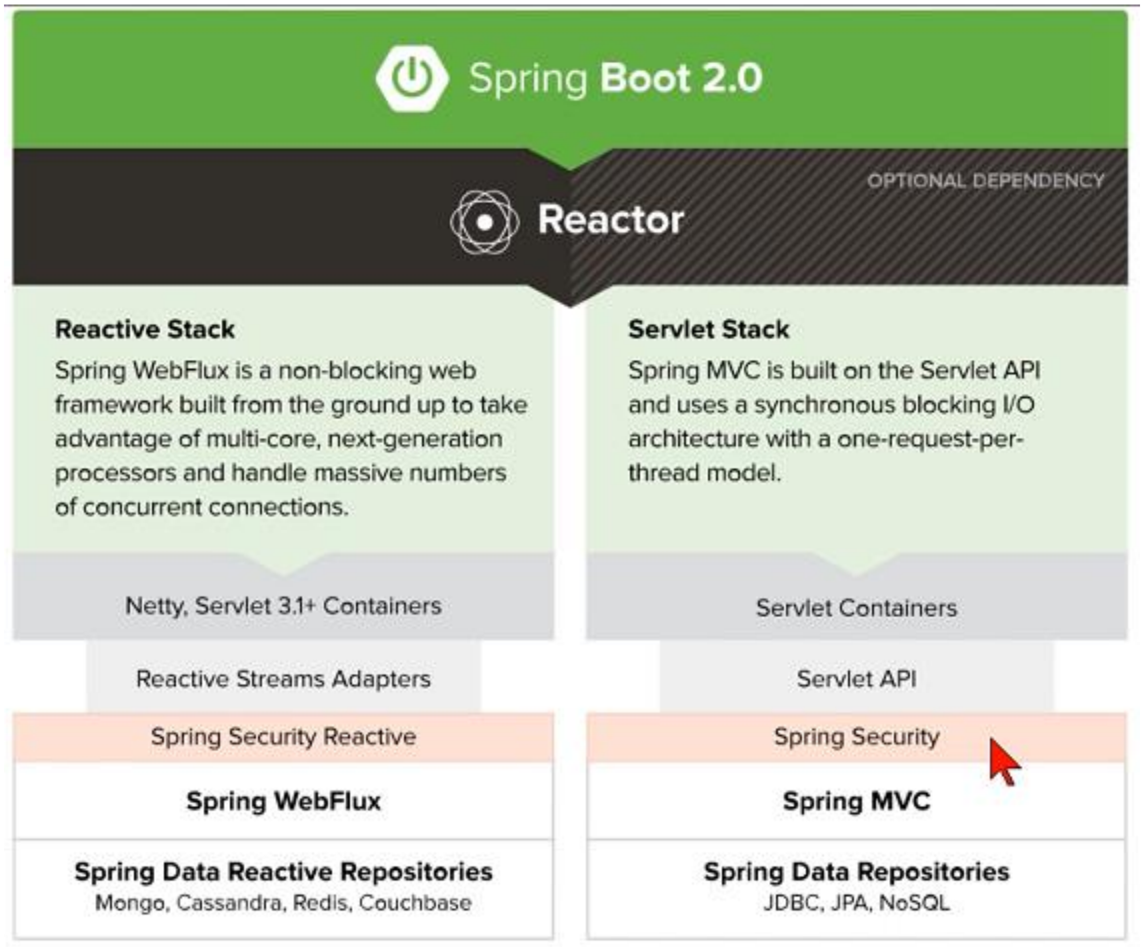
Spring

- **Intro:**

Spring is a Java Framework that makes it easier, quicker and safer to code → https://spring.io/

Spring framework evolves fast, providing modern tools for coding. Spring framework provides tools such as Spring Boot, designed to build any simple Java App; Spring Cloud, designed for distributed services based on microservices architecture; and Spring Cloud Data Flow, designed to connect Enterprises to IoT.

To this day, Spring Framework version is at 5.0 and Spring Boot is at 2.0. We have a technologies stack called Reactor, which provides two different architectural approaches: Reactive Stack and Servlet Stack.


Spring Boot 2.0
Reactor
OPTIONAL DEPENDENCY
Reactive Stack
Spring WebFlux is a non-blocking web framework built from the ground up to take advantage of multi-core, next-generation processors and handle massive numbers of concurrent connections.
Servlet Stack
Spring MVC is built on the Servlet API and uses a synchronous blocking I/O architecture with a one-request-per-thread model.
Netty, Servlet 3.1+ Containers
Servlet Containers
Reactive Streams Adapters
Servlet API
Spring Security Reactive
Spring Security
Spring WebFlux
Spring MVC
Spring Data Reactive Repositories
Mongo, Cassandra, Redis, Couchbase
Spring Data Repositories
JDBC, JPA, NoSQL

Basically, any Java application can now be easily build using Spring Boot, which we will be using, avoiding xml configurations and others.

Spring Initializr is the base to get started into building anything such as REST APIs, WebSockets, web streaming, tasks, etc. Security is simplified as well as SQL and NoSQL support. Even app servers are embedded into Spring Boot. Works with any IDE through dependencies.

- **Tools:**

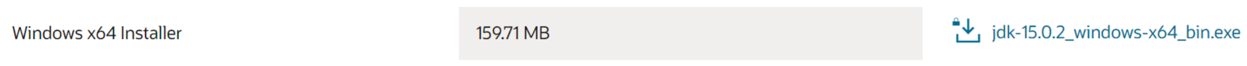
We will be using Java version 15 JDK, the latest stable version:
https://www.globalmentoring.com.mx/software/jdk15/index.html

https://www.oracle.com/java/technologies/javase/jdk15-archive-downloads.html

And Netbeans as the preferred IDE.

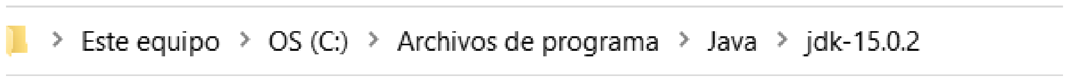
- **JDK Installation:**

https://www.oracle.com/java/technologies/javase/jdk15-archive-downloads.html



Windows x64 Installer	159.71 MB	jdk-15.0.2_windows-x64_bin.exe

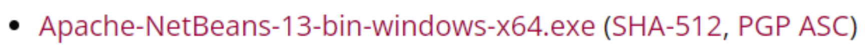
This is the default installation route:



> Este equipo > OS (C:) > Archivos de programa > Java > jdk-15.0.2

- **Apache Netbeans Installation:**

https://netbeans.apache.org/download/index.html



- Apache-NetBeans-13-bin-windows-x64.exe (SHA-512, PGP ASC)

Make sure that the Netbeans version supports the JDK version you intend to use. During Netbeans installation, provide the local route where your jdk version is stored. Unplug antivirus if it interferes with the IDE installation.

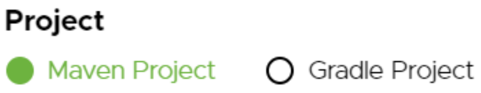
- **Apache Netbeans IDE:**

Change to dark theme: https://www.geeksforgeeks.org/how-to-change-the-theme-of-netbeans-12-0-to-dark-mode/

Tools → Options → Appearance → Look and Feel Tab → Dark Nimbus → Apply → Restart Netbeans.

- **Spring Initializr:**

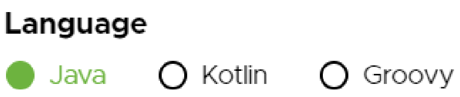
To start a Spring Project using Initializr, go to: https://start.spring.io/



Project

[✓ Maven Project] Gradle Project

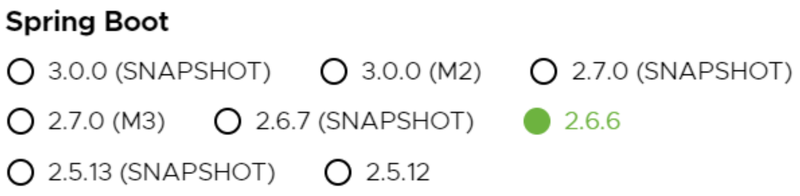
First choose the preferred Java Dependency Manager.



Language

[✓ Java] Kotlin Groovy

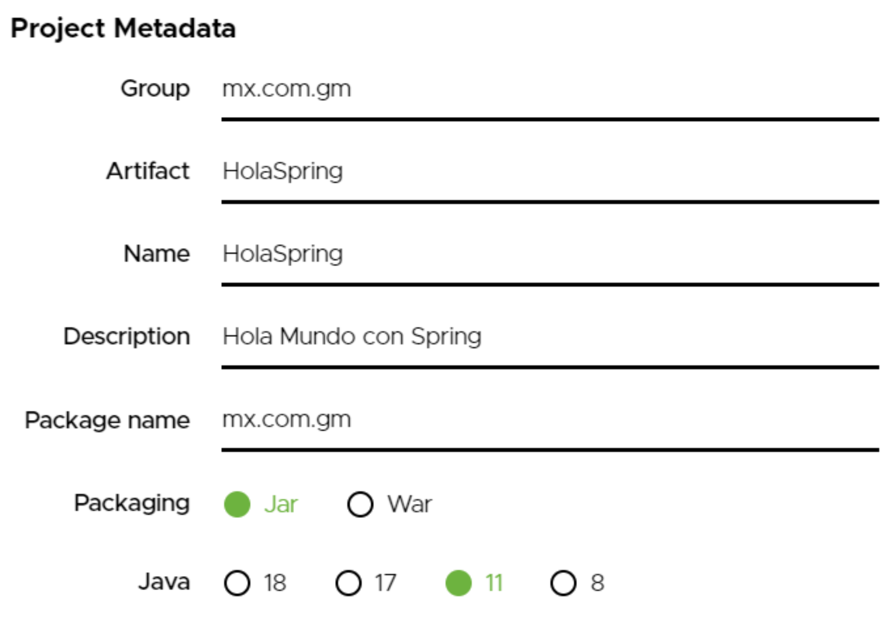
Then select the language to be used in your code.



Spring Boot

3.0.0 (SNAPSHOT) 3.0.0 (M2) 2.7.0 (SNAPSHOT)
2.7.0 (M3) 2.6.7 (SNAPSHOT) [✓ 2.6.6]
2.5.13 (SNAPSHOT) 2.5.12

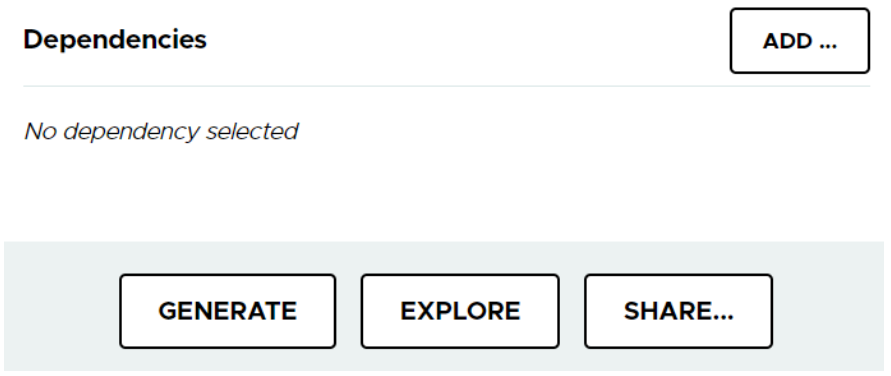
Then select the Spring Boot version to be used, SB will facilitate tasks such as the dependency management or the app server to be used. Use the stable version preferably, not SNAPSHOT or M.



