();
50     }
51
52 }
```

- The **@EnableWebSecurity** annotation turns on Spring Security and integrates with Spring MVC.
- Now, this security configuration requires the use of the *UserService*. We'll use the **@Autowired** annotation to inject the *UserService* into this class.
- The password encoder that we use to generate an encryption hash of our password is the **BCryptPasswordEncoder**. We instantiate this password encoder within this bean here on line 23 and 22.

- We want Spring Security to authenticate only those users who have registered with our application and are present in the users table in the underlying MySQL database. So we need to specify the authentication provider to Spring Security.
  We specify this authentication provider as a Spring bean. This is the **DaoAuthenticationProvider**. We instantiate the Dao, set the user details service that the Dao should use, which is the user service that we had set up earlier.

  This user service integrates with our underlying users table in our MySQL database to find registered users.

  ```
  @Bean
  public DaoAuthenticationProvider authenticationProvider() {
        DaoAuthenticationProvider authProvider =
              new DaoAuthenticationProvider();
        authProvider.setUserDetailsService(userService);
        authProvider.setPasswordEncoder(passwordEncoder());
        return authProvider;
  }
  ```

  We also configure the passwordEncoder for this authentication provider, which is the BCryptPasswordEncoder. This we get by invoking the passwordEncoder method.
  And we return an instance of this authentication Provider.

- We have everything set up, we need to configure Spring Security to use the authentication provider that we have constructed.

  We do this within the configure method on line 36. We Override this method from the WebSecurityConfigurerAdapter base class. On this authentication manager builder, we set the authentication provider that we want Spring Security to use.

  ```
  @Override
  protected void configure(AuthenticationManagerBuilder auth)
  throws Exception {
        auth.authenticationProvider(authenticationProvider());
  }
  ```

- And then finally, we configure what pages we want secured and what pages should be available to all within the configure method overridden on line 41. We want our request to be secure.

  ```
  @Override
  protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
              .antMatchers("/register").permitAll()
              .antMatchers("/confirm").permitAll()
              .anyRequest().authenticated()
              .and()
              .formLogin()
                    .loginPage("/login")
                    .permitAll();
  }
  ```

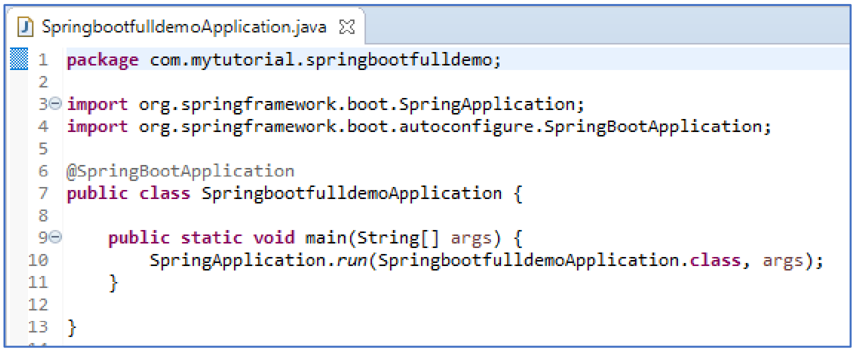
However, any request that is made to the `"/register"` path should be permitted to all without authentication. Any request to the `"/confirm"` path should also be permitted to all without authentication.

All requests other than to these two paths should be authenticated. So `.anyRequest().authenticated()` does that, this is on line 45.

Otherwise, we need to display a login page at `"/login"` to the user. And on line 49, you can see that access to the login page is also permitted to all users without authentication.

9. Inspect Entry Point of Spring boot Application

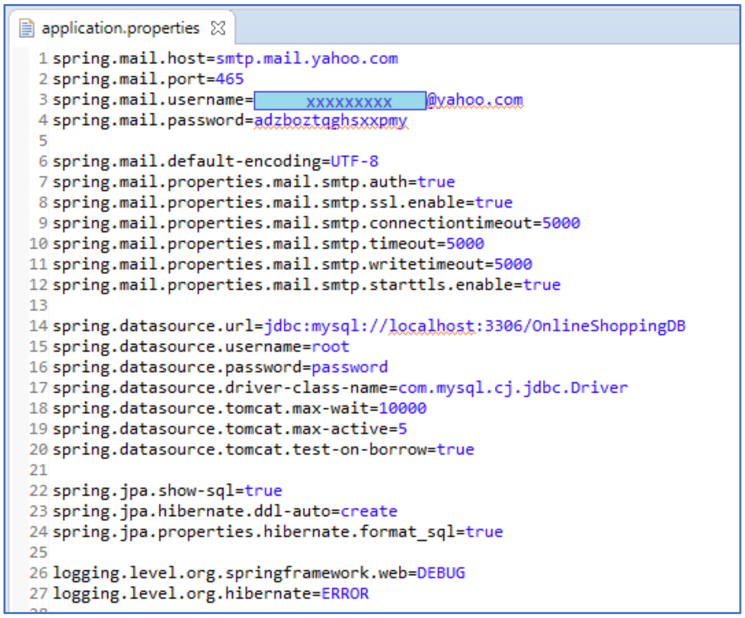
Now, we'll look at the last few files that make up our app. Here's the entry point to our application, the **SpringBootApplication.java** file.



```java
package com.mytutorial.springbootfulldemo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringbootfulldemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringbootfulldemoApplication.class, args);
    }

}
```

10. Configure application properties

Let's head over to application properties, where we'll configure the Java mail sender, as well as the connection to our underlying MySQL database. On the first six lines of this properties file, we have the configuration properties for the Java mail sender that our Spring Boot application uses. We'll send mail from *xxxxxx@yahoo.com* using *smtp.mail.yahoo.com*.



```
 1 spring.mail.host=smtp.mail.yahoo.com
 2 spring.mail.port=465
 3 spring.mail.username=      xxxxxxxxx      @yahoo.com
 4 spring.mail.password=adzboztqghsxxpmy
 5
 6 spring.mail.default-encoding=UTF-8
 7 spring.mail.properties.mail.smtp.auth=true
 8 spring.mail.properties.mail.smtp.ssl.enable=true
 9 spring.mail.properties.mail.smtp.connectiontimeout=5000
10 spring.mail.properties.mail.smtp.timeout=5000
11 spring.mail.properties.mail.smtp.writetimeout=5000
12 spring.mail.properties.mail.smtp.starttls.enable=true
13
14 spring.datasource.url=jdbc:mysql://localhost:3306/OnlineShoppingDB
15 spring.datasource.username=root
16 spring.datasource.password=password
17 spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
18 spring.datasource.tomcat.max-wait=10000
19 spring.datasource.tomcat.max-active=5
20 spring.datasource.tomcat.test-on-borrow=true
21
22 spring.jpa.show-sql=true
23 spring.jpa.hibernate.ddl-auto=create
24 spring.jpa.properties.hibernate.format_sql=true
25
26 logging.level.org.springframework.web=DEBUG
27 logging.level.org.hibernate=ERROR
```

On last lines, we have some logging related properties. We have debug level logging for the springframework.web namespace. And we have error level logging for the hibernate ORM framework.

Lines 14 to 16 contain the configuration properties for the underlying MySQL database. The database url is `jdbc:mysql://localhost:3306/OnlineShoppingDB`. The username is *root*, the password is *password*.

On lines 18-20, we have configuration properties for our tomcat server.

And finally on lines 22 - 24, we have configuration properties related to the hibernate ORM framework that integrates with the database for us. We'll show the sql that is executed. `spring.jpa.hibernate.ddl-auto=create` is the setting that is best used for prototyping and testing your code.

Each time your application is run, the old tables within your database will be dropped and new tables will be created.

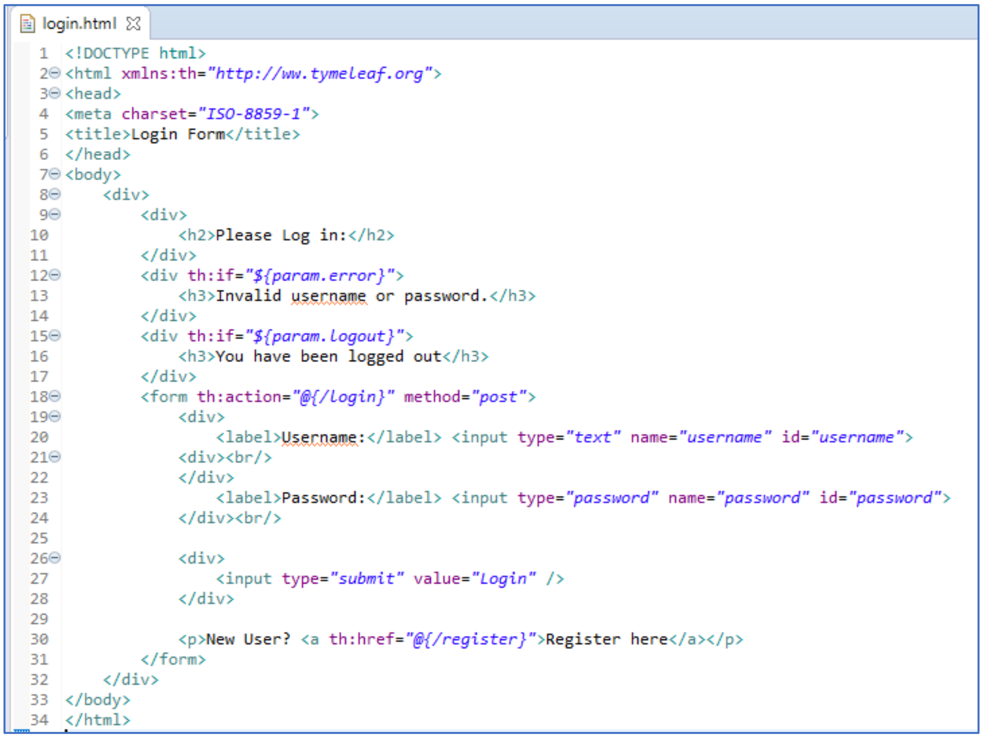
So we start afresh each time we run this application. All of our old data will be cleared.

11. Prepare View Layer Html Page

Let's look at the UI code.

- **login.html** is not very different from what we had before. You have some error messages up top, we have an input text box which accepts the username and password. When this form is submitted, we'll make a POST request to `/login`. This is on line 18.

login.html

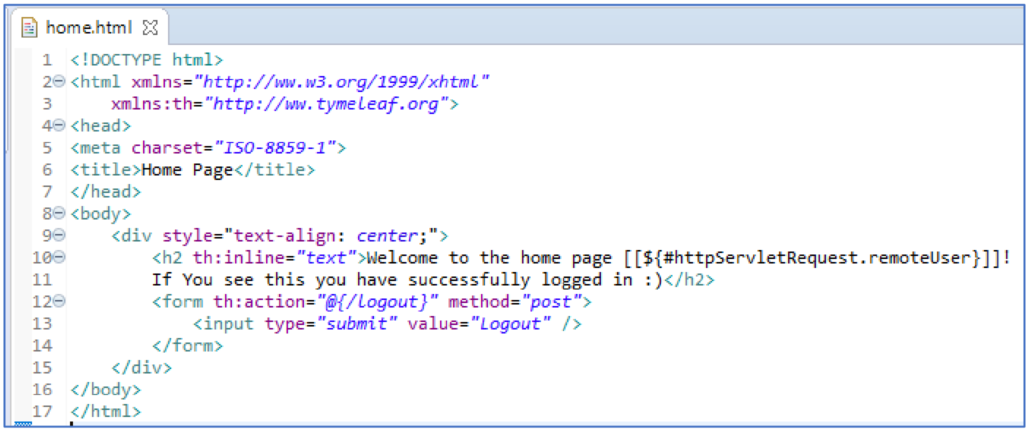