Project Metadata

Group	mx.com.gm
Artifact	HolaSpring
Name	HolaSpring
Description	Hola Mundo con Spring
Package name	mx.com.gm
Packaging	[✓ Jar] War
Java	18 17 [✓ 11] 8

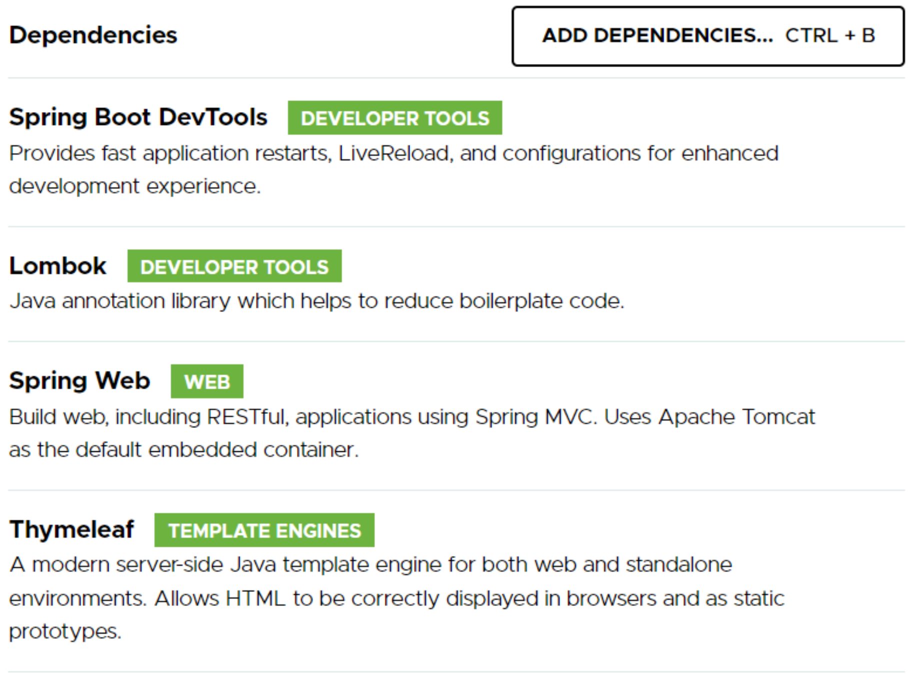
Provide the project metadata in order to create the working tree. Select the packaging type and remember that Jars can also now be used for web apps. Select the Java version to be used, in this case since 15 is not an option, select the immediate previous version.



Dependencies

No dependency selected

Last, we have the option to add dependencies. Remember Spring Boot already adds some, unlike simple Spring projects.



Dependencies

ADD DEPENDENCIES... CTRL + B

Spring Boot DevTools DEVELOPER TOOLS
Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

Lombok DEVELOPER TOOLS
Java annotation library which helps to reduce boilerplate code.

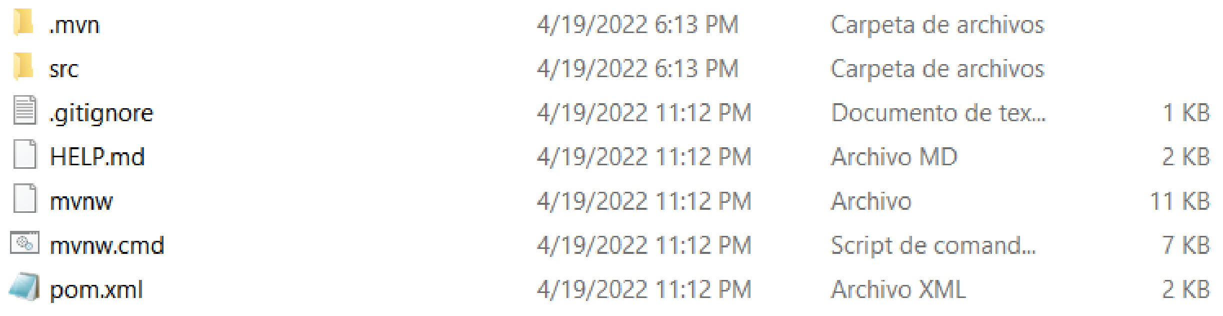
Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

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.

Add the dependencies relevant to your project and hit Generate.

Basically, Spring will act as our Container, while Spring Boot will facilitate all of the tedious configurations for us.

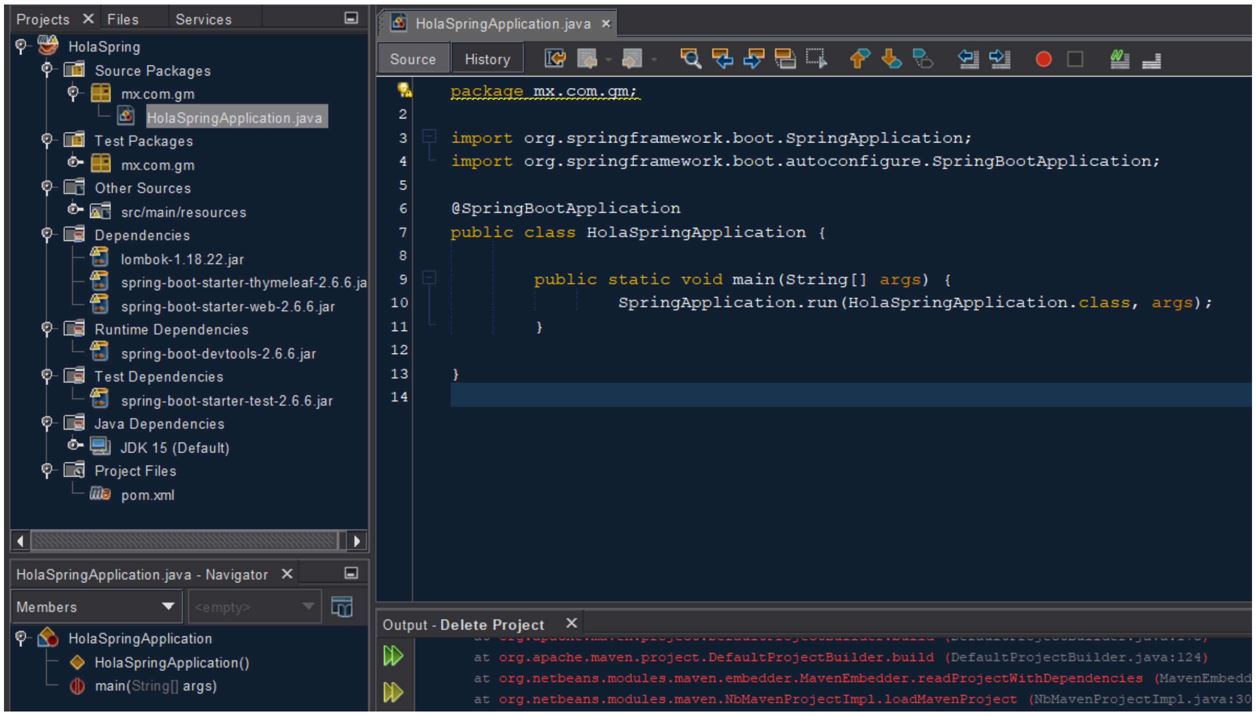
A zip file will be provided with our project. Unzip it and save it somewhere. It contains all of the files to your ready to run project. It even has a file for a Maven simulator (mvnw.cmd) in case you don't have it installed.



.mvn	4/19/2022 6:13 PM	Carpeta de archivos	
src	4/19/2022 6:13 PM	Carpeta de archivos	
.gitignore	4/19/2022 11:12 PM	Documento de tex...	1 KB
HELP.md	4/19/2022 11:12 PM	Archivo MD	2 KB
mvnw	4/19/2022 11:12 PM	Archivo	11 KB
mvnw.cmd	4/19/2022 11:12 PM	Script de comand...	7 KB
pom.xml	4/19/2022 11:12 PM	Archivo XML	2 KB

In src → main → java our class files will be contained. While in src → main → resources our configuration static files will be stored (js, css, images, etc), our Thymeleaf templates, and the application.properties file where we will be able to define the spring framework configurations (login, datasource connections, ports, etc).

To import this project into Netbeans, go to File → Open Project → Select Project Folder.



```java
package mx.com.gm;

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

@SpringBootApplication
public class HolaSpringApplication {

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


}
```

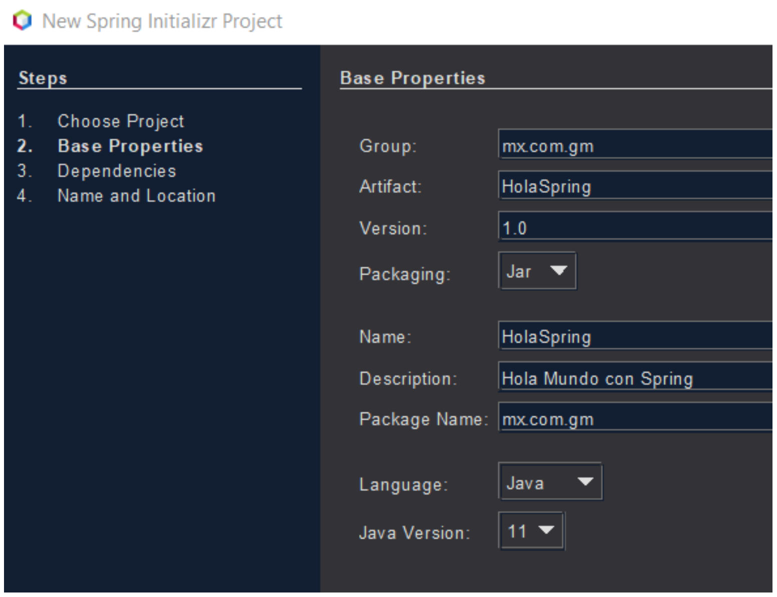
```
    at org.apache.maven.project.DefaultProjectBuilder.build (DefaultProjectBuilder.java:124)
    at org.netbeans.modules.maven.embedder.MavenEmbedder.readProjectWithDependencies (MavenEmbedd
    at org.netbeans.modules.maven.NbMavenProjectImpl.loadMavenProject (NbMavenProjectImpl.java:30
```

- **Create Project from Scratch with Spring Boot Plugin in Netbeans:**

Download the Spring Boot Netbeans Plugin → https://plugins.netbeans.apache.org/catalogue/?id=4

Go to Netbeans Tools → Plugins → Downloaded Tab → Add plugins → select plugin storage location → Open → Install → Accept terms → Install → Continue → Restart IDE.

Now we can create our project from scratch. Go to File → New Project → Java with Maven → Spring Boot Initializr Project → Next → Introduce data, select desired dependencies, name and location → Select path the closest to C:// to store project → Finish.