```
1  <!DOCTYPE html>
2  <html xmlns:th="http://ww.tymeleaf.org">
3  <head>
4  <meta charset="ISO-8859-1">
5  <title>Login Form</title>
6  </head>
7  <body>
8      <div>
9          <div>
10             <h2>Please Log in:</h2>
11         </div>
12         <div th:if="${param.error}">
13             <h3>Invalid username or password.</h3>
14         </div>
15         <div th:if="${param.logout}">
16             <h3>You have been logged out</h3>
17         </div>
18         <form th:action="@{/login}" method="post">
19             <div>
20                 <label>Username:</label> <input type="text" name="username" id="username">
21             <div><br/>
22             </div>
23                 <label>Password:</label> <input type="password" name="password" id="password">
24             </div><br/>
25
26             <div>
27                 <input type="submit" value="Login" />
28             </div>
29
30             <p>New User? <a th:href="@{/register}">Register here</a></p>
31         </form>
32     </div>
33 </body>
34 </html>
```

For a new user on line 30, we have a link to the `/register` page.

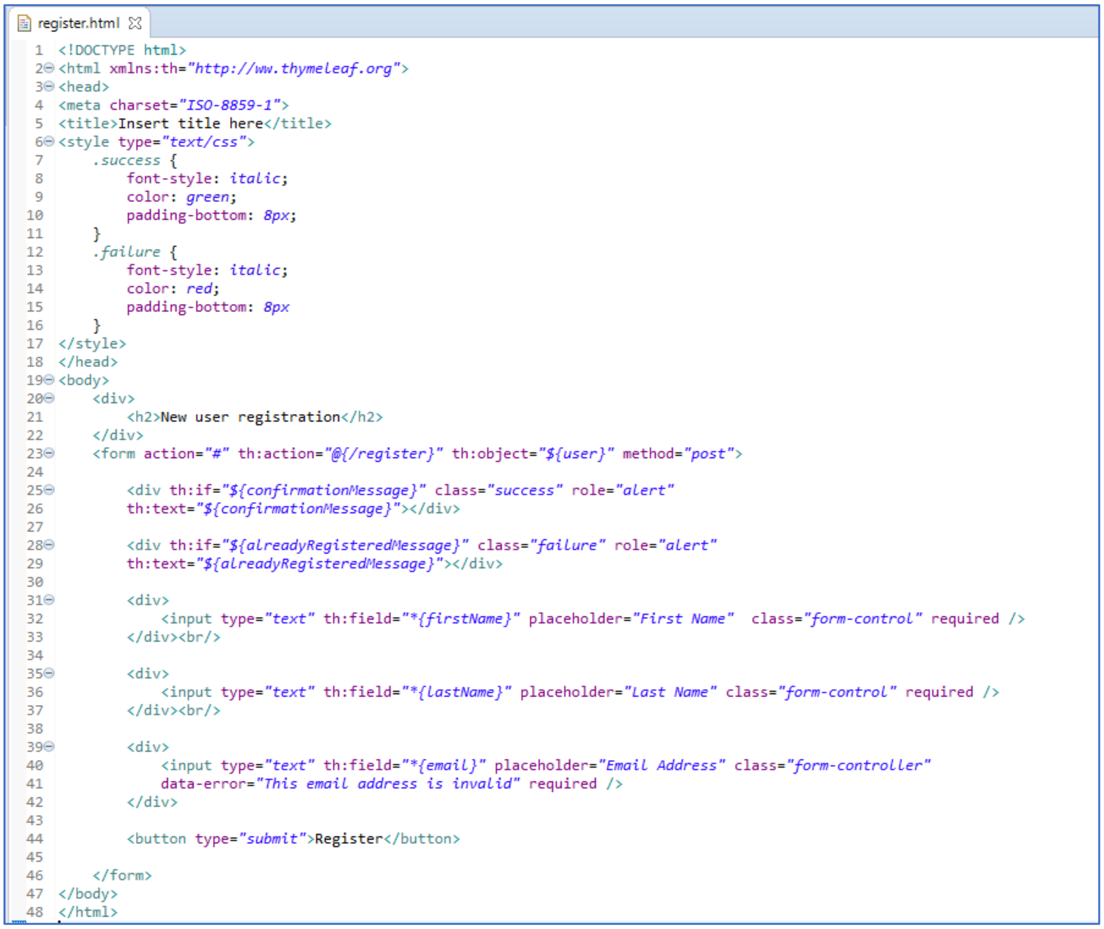
- The homepage is no different from what we had in our previous demo. We have a header which says, "*Welcome to the home page* [the name of the user]. *If you see this, you've successfully logged in*". We also have a button allowing users to logout.

home.html