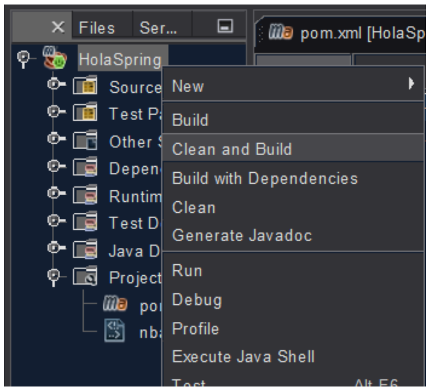
New Spring Initializr Project

Steps
1. Choose Project
2. **Base Properties**
3. Dependencies
4. Name and Location

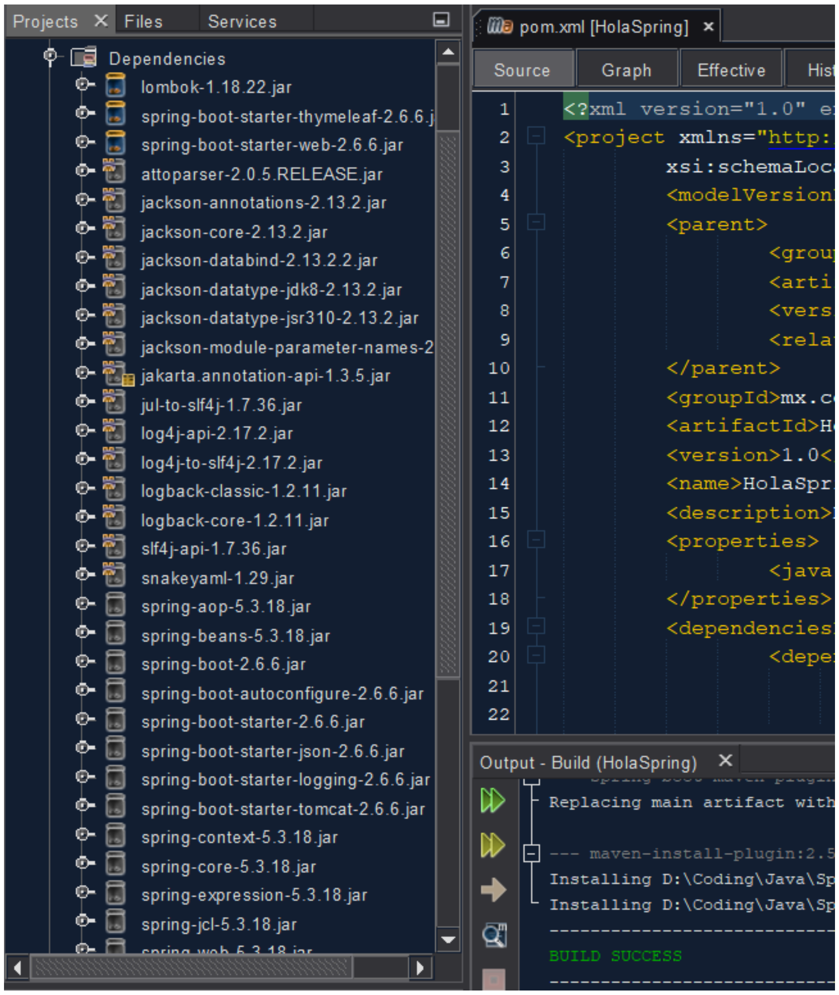
Base Properties

Group:	mx.com.gm
Artifact:	HolaSpring
Version:	1.0
Packaging:	Jar
Name:	HolaSpring
Description:	Hola Mundo con Spring
Package Name:	mx.com.gm
Language:	Java
Java Version:	11

Run the project for the first time to make sure that the dependencies are downloaded, go to project and right click Clean and Build. This will download any missing dependency. Make sure the output throws BUILD SUCCESS, otherwise disable security software such as antivirus that might interfere with dependency download.



If your dependencies are still not being downloaded, close your IDE, then go to C://Users/myuser/.m2 (this folder contains all maven dependencies) → Erase the repository folder containing all dependencies.



```
Dependencies
    lombok-1.18.22.jar
    spring-boot-starter-thymeleaf-2.6.6.j
    spring-boot-starter-web-2.6.6.jar
    attoparser-2.0.5.RELEASE.jar
    jackson-annotations-2.13.2.jar
    jackson-core-2.13.2.jar
    jackson-databind-2.13.2.2.jar
    jackson-datatype-jdk8-2.13.2.jar
    jackson-datatype-jsr310-2.13.2.jar
    jackson-module-parameter-names-2
    jakarta.annotation-api-1.3.5.jar
    jul-to-slf4j-1.7.36.jar
    log4j-api-2.17.2.jar
    log4j-to-slf4j-2.17.2.jar
    logback-classic-1.2.11.jar
    logback-core-1.2.11.jar
    slf4j-api-1.7.36.jar
    snakeyaml-1.29.jar
    spring-aop-5.3.18.jar
    spring-beans-5.3.18.jar
    spring-boot-2.6.6.jar
    spring-boot-autoconfigure-2.6.6.jar
    spring-boot-starter-2.6.6.jar
    spring-boot-starter-json-2.6.6.jar
    spring-boot-starter-logging-2.6.6.jar
    spring-boot-starter-tomcat-2.6.6.jar
    spring-context-5.3.18.jar
    spring-core-5.3.18.jar
    spring-expression-5.3.18.jar
    spring-jcl-5.3.18.jar
    spring-web-5.3.18.jar
```

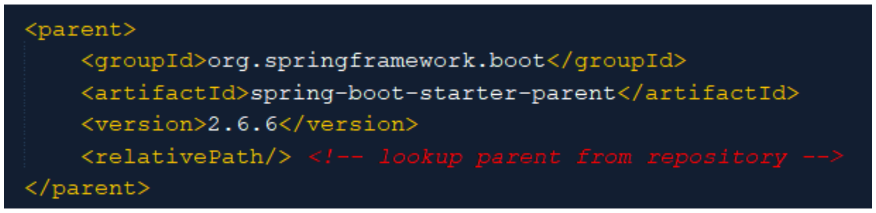
```xml
<?xml version="1.0" e
<project xmlns="http:
        xsi:schemaLoc
        <modelVersion
        <parent>
                <grou
                <arti
                <vers
                <rela
        </parent>
        <groupId>mx.c
        <artifactId>H
        <version>1.0<
        <name>HolaSpr
        <description>
        <properties>
                <java
        </properties>
        <dependencies
                <depe
```

```
  Replacing main artifact with

  --- maven-install-plugin:2.5
  Installing D:\Coding\Java\Sp
  Installing D:\Coding\Java\Sp
  ----------------------------
  BUILD SUCCESS
  ----------------------------
```

You can now appreciate all of the dependencies downloaded. If you don't get BUILD SUCCESS, one of them might be corrupt and you can try deleting the folder again as stated above.

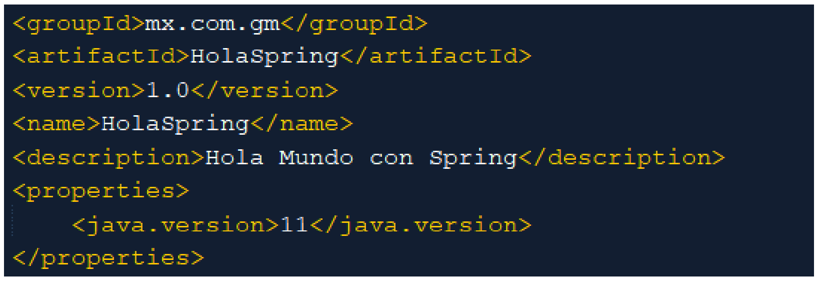
- **pom.xml:**

Now let’s take a look at our main project configuration file → pom.xml



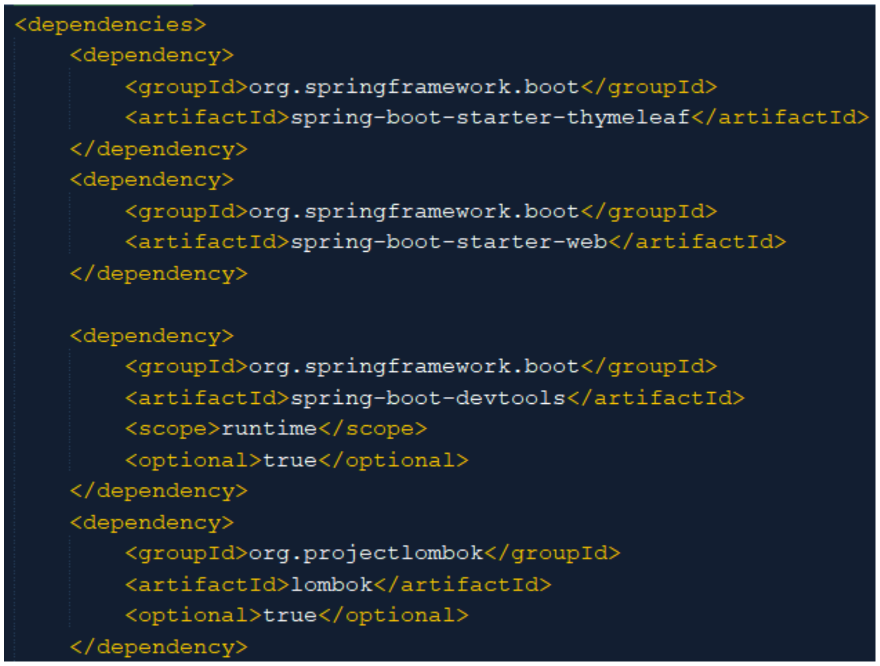
```xml
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.6.6</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
```

The parent dependency will be spring-boot-starter-parent, in this case version 2.6.6.



```xml
<groupId>mx.com.gm</groupId>
<artifactId>HolaSpring</artifactId>
<version>1.0</version>
<name>HolaSpring</name>
<description>Hola Mundo con Spring</description>
<properties>
    <java.version>11</java.version>
</properties>
```

Project info will also be stated here, as well as the java version.

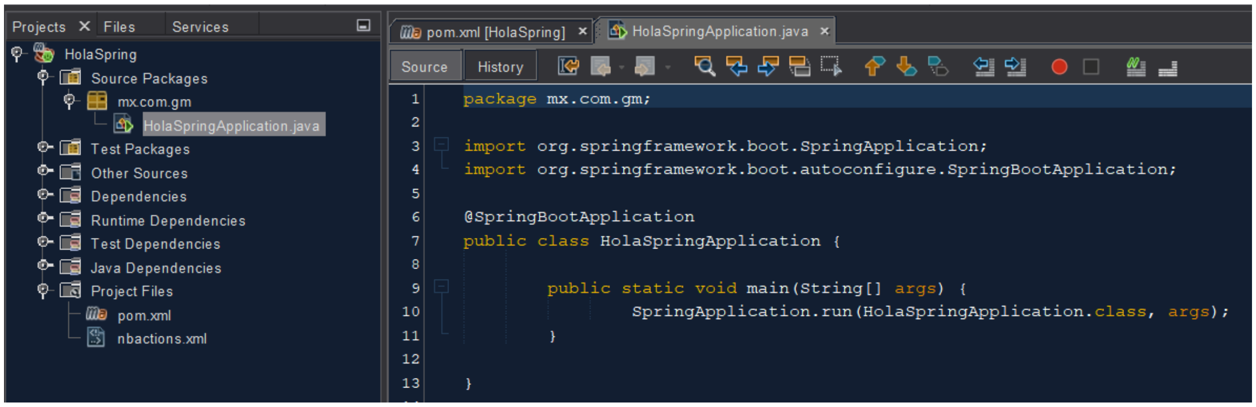


```xml
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
  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

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <scope>runtime</scope>
        <optional>true</optional>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>
```

These are secondary dependencies, the ones we decided to import and they will indirectly import many other dependencies to their name.