```
<!DOCTYPE html>
<html xmlns="http://ww.w3.org/1999/xhtml"
    xmlns:th="http://ww.tymeleaf.org">
<head>
<meta charset="ISO-8859-1">
<title>Home Page</title>
</head>
<body>
    <div style="text-align: center;">
        <h2 th:inline="text">Welcome to the home page [[${#httpServletRequest.remoteUser}]]!
        If You see this you have successfully logged in :)</h2>
        <form th:action="@{/logout}" method="post">
            <input type="submit" value="Logout" />
        </form>
    </div>
</body>
</html>
```

- A new page that we have in this app is the registration page, **register.html**.



register.html

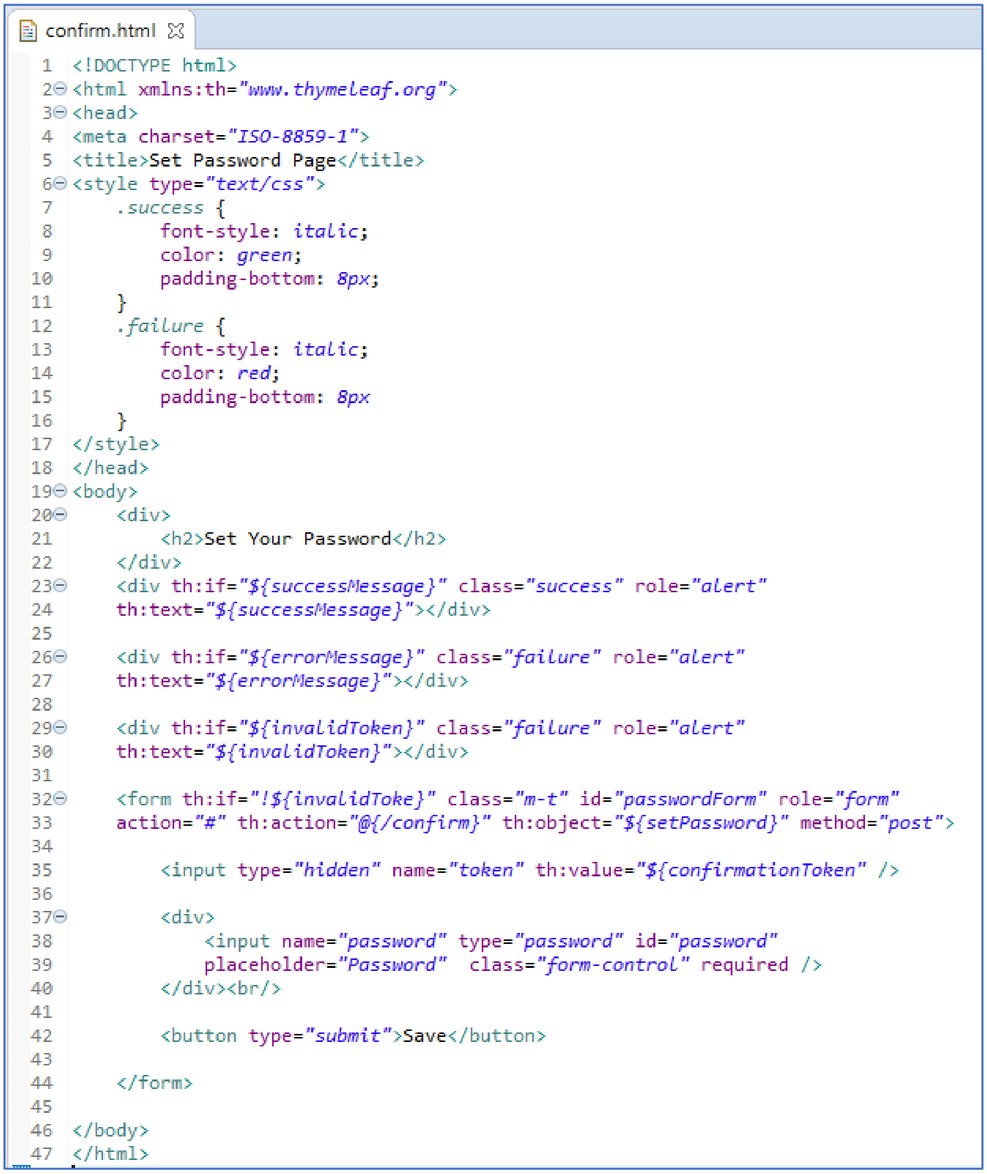
```
1  <!DOCTYPE html>
2  <html xmlns:th="http://ww.thymeleaf.org">
3  <head>
4  <meta charset="ISO-8859-1">
5  <title>Insert title here</title>
6  <style type="text/css">
7      .success {
8          font-style: italic;
9          color: green;
10         padding-bottom: 8px;
11     }
12     .failure {
13         font-style: italic;
14         color: red;
15         padding-bottom: 8px
16     }
17 </style>
18 </head>
19 <body>
20     <div>
21         <h2>New user registration</h2>
22     </div>
23     <form action="#" th:action="@{/register}" th:object="${user}" method="post">
24 
25         <div th:if="${confirmationMessage}" class="success" role="alert"
26         th:text="${confirmationMessage}"></div>
27 
28         <div th:if="${alreadyRegisteredMessage}" class="failure" role="alert"
29         th:text="${alreadyRegisteredMessage}"></div>
30 
31         <div>
32             <input type="text" th:field="*{firstName}" placeholder="First Name"  class="form-control" required />
33         </div><br/>
34 
35         <div>
36             <input type="text" th:field="*{lastName}" placeholder="Last Name" class="form-control" required />
37         </div><br/>
38 
39         <div>
40             <input type="text" th:field="*{email}" placeholder="Email Address" class="form-controller"
41             data-error="This email address is invalid" required />
42         </div>
43 
44         <button type="submit">Register</button>
45 
46     </form>
47 </body>
48 </html>
```

We set up a form here on line 23. When this form is submitted, we make a POST request to /register. If you have messages from the server, these are displayed to the user on lines 25, and 28. A confirmation message, or an already registered message.

The form itself is very, very simple. We simply accept the first name and last name of the user, along with the email address.

- Next, we look at the Confirm page.

confirm.html



```
1  <!DOCTYPE html>
2  <html xmlns:th="www.thymeleaf.org">
3  <head>
4  <meta charset="ISO-8859-1">
5  <title>Set Password Page</title>
6  <style type="text/css">
7      .success {
8          font-style: italic;
9          color: green;
10         padding-bottom: 8px;
11     }
12     .failure {
13         font-style: italic;
14         color: red;
15         padding-bottom: 8px
16     }
17 </style>
18 </head>
19 <body>
20     <div>
21         <h2>Set Your Password</h2>
22     </div>
23     <div th:if="${successMessage}" class="success" role="alert"
24     th:text="${successMessage}"></div>
25
26     <div th:if="${errorMessage}" class="failure" role="alert"
27     th:text="${errorMessage}"></div>
28
29     <div th:if="${invalidToken}" class="failure" role="alert"
30     th:text="${invalidToken}"></div>
31
32     <form th:if="!${invalidToke}" class="m-t" id="passwordForm" role="form"
33     action="#" th:action="@{/confirm}" th:object="${setPassword}" method="post">
34
35         <input type="hidden" name="token" th:value="${confirmationToken" />
36
37         <div>
38             <input name="password" type="password" id="password"
39             placeholder="Password"  class="form-control" required />
40         </div><br/>
41
42         <button type="submit">Save</button>
43
44     </form>
45
46 </body>
47 </html>
```

When the user clicks on the link that he gets in the URL, what he gets to is this Confirm page. The Confirm page is fairly simple. It can display a number of messages which are set by our backend. This is the code on lines 23 through 30.

Otherwise, all the user has to do is set a password using the input textbox specified. There is a form here that is displayed if the token is a valid, if not invalid token on line 32.

We've specified an input password box on line 38. The user will enter his or her password here.

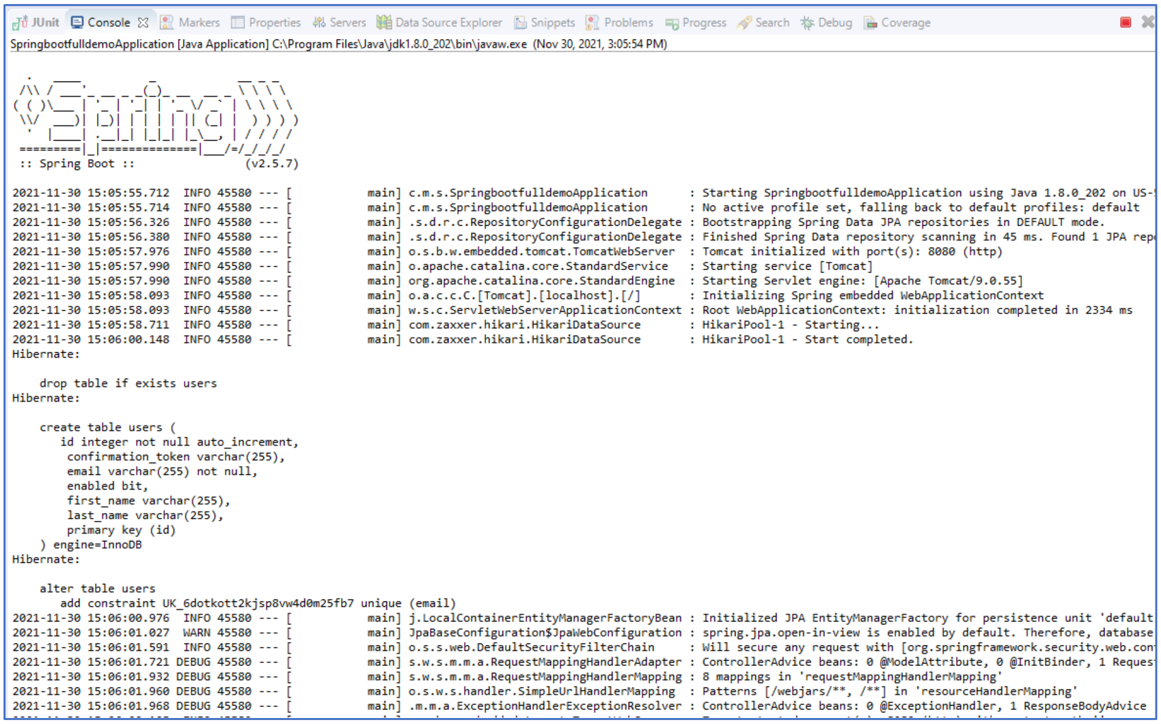
Note on line 35, that the confirmation token for this user is passed as a hidden field within this form. And when the user submits the form, a POST request will be sent to the **/confirm** path.

12. Enable Less Secure Apps on Mail Provider Setting

We've configured our email provider to send email from xxxxxx@yahoo.com. So I'm going to head over here and ensure that Less secure apps is enabled. The slide for less secure apps has been turned on for this account. We are good to go. As we've discussed in our earlier demo on sending email, we need to ensure that IMAP access for this account is also turned on. You can do this from within Yahoo settings.

13. Run and Test the Application

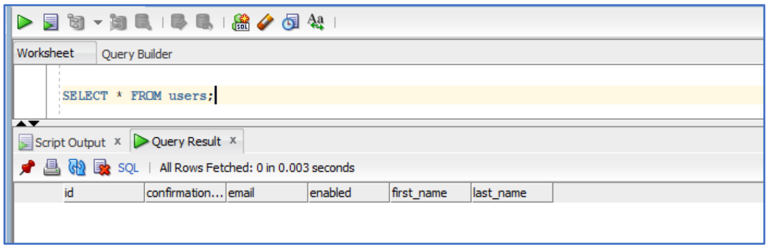
When we run our application, the Java Persistence API and the hibernate ORM framework will automatically drop all previous users table and create a new users table for us. Notice within the console logs, we have a drop table if exists users. And then a little bit below that, create table users with all of the columns specified in the user entity.



SpringbootfulldemoApplication [Java Application] C:\Program Files\Java\jdk1.8.0_202\bin\javaw.exe (Nov 30, 2021, 3:05:54 PM)

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.5.7)

2021-11-30 15:05:55.712  INFO 45580 --- [           main] c.m.s.SpringbootfulldemoApplication      : Starting SpringbootfulldemoApplication using Java 1.8.0_202 on US-
2021-11-30 15:05:55.714  INFO 45580 --- [           main] c.m.s.SpringbootfulldemoApplication      : No active profile set, falling back to default profiles: default
2021-11-30 15:05:56.326  INFO 45580 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEFAULT mode.
2021-11-30 15:05:56.380  INFO 45580 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 45 ms. Found 1 JPA rep
2021-11-30 15:05:57.976  INFO 45580 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2021-11-30 15:05:57.990  INFO 45580 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2021-11-30 15:05:57.990  INFO 45580 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.55]
2021-11-30 15:05:58.093  INFO 45580 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2021-11-30 15:05:58.093  INFO 45580 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 2334 ms
2021-11-30 15:05:58.711  INFO 45580 --- [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Starting...
2021-11-30 15:06:00.148  INFO 45580 --- [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Start completed.
Hibernate: 

    drop table if exists users
Hibernate: 

    create table users (
       id integer not null auto_increment,
        confirmation_token varchar(255),
        email varchar(255) not null,
        enabled bit,
        first_name varchar(255),
        last_name varchar(255),
        primary key (id)
    ) engine=InnoDB
Hibernate: 

    alter table users
       add constraint UK_6dotkott2kjsp8vw4d0m25fb7 unique (email)
2021-11-30 15:06:00.976  INFO 45580 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default
2021-11-30 15:06:01.027  WARN 45580 --- [           main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database
2021-11-30 15:06:01.591  INFO 45580 --- [           main] o.s.s.web.DefaultSecurityFilterChain     : Will secure any request with [org.springframework.security.web.con
2021-11-30 15:06:01.721 DEBUG 45580 --- [           main] s.w.s.m.m.a.RequestMappingHandlerAdapter : ControllerAdvice beans: 0 @ModelAttribute, 0 @InitBinder, 1 Reques
2021-11-30 15:06:01.932 DEBUG 45580 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : 8 mappings in 'requestMappingHandlerMapping'
2021-11-30 15:06:01.960 DEBUG 45580 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Patterns [/webjars/**, /**] in 'resourceHandlerMapping'
2021-11-30 15:06:01.968 DEBUG 45580 --- [           main] .m.m.a.ExceptionHandlerExceptionResolver : ControllerAdvice beans: 0 @ExceptionHandler, 1 ResponseBodyAdvice
```