- **Main App Class:**

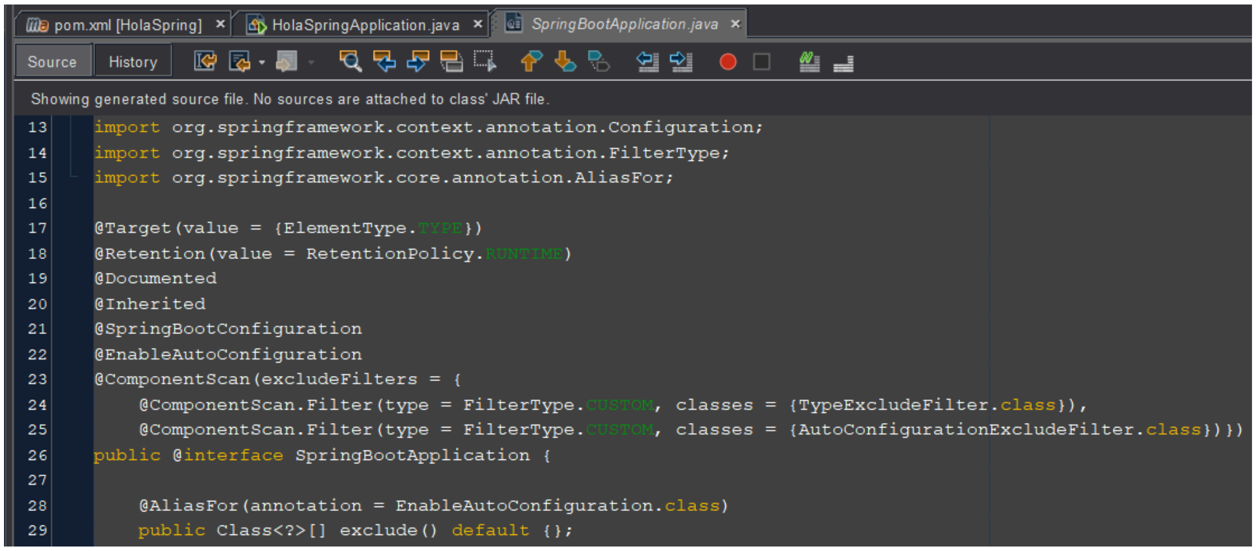
Our main app class is a regular java class which was automatically created by Spring Initializr, but with the @SpringBootApplication annotation.



```java
package mx.com.gm;

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

@SpringBootApplication
public class HolaSpringApplication {

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

}
```

The application class must have a main method, inside which we will be able to run the Spring Application. Right click, run file to execute the java app.

The @SpringBootApplication class is composed by several other annotations itself, this is the way SpringBoot facilitates Java App building.



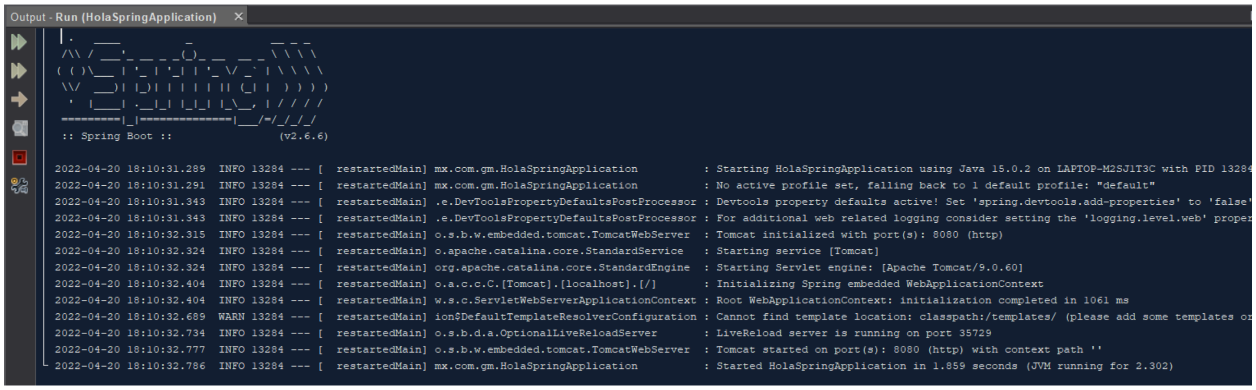
Showing generated source file. No sources are attached to class' JAR file.

```java
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.FilterType;
import org.springframework.core.annotation.AliasFor;

@Target(value = {ElementType.TYPE})
@Retention(value = RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
    @ComponentScan.Filter(type = FilterType.CUSTOM, classes = {TypeExcludeFilter.class}),
    @ComponentScan.Filter(type = FilterType.CUSTOM, classes = {AutoConfigurationExcludeFilter.class})})
public @interface SpringBootApplication {

    @AliasFor(annotation = EnableAutoConfiguration.class)
    public Class<?>[] exclude() default {};
```

Since Spring Framework is a java classes (beans) container, these beans will live inside the Spring container.

- **Executing App:**

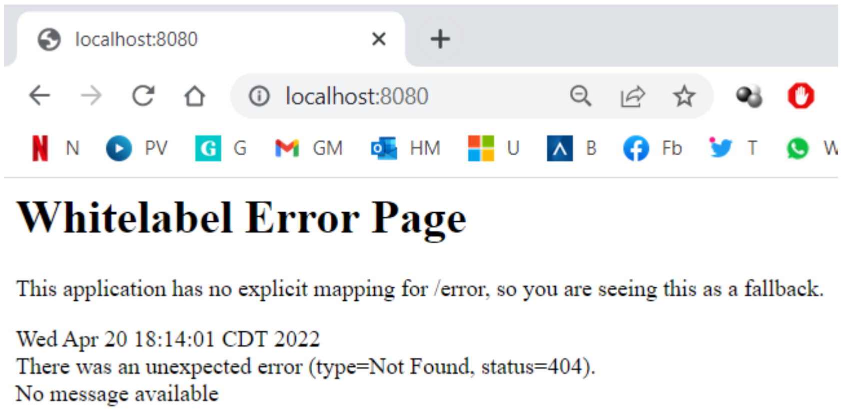
Right click over main app class (annotated with @SpringBootApplication) and hit Run File.



Output - Run (HolaSpringApplication)

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.6.6)

2022-04-20 18:10:31.289  INFO 13284 --- [  restartedMain] mx.com.gm.HolaSpringApplication          : Starting HolaSpringApplication using Java 15.0.2 on LAPTOP-M2SJ1T3C with PID 13284
2022-04-20 18:10:31.291  INFO 13284 --- [  restartedMain] mx.com.gm.HolaSpringApplication          : No active profile set, falling back to 1 default profile: "default"
2022-04-20 18:10:31.343  INFO 13284 --- [  restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : Devtools property defaults active! Set 'spring.devtools.add-properties' to 'false'
2022-04-20 18:10:31.343  INFO 13284 --- [  restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : For additional web related logging consider setting the 'logging.level.web' proper
2022-04-20 18:10:32.315  INFO 13284 --- [  restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2022-04-20 18:10:32.324  INFO 13284 --- [  restartedMain] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2022-04-20 18:10:32.324  INFO 13284 --- [  restartedMain] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.60]
2022-04-20 18:10:32.404  INFO 13284 --- [  restartedMain] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2022-04-20 18:10:32.404  INFO 13284 --- [  restartedMain] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1061 ms
2022-04-20 18:10:32.689  WARN 13284 --- [  restartedMain] ion$DefaultTemplateResolverConfiguration : Cannot find template location: classpath:/templates/ (please add some templates or
2022-04-20 18:10:32.734  INFO 13284 --- [  restartedMain] o.s.b.d.a.OptionalLiveReloadServer       : LiveReload server is running on port 35729
2022-04-20 18:10:32.777  INFO 13284 --- [  restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2022-04-20 18:10:32.786  INFO 13284 --- [  restartedMain] mx.com.gm.HolaSpringApplication          : Started HolaSpringApplication in 1.859 seconds (JVM running for 2.302)
```

Watch the output log. This will tell you whether the application is running and in which Tomcat port.



Whitelabel Error Page

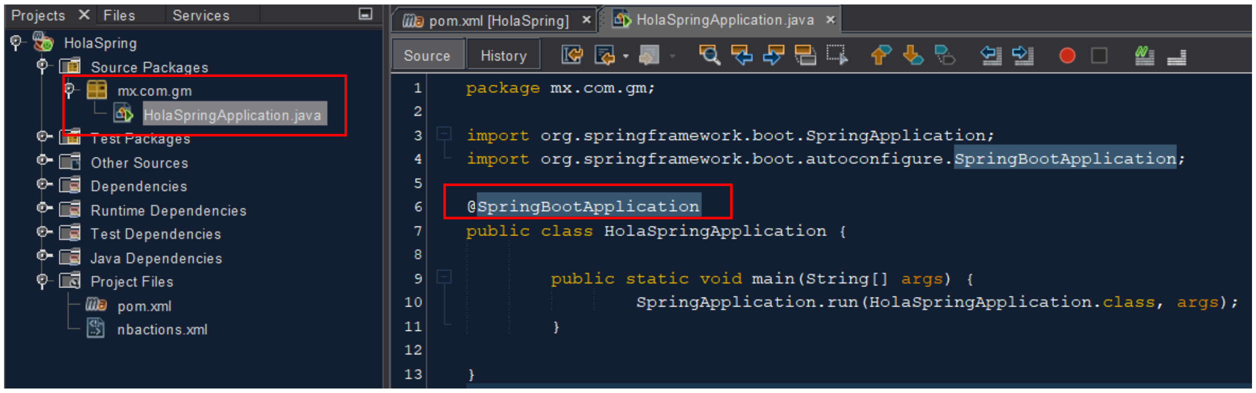
This application has no explicit mapping for /error, so you are seeing this as a fallback.

Wed Apr 20 18:14:01 CDT 2022
There was an unexpected error (type=Not Found, status=404).
No message available

You can now deploy your app in any browser.

- **REST Controller:**

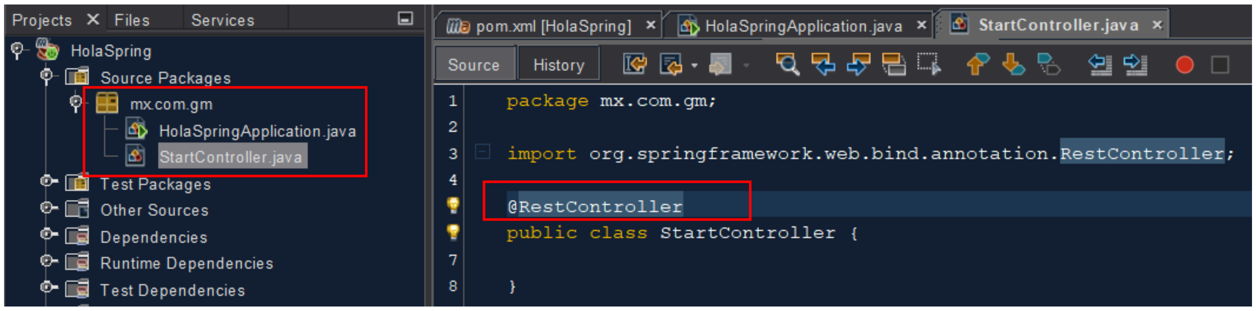
Since Spring a Java Classes Container, all of our java classes need to be inside the same package containing the @SpringBootApplication annotated class, in this case the package is called mx.com.gm.



```java
package mx.com.gm;

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

@SpringBootApplication
public class HolaSpringApplication {

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

}
```

The @ComponentScan annotation inside the SpringBootApplication class, will make sure that all classes inside this package are scanned.

So, let’s create a new class inside this package for our REST Controller. A REST Controller will allow us to send information to our browser.

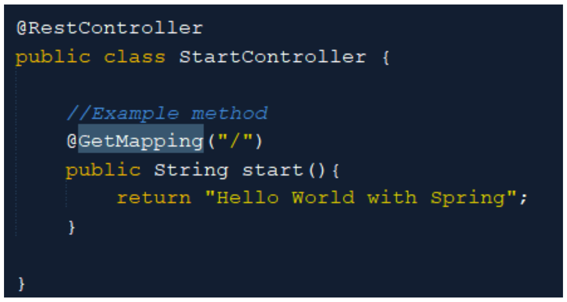


```java
package mx.com.gm;

import org.springframework.web.bind.annotation.RestController;

@RestController
public class StartController {

}
```

Our new class is called StartController.java and it needs to be annotated as a @RestController so that Spring can acknowledge it. Make sure to import the right class (org.springframework.web.bind.annotation.RestController;).

Since @RestController has inherited the @Component annotation, this class will be added to the Spring Container and managed by it.

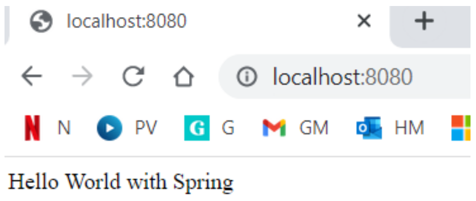
Let’s define a method for our controller class:



```
@RestController
public class StartController {

    //Example method
    @GetMapping("/")
    public String start(){
        return "Hello World with Spring";
    }

}
```

This method will need a unique path (in the URL) in order to be executed, this is known as “Mapping”. Se we will use the @GetMapping() annotation in order to define this path. Implicit in the annotation comes the request type, which is a GET. Since we defined the path to be “/”, this is how we can access this method from our browser:

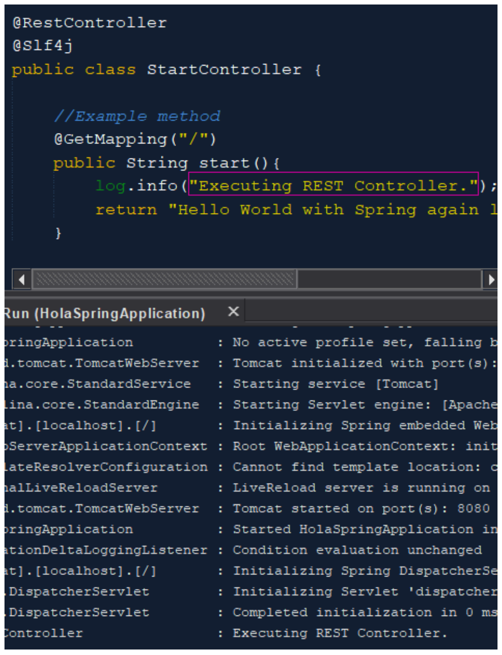
http://localhost:8080/



Hello World with Spring

Thanks to the devtools dependency we added to our project, whenever we save changes into our project, the tomcat server automatically reflects them into our app, a refresh page will do.

Now let’s add some log information into our Controller class, use the @Slf4j annotation belonging to Lombok. Use log.LEVEL To register log info.

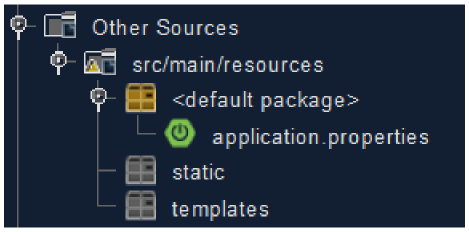


```
@RestController
@Slf4j
public class StartController {

    //Example method
    @GetMapping("/")
    public String start(){
        log.info("Executing REST Controller.");
        return "Hello World with Spring again l
    }
```

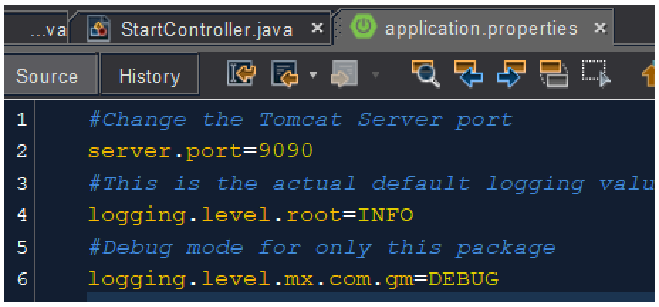
```
pringApplication                  : No active profile set, falling b
d.tomcat.TomcatWebServer          : Tomcat initialized with port(s):
na.core.StandardService           : Starting service [Tomcat]
lina.core.StandardEngine          : Starting Servlet engine: [Apache
at].[localhost].[/]               : Initializing Spring embedded Web
bServerApplicationContext         : Root WebApplicationContext: init
lateResolverConfiguration         : Cannot find template location: c
nalLiveReloadServer               : LiveReload server is running on
d.tomcat.TomcatWebServer          : Tomcat started on port(s): 8080
pringApplication                  : Started HolaSpringApplication in
ationDeltaLoggingListener         : Condition evaluation unchanged
at].[localhost].[/]               : Initializing Spring DispatcherSe
.DispatcherServlet                : Initializing Servlet 'dispatcher
.DispatcherServlet                : Completed initialization in 0 ms
Controller                        : Executing REST Controller.
```

- **Resources Folder:**

There's another important folder called Other Sources. This folder contains the src/main/resources folder, which contains a default package with the application.properties file. This is the Spring configuration file. It's empty by default since all default configs have already been applied. The other packages will contain static files and html templates.

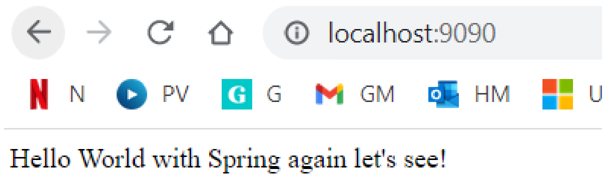


Let's modify some default values in the application.properties file on purpose.

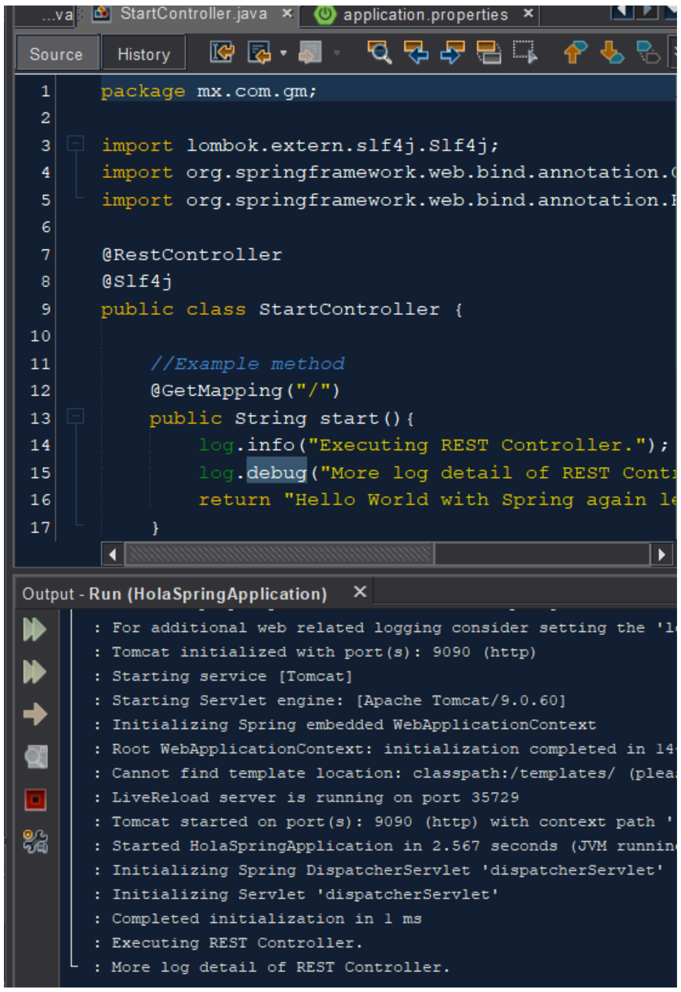


```
#Change the Tomcat Server port
server.port=9090
#This is the actual default logging valu
logging.level.root=INFO
#Debug mode for only this package
logging.level.mx.com.gm=DEBUG
```

Now stop and restart the server, since we changed ports.



Hello World with Spring again let's see!



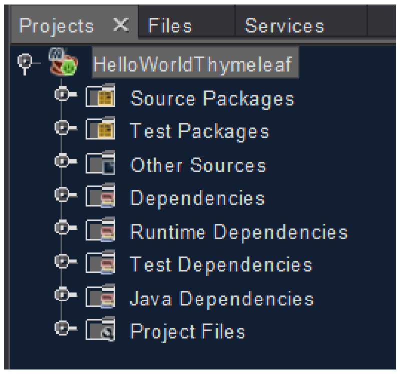
```
package mx.com.gm;

import lombok.extern.slf4j.Slf4j;
import org.springframework.web.bind.annotation.G
import org.springframework.web.bind.annotation.R

@RestController
@Slf4j
public class StartController {

    //Example method
    @GetMapping("/")
    public String start(){
        log.info("Executing REST Controller.");
        log.debug("More log detail of REST Contr
        return "Hello World with Spring again le
    }
```

```
: For additional web related logging consider setting the 'l
: Tomcat initialized with port(s): 9090 (http)
: Starting service [Tomcat]
: Starting Servlet engine: [Apache Tomcat/9.0.60]
: Initializing Spring embedded WebApplicationContext
: Root WebApplicationContext: initialization completed in 14
: Cannot find template location: classpath:/templates/ (plea
: LiveReload server is running on port 35729
: Tomcat started on port(s): 9090 (http) with context path '
: Started HolaSpringApplication in 2.567 seconds (JVM runnin
: Initializing Spring DispatcherServlet 'dispatcherServlet'
: Initializing Servlet 'dispatcherServlet'
: Completed initialization in 1 ms
: Executing REST Controller.
: More log detail of REST Controller.
```

We can now see the new information deployed.

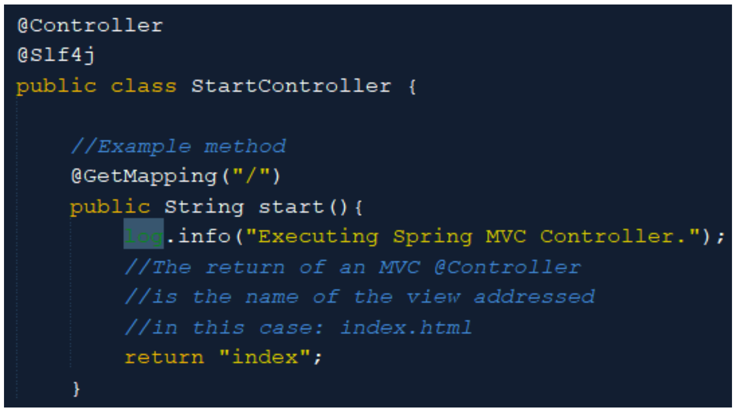
- **Project with Thymeleaf and Spring MVC:**

Copy the previous project and rename it.



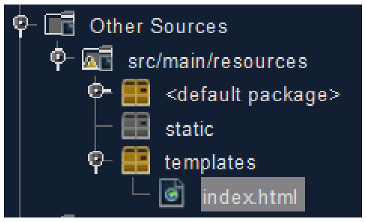
The first change in our project will be to use a simple controller in order to align with Spring MVC and the deployment technology to be used will be Thymeleaf, which uses html views by default.

Use @Controller annotation, which works similarly to @RestController. The main difference is the return type. @Controllers return the name of the view to be deployed.

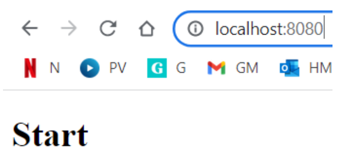