We can now switch over to MySQL workbench once again and run a SELECT * FROM users. You can see that the users table has been created but there are no records within this table yet. We have no registered users.

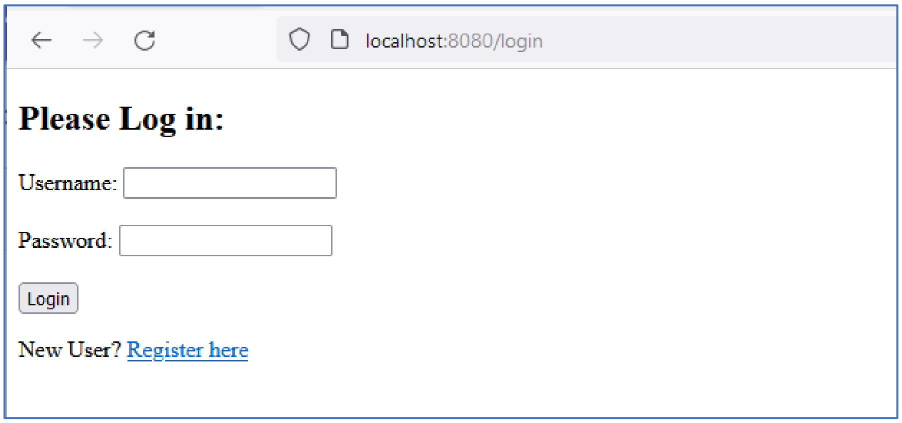


```
SELECT * FROM users;
```

All Rows Fetched: 0 in 0.003 seconds

id	confirmation...	email	enabled	first_name	last_name

Our app is running, let's head over to localhost 8080 and play around with it. We are immediately directed to the login page.



localhost:8080/login

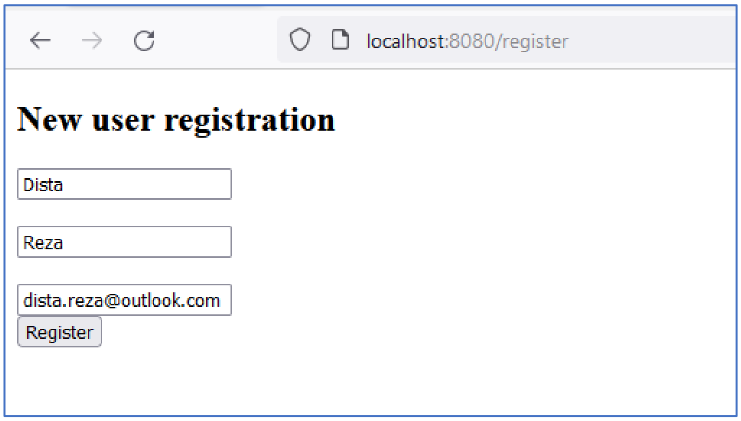
Please Log in:

Username:

Password:

New User? Register here

We have no users who have registered with our application. So I'm going to click on this link which says register here, that'll take me to the registration page. I'm going to register as Dista Reza with the email address dista.reza@outlook.com.



localhost:8080/register

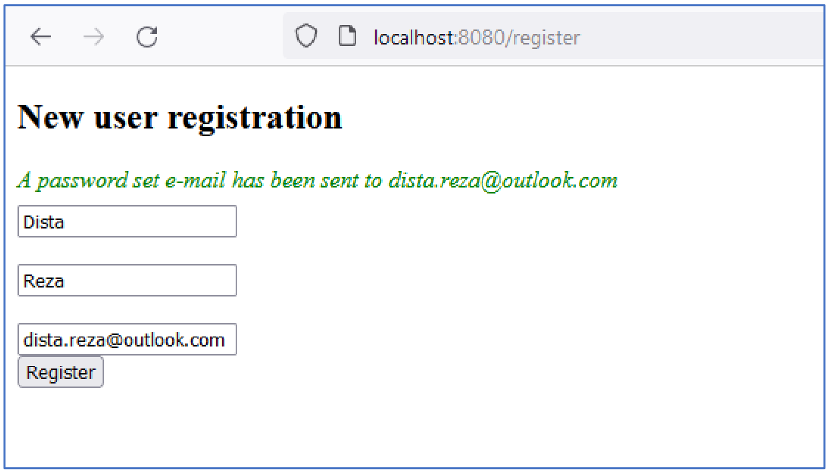
New user registration

Dista

Reza

dista.reza@outlook.com

Click on the Register option, and you should see a success message which says A password set e-mail has been sent to dista.reza@outlook.com.



localhost:8080/register

New user registration

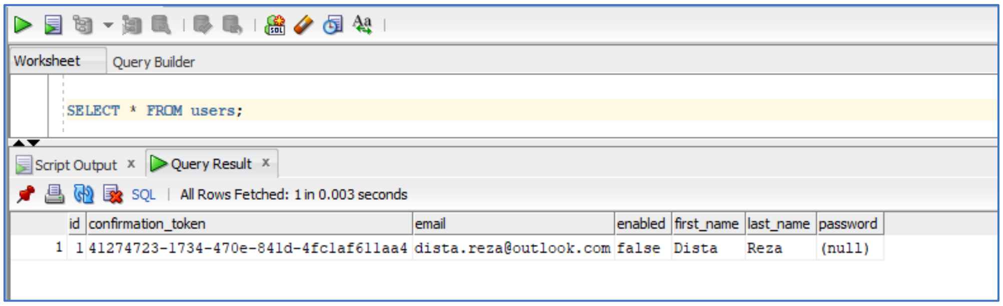
A password set e-mail has been sent to dista.reza@outlook.com

Dista

Reza

dista.reza@outlook.com

Now before we check our email, let's run a SELECT * FROM users on our users table. And you can see the entry for Bob right there. There are two things to observe here. The enabled column for Dista Reza is set to 0. And the password is set to Null. So Dista Reza isn't enabled yet. He has not set his password.