```
@Controller
@Slf4j
public class StartController {

    //Example method
    @GetMapping("/")
    public String start(){
        log.info("Executing Spring MVC Controller.");
        //The return of an MVC @Controller
        //is the name of the view addressed
        //in this case: index.html
        return "index";
    }
```

Create said view in the respective templates package (for Thymeleaf):

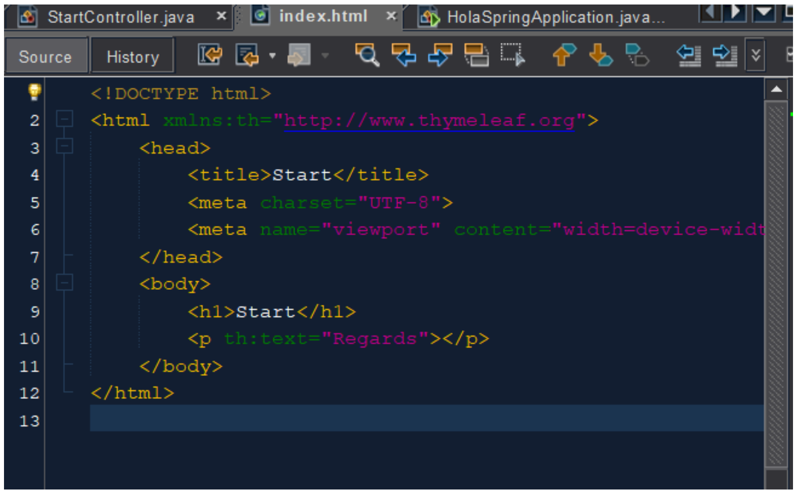


Now browse for the method mapping to get index.html shown:



Start

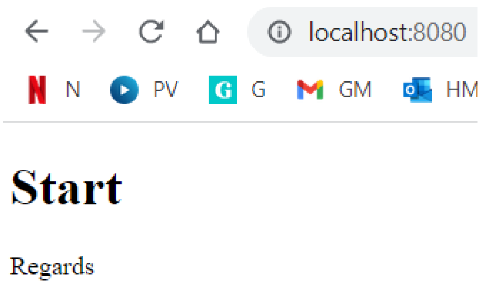
- **Dynamic Info with Thymeleaf:**



```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
    <head>
        <title>Start</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-widt
    </head>
    <body>
        <h1>Start</h1>
        <p th:text="Regards"></p>
    </body>
</html>
```

In order to be able to manage dynamic information in our view, we will need to declare the thymeleaf namespace. Disable HTML error check.

Add a paragraph thymeleaf text to send a message.



```
Start

Regards
```

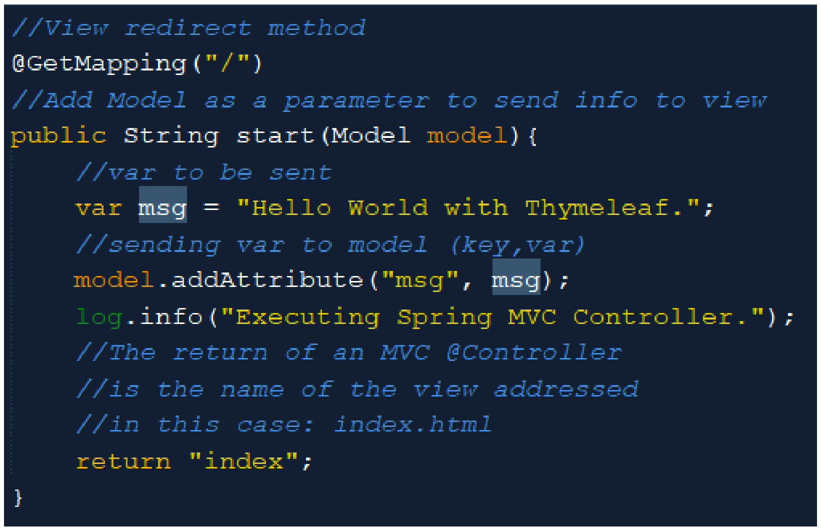
Remember Thymeleaf will automatically store cache for your app. To avoid this, declare it in the application properties.

```
#Avoid Thymeleaf storing cache from your app
spring.thymeleaf.cache=false
```

It's also useful to have Livereaload Chrome extension so that we can automatically refresh anytime we add changes in our code.

Sharing Info to View with Spring MVC and Thymeleaf:

In order to send info to our view, we will need to send the Model parameter into our method.

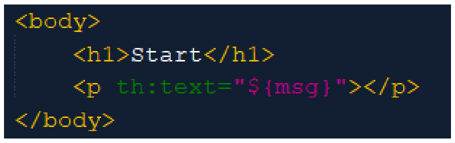


```java
//View redirect method
@GetMapping("/")
//Add Model as a parameter to send info to view
public String start(Model model){
    //var to be sent
    var msg = "Hello World with Thymeleaf.";
    //sending var to model (key,var)
    model.addAttribute("msg", msg);
    log.info("Executing Spring MVC Controller.");
    //The return of an MVC @Controller
    //is the name of the view addressed
    //in this case: index.html
    return "index";
}
```

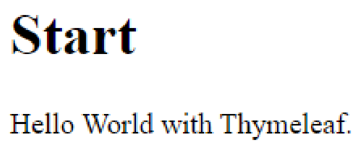
Since Spring introduces dependency injection concept, the moment we declare our class as a @Controller this will be available in the Spring Factory.

When we use @GetMapping() type methods, we can receive the Model argument for this method and use model's methods. This way, Spring factory will instantiate automatically any of the classes we decide to use as Parameters.

Now, we have defined a model attribute and the key we set for it will now be useful to reference it from the view through expression language → ${*key*}.



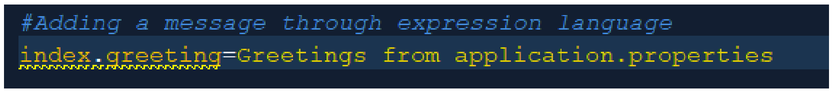
```html
<body>
    <h1>Start</h1>
    <p th:text="${msg}"></p>
</body>
```



Start

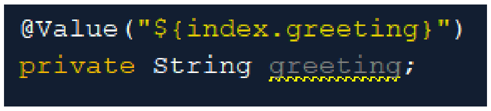
Hello World with Thymeleaf.

We can also use a properties file to set a message:



```
#Adding a message through expression language
index.greeting=Greetings from application.properties
```

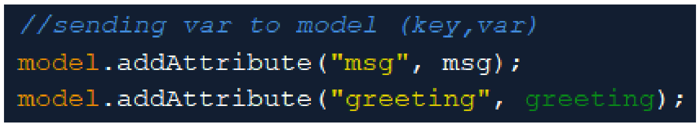
And call it from the Controller, for which we will need to define a variable annotated with the @Value(${*key*}) annotation, make sure to import this class from springfactory.



```
@Value("${index.greeting}")
private String greeting;
```

Through dependency injection, spring factory will assign the value to this newly declared variable.

Now simply add it to the model:

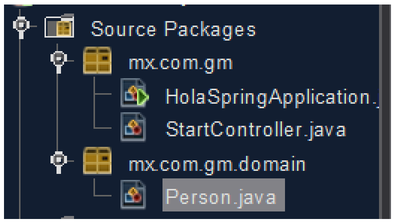


```
//sending var to model (key,var)
model.addAttribute("msg", msg);
model.addAttribute("greeting", greeting);
```

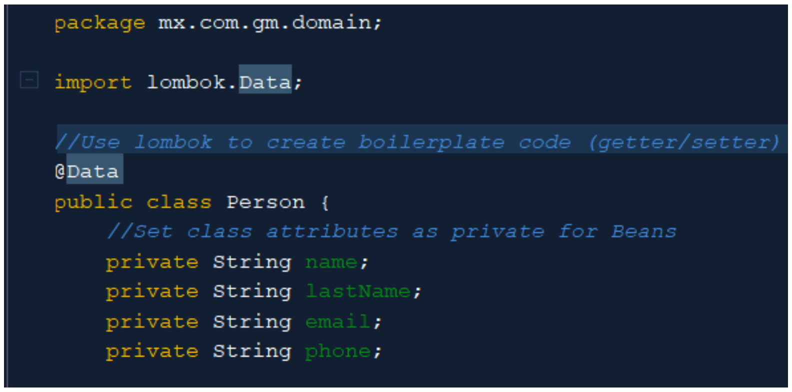
So that we can retrieve it at the view:

- **Domain Class:**

Now we will create a domain class: Person.java, which remember needs to be inside our main package for Spring to scan it and acknowledge it:

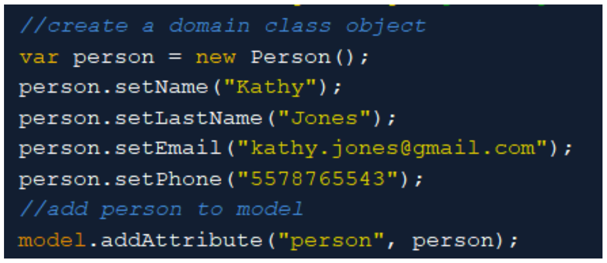


Use Lombok annotation @Data in order to automatically create the boilerplate standard code for Java Beans for our domain class (empty constructor(), getters, setters, toString(), equals() and hashCode() methods).

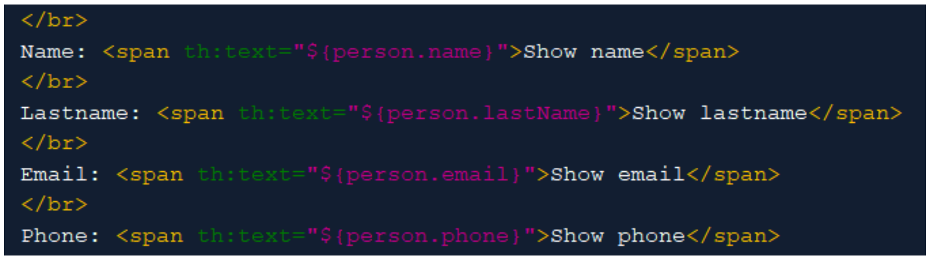


```
package mx.com.gm.domain;

import lombok.Data;

//Use lombok to create boilerplate code (getter/setter)
@Data
public class Person {
    //Set class attributes as private for Beans
    private String name;
    private String lastName;
    private String email;
    private String phone;
```

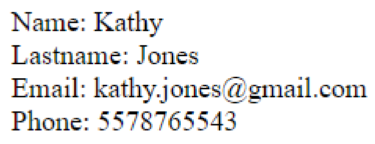
Add the object to the controller so that we can call it from the view:



```
//create a domain class object
var person = new Person();
person.setName("Kathy");
person.setLastName("Jones");
person.setEmail("kathy.jones@gmail.com");
person.setPhone("5578765543");
//add person to model
model.addAttribute("person", person);
```



```
</br>
Name: <span th:text="${person.name}">Show name</span>
</br>
Lastname: <span th:text="${person.lastName}">Show lastname</span>
</br>
Email: <span th:text="${person.email}">Show email</span>
</br>
Phone: <span th:text="${person.phone}">Show phone</span>
```