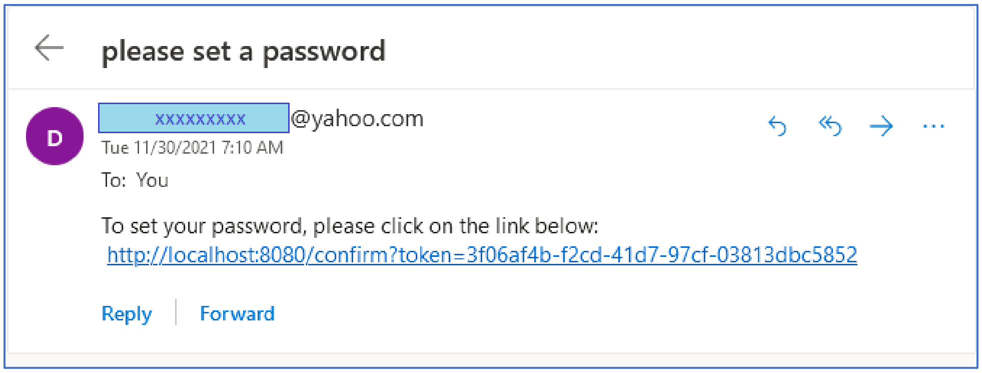


```
SELECT * FROM users;
```

All Rows Fetched: 1 in 0.003 seconds

	id	confirmation_token	email	enabled	first_name	last_name	password
1	1	41274723-1734-470e-841d-4fc1af611aa4	dista.reza@outlook.com	false	Dista	Reza	(null)

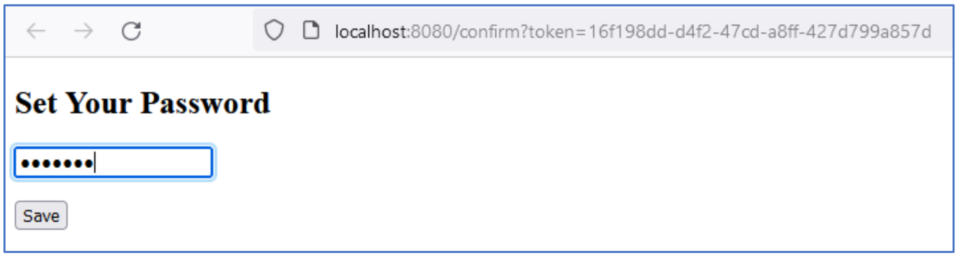
So let's head over to Dista Reza's email and you can see the email from *xxxxxx@yahoo.com* there saying please set a password. If you click on this email, you will see a link within that email along with a confirmation token.


please set a password

xxxxxxxxx@yahoo.com
Tue 11/30/2021 7:10 AM
To: You

To set your password, please click on the link below:
http://localhost:8080/confirm?token=3f06af4b-f2cd-41d7-97cf-03813dbc5852

This confirmation token is associated with Dista Reza. And when Dista Reza clicks on this link, he'll be taken to a confirm password page. This is where he can set his password.

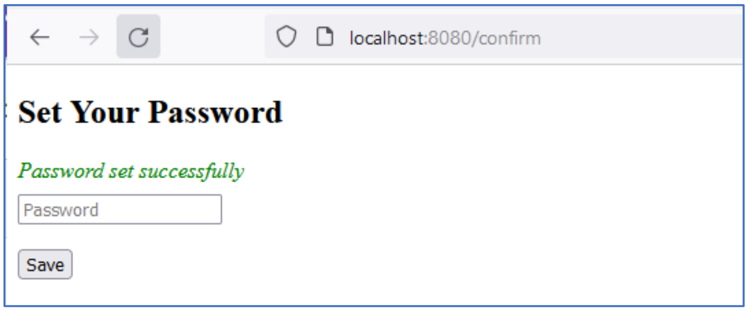

localhost:8080/confirm?token=16f198dd-d4f2-47cd-a8ff-427d799a857d

Set Your Password

•••••••

Notice the URL path **/confirm** with token set to *Dista Reza*'s token. I'm going to type in a password here and hit the Save button.

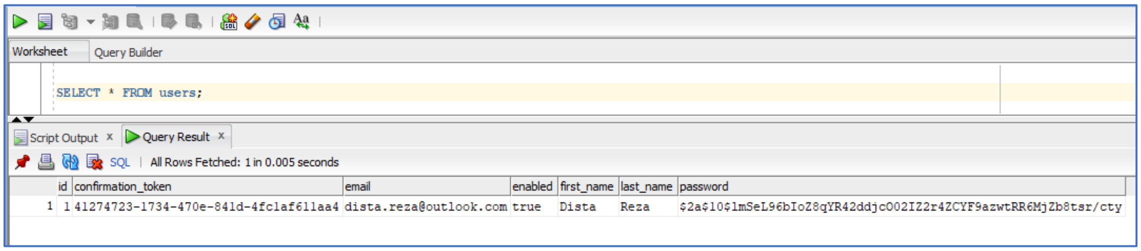
You should get a success message saying Password set successfully.



Set Your Password

Password set successfully

Dista Reza can now log into our app. But before that, let's run a SELECT * FROM users. On our MySQL Workbench, you can see that Bob is now enabled. The enabled column has value 1, and you can see the password column also has the Bcrypt encoded password.

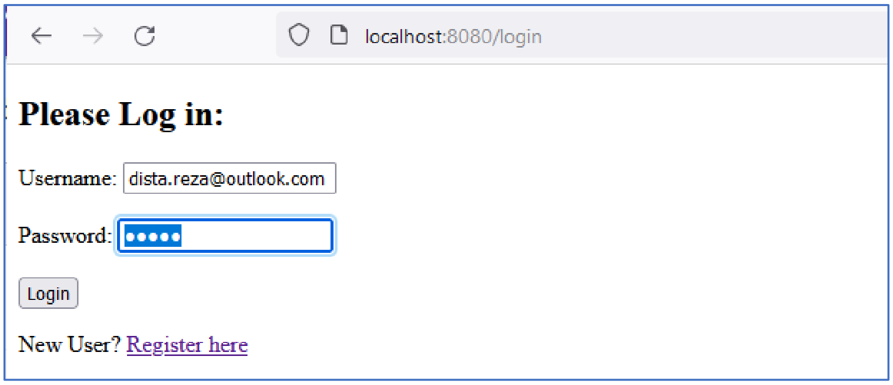


```
SELECT * FROM users;
```

All Rows Fetched: 1 in 0.005 seconds

	id	confirmation_token	email	enabled	first_name	last_name	password
1	1	41274723-1734-470e-841d-4fc1af611aa4	dista.reza@outlook.com	true	Dista	Reza	$2a$10$lmSeL96bIoZ8qYR42ddjcO02IZ2r4ZCYF9azwtRR6MjZb8tsr/cty

Now let's head back to localhost 8080 and log in as Dista Reza. We'll use *dista.reza@outlook.com* and the password that we had specified earlier.



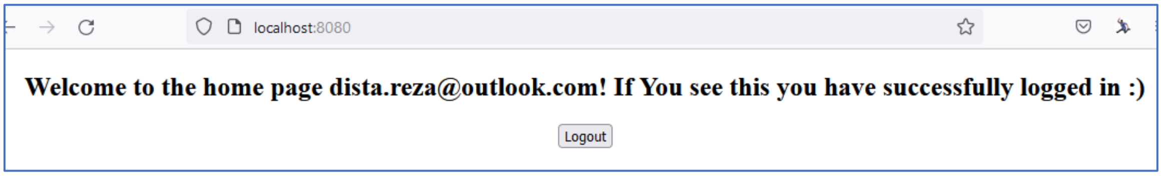
Please Log in:

Username: dista.reza@outlook.com

Password: •••••

New User? Register here

Click on the login button and our login will be successful. Here is the home page.



Welcome to the home page dista.reza@outlook.com! If You see this you have successfully logged in :)

Course Summary

JavaMailSender and the Twilio platform

Routing and filtering of requests

Securing access using Spring security

In this course,

- we integrated our Spring Boot application with external services, such as the JavaMailSender API and the Twilio platform.
- We configured interceptors and routers in our application, and we secured the request parts of our application using Spring security.
- We first configured our Spring Boot application to use the JavaMailSender API. We enabled mail settings for our Yahoo account and used that to successfully send emails with and without attachments.
- We also integrated our app with the Twilio platform, which allowed us to send messages and make phone calls.
- We then explored routing and filtering of requests to our web app, we saw the use of interceptors, which allowed us to pre-process request to our application, and post-process responses from our application. We use the **Zuul routing and filtering module from Netflix** to set-up an edge service in front of our application's APIs.
- Finally, we rounded off this course by integrating and working with this **Spring security module**.
- We configured the login service for our application using the built-in default user, in-memory users and finally registered users stored in a SQL database.

Quiz

1. What utility can you use to generate unique confirmation tokens?

   TokenGen
   UniqueTokenGenerator
   ✓ UUID
   ConfirmationTokenGenerator

2. In what order are Zuul filters executed?

   PreFilter, RouteFilter, PostFilter, ErrorFilter
   RouteFilter, PreFilter, ErrorFilter, PostFilter
   ✓ PreFilter, RouteFilter, ErrorFilter, PostFilter
   ErrorFilter, PreFilter, RouteFilter, PostFilter

3. What is Twilio?

   ✓ A cloud platform as a service which allows developers to send messages and make phone calls
   A third party caching services provider which helps speed up application responses
   A platform for streaming messages like Twitter
   An API testing tool to test CRUD operations

4. Using Spring Data, how do you perform CRUD operations on a database?

   Implement the CrudRepository<T, ID> inteface to perform CRUD operations on your table
   Use the Hibernate Persistence Manager to execute CRUD operations
   ✓ Set up an interface which extends CrudRepository<T, ID> and Spring Data will automatically provide an implementation
   Use JDBC templates directly to implement the CRUD operations

5. Which interface should you implement to specify interceptors for your web requests?

   WebInterceptor
   Interceptor
   ✓ HandlerInterceptor
   BaseInterceptor

6. What is Zuul?

   ✓ A gateway service that allows routing and filtering of requests to your web service
   A security service used to add a login page to access your APIs
   A technique used to monetize access to your APIs
   A session manager used to track client sessions with your APIs

7. What base class should you extend when you want to configure the security settings for your application?

   SpringSecurityAdapter
   WebMvcConfigurer

SecurityConfigurer

✔ WebSecurityConfigurerAdapter

8. What method would you override in the WebSecurityConfigurerAdapter, in order to specify which pages should be behind the login page?

login()

✔ configure()

secure()

authenticate()

9. What information does the DaoAuthenticationProvider need to perform authentication against your custom database store?

The JDBC URL to your database table that holds registered users

An in memory instance of all the users in your application

✔ An implementation of the UserDetailsService

✔ A password encoder to match passwords of logged in users

10. What are characteristics of the Java mail sender?

✔ You can specify HTML content in the text of the email

No additional configuration of the email service is required

Does not need the password of the from email address

✔ It is possible to send email with attachments

11. When you do not specify a user explicitly, what is the password for the default user that Spring Security configures?

The password is always "password123"

The password is generated the first time you log in

✔ The password is generated and displayed in the console logs

The password is the empty string by default