```
Name: Kathy
Lastname: Jones
Email: kathy.jones@gmail.com
Phone: 5578765543
```

- **Iterating Objects with Thymeleaf:**

We will be creating a list of People (Domain Class Objects) for its iteration.

Watch these two ways to create and fill the array list:

```
//Create an array list to store people
var peopleList = new ArrayList();
//fill array list
peopleList.add(person);
peopleList.add(person2);
//Another notation to create the array list
var peopleList2 = Arrays.asList(person, person2);
```

Now let's add it to the model:

```
//add array list to model
model.addAttribute("peopleList",peopleList);
```

And use it to display info in the view:

```
<!-- Validate if array is not empty -->
<div th:if="${peopleList != null and !peopleList.empty}">
    <table border="1">
        <tr>
            <th>Name</th>
            <th>Lastname</th>
            <th>E-Mail</th>
            <th>Phone</th>
        </tr>
        <!-- Using Thymeleaf to iterate through array -->
        <tr th:each="person : ${peopleList}">
            <td th:text="${person.name}">Show name</td>
            <td th:text="${person.lastName}">Show lastname</td>
            <td th:text="${person.email}">Show email</td>
            <td th:text="${person.phone}">Show phone</td>
        </tr>
    </table>
</div>
<div th:if="${peopleList == null and peopleList.empty}">
    The list of people is empty.
</div>
```

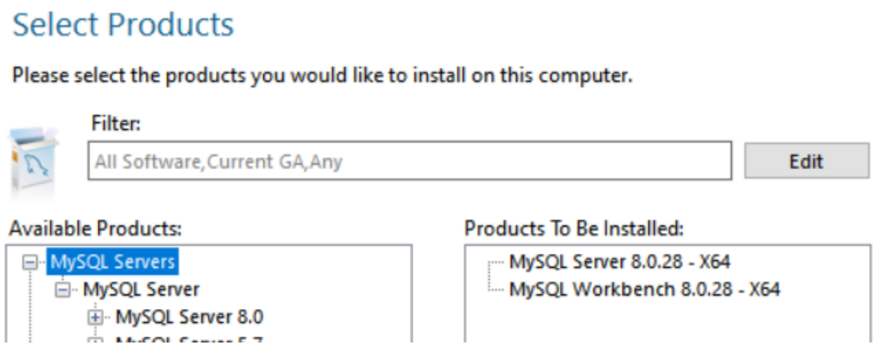
We used a table to display the information in the view side. Thymeleaf can iterate through arrays by using the *"each"* label. Assign a variable name to the name of the array you sent to the model.

Name	Lastname	E-Mail	Phone
Kathy	Jones	kathy.jones@gmail.com	5578765543
Mark	Twain	mark.twain@gmail.com	5543556543

- **MySQL:**

Download MySQL Community Server: https://dev.mysql.com/downloads/mysql/

Customize your installation to include the relevant software only The Database Manager (MySQL Server) and its IDE (MySQL Workbench). Select Custom Installation:



Select Products

Please select the products you would like to install on this computer.

Filter: All Software,Current GA,Any

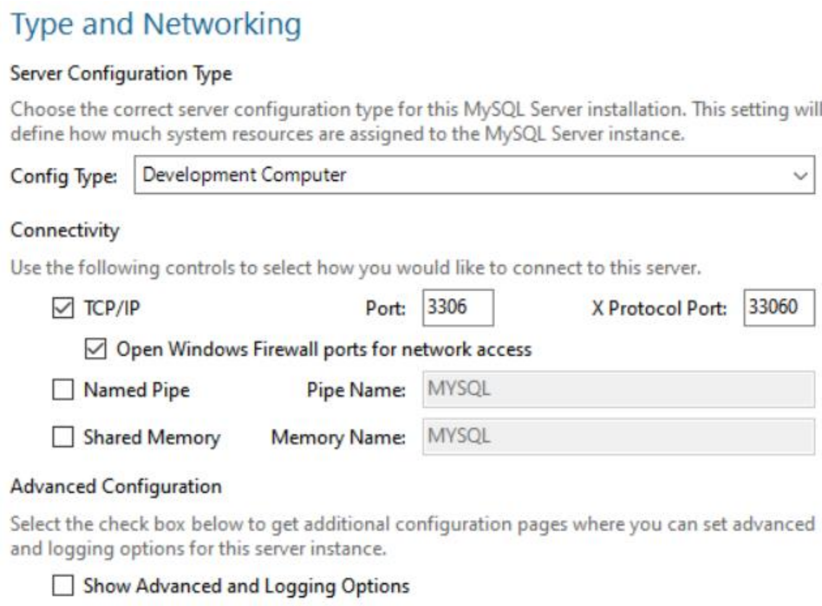
Available Products:

- MySQL Servers
  - MySQL Server
    - MySQL Server 8.0

Products To Be Installed:

- MySQL Server 8.0.28 - X64
- MySQL Workbench 8.0.28 - X64

When configuring the DB Server, take note of relevant data such as the port assigned and password:



Type and Networking

Server Configuration Type

Choose the correct server configuration type for this MySQL Server installation. This setting will define how much system resources are assigned to the MySQL Server instance.

Config Type: Development Computer

Connectivity

Use the following controls to select how you would like to connect to this server.

☑ TCP/IP Port: 3306 X Protocol Port: 33060

☑ Open Windows Firewall ports for network access

☐ Named Pipe Pipe Name: MYSQL

☐ Shared Memory Memory Name: MYSQL

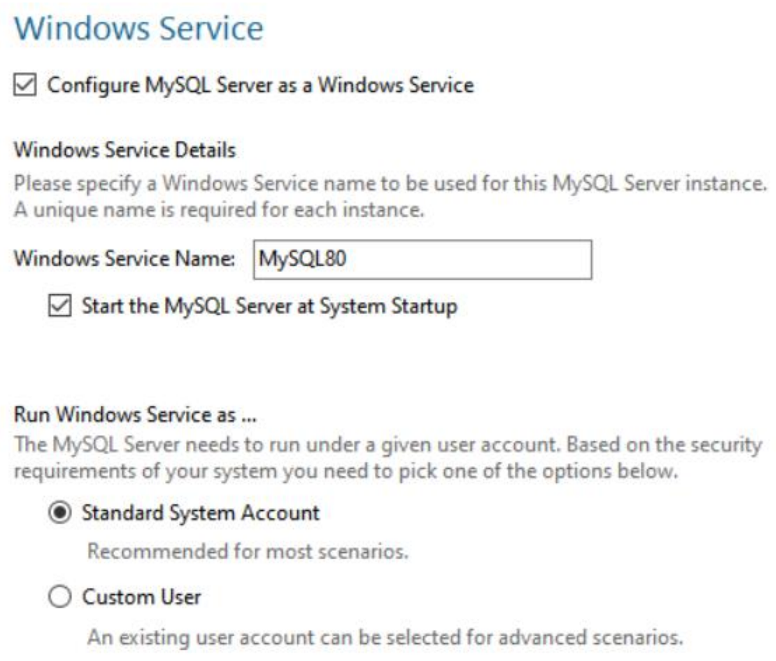
Advanced Configuration

Select the check box below to get additional configuration pages where you can set advanced and logging options for this server instance.

☐ Show Advanced and Logging Options

Root password: admin

This option will allow for the DB to be automatically started when we start Windows.



Windows Service

☑ Configure MySQL Server as a Windows Service

Windows Service Details

Please specify a Windows Service name to be used for this MySQL Server instance.
A unique name is required for each instance.

Windows Service Name: MySQL80

☑ Start the MySQL Server at System Startup

Run Windows Service as ...

The MySQL Server needs to run under a given user account. Based on the security requirements of your system you need to pick one of the options below.

● Standard System Account

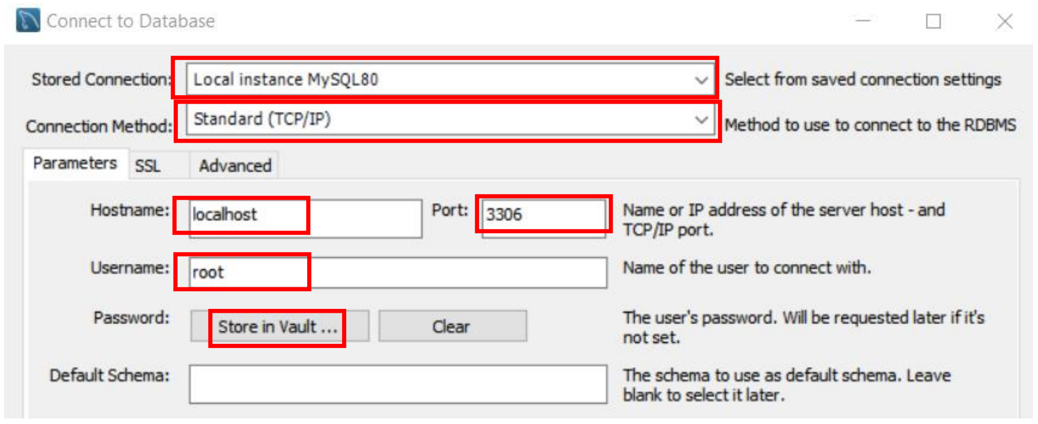
Recommended for most scenarios.

○ Custom User

An existing user account can be selected for advanced scenarios.

If you had trouble in any of these steps, please disable antivirus software that might be interfering with the setup.

Now in the MySQL IDE, go to Database → Connect to DB → Select Local instance → Input known data → Store in vault and input root user password → OK.



Connect to Database

Stored Connection: Local instance MySQL80 — Select from saved connection settings

Connection Method: Standard (TCP/IP) — Method to use to connect to the RDBMS

Parameters | SSL | Advanced

Hostname: localhost Port: 3306 — Name or IP address of the server host - and TCP/IP port.

Username: root — Name of the user to connect with.

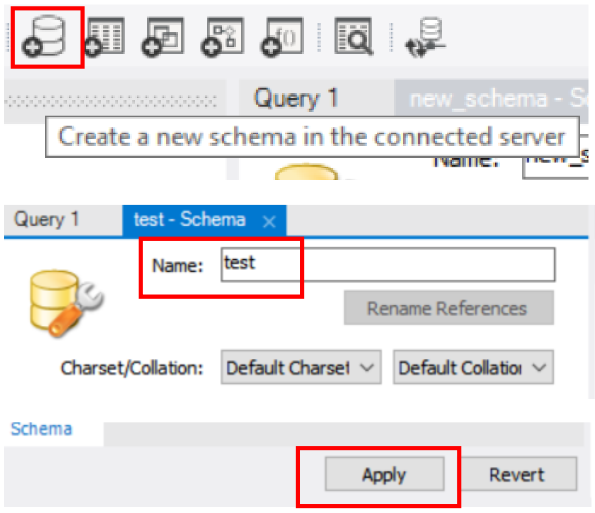
Password: Store in Vault ... Clear — The user's password. Will be requested later if it's not set.

Default Schema: — The schema to use as default schema. Leave blank to select it later.

The selected Schema will be the one by default. A Schema is basically a DB in MySQL.

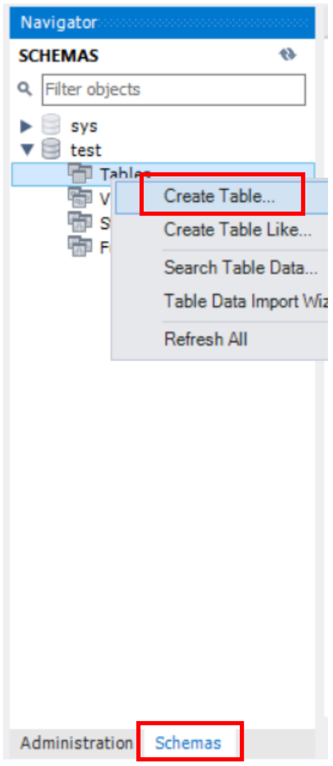
- **MySQL – Creating DB:**

Go to the Create a new Schema icon and click it:



Give the DB a name and Apply → Apply → Finish.

Now from the Schemas Tab, create a new table:



Now let's populate the DB with a table with the data that the Person class holds:

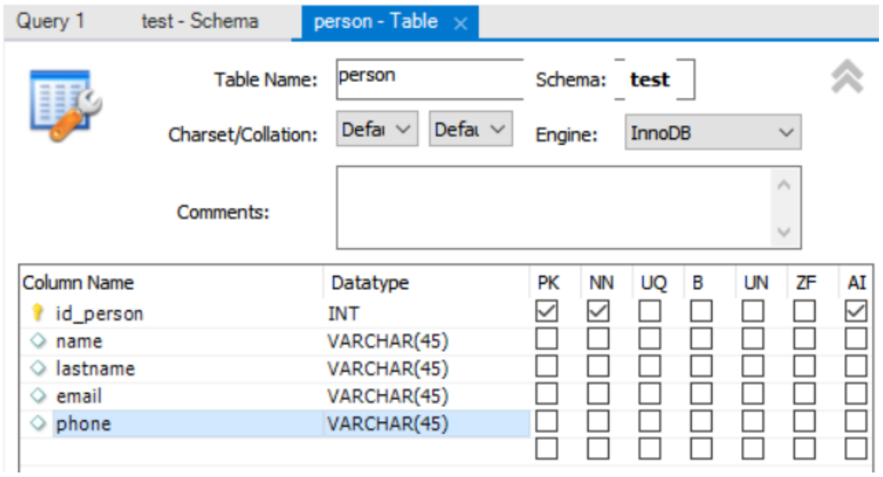
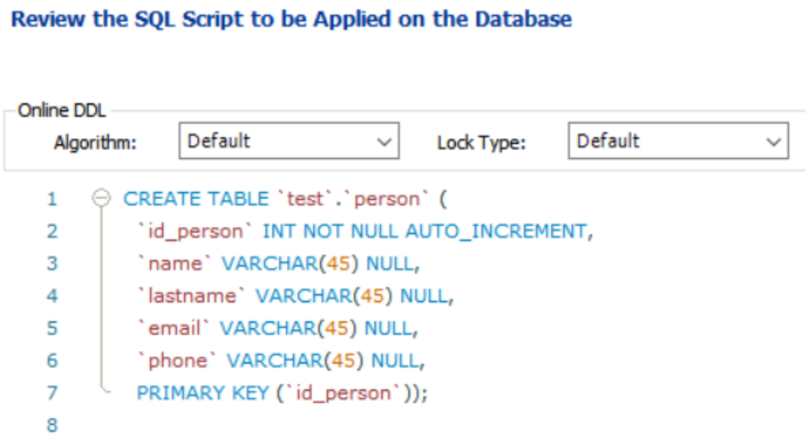


Table Name: person
Schema: test
Charset/Collation: Defa Defa
Engine: InnoDB

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI
id_person	INT	✓	✓					✓
name	VARCHAR(45)							
lastname	VARCHAR(45)							
email	VARCHAR(45)							
phone	VARCHAR(45)							

The first column will be the Primary Key, it has to be Not Null and since the Id needs to be different for each row, set it as Auto-Incrementable.

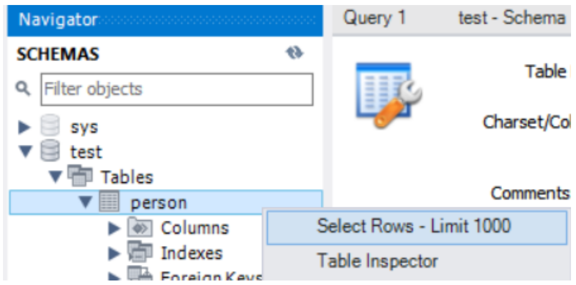
Hit Apply and the IDE will automatically generate the SQL code to be applied as a script to the DB. Hit Finish.

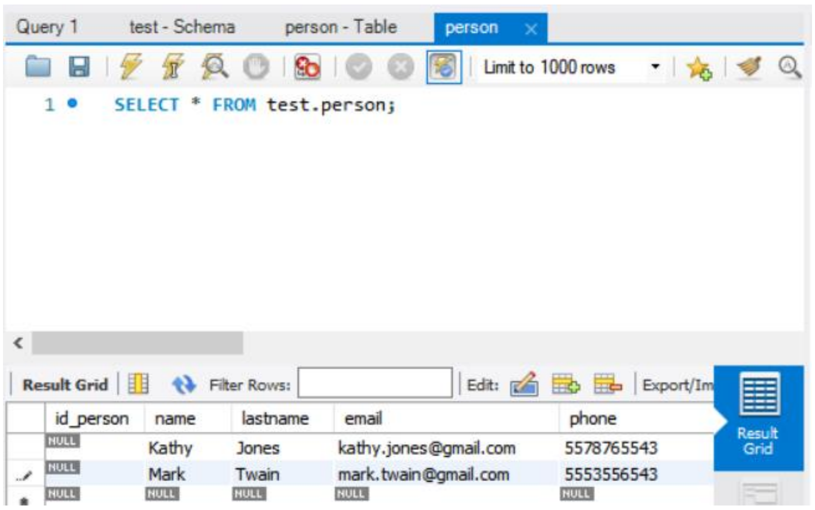


Algorithm: Default
Lock Type: Default

```
CREATE TABLE `test`.`person` (
  `id_person` INT NOT NULL AUTO_INCREMENT,
  `name` VARCHAR(45) NULL,
  `lastname` VARCHAR(45) NULL,
  `email` VARCHAR(45) NULL,
  `phone` VARCHAR(45) NULL,
  PRIMARY KEY (`id_person`));
```

We can now see the table created. Hit Select Rows in order to populate the table:





```
SELECT * FROM test.person;
```

id_person	name	lastname	email	phone
NULL	Kathy	Jones	kathy.jones@gmail.com	5578765543
NULL	Mark	Twain	mark.twain@gmail.com	5553556543
NULL	NULL	NULL	NULL	NULL

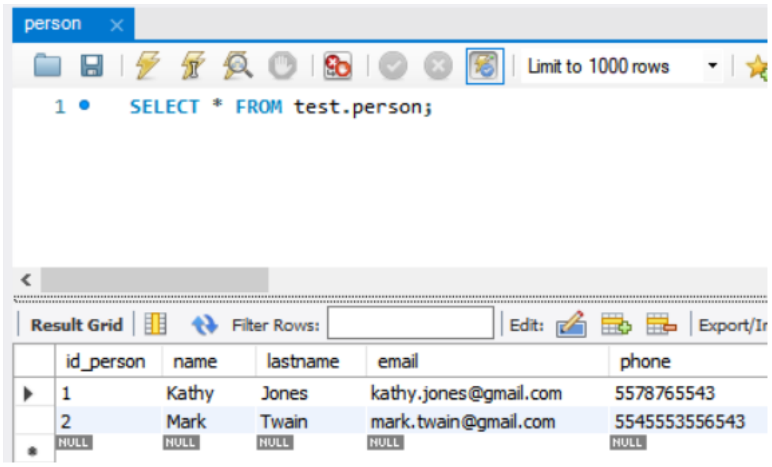
Add some example info and hit Apply.

SQL:

INSERT INTO `test`.`person` (`name`, `lastname`, `email`, `phone`) VALUES ('Kathy', 'Jones', 'kathy.jones@gmail.com', '5578765543');

INSERT INTO `test`.`person` (`name`, `lastname`, `email`, `phone`) VALUES ('Mark', 'Twain', 'mark.twain@gmail.com', '5545553556543');

Close the person table tab and open it again to watch the table updated, notice an ID has been assigned to each row:

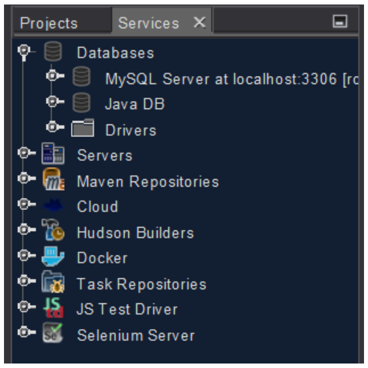


```
SELECT * FROM test.person;
```

id_person	name	lastname	email	phone
1	Kathy	Jones	kathy.jones@gmail.com	5578765543
2	Mark	Twain	mark.twain@gmail.com	5545553556543
NULL	NULL	NULL	NULL	NULL

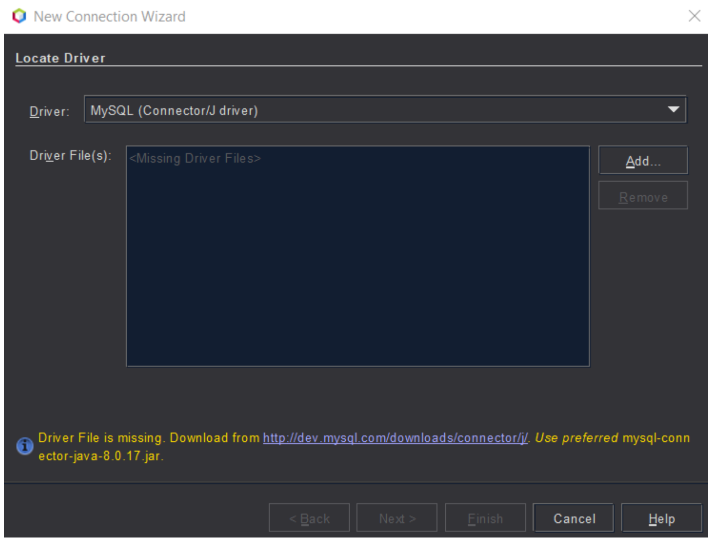
- **MySQL – Connect from Netbeans:**

From Netbeans go to Window → Services → Databases → Right click → New Connection:



Select MySQL Driver, if it's missing, download it from the URL provided by the IDE:

https://dev.mysql.com/downloads/connector/j/

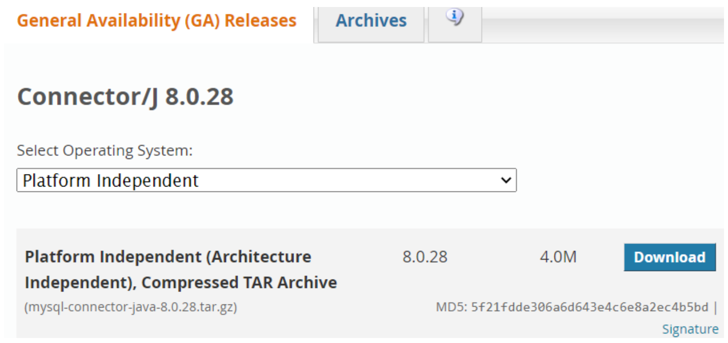


Driver: MySQL (Connector/J driver)

Driver File(s): <Missing Driver Files>

Driver File is missing. Download from http://dev.mysql.com/downloads/connector/j/. *Use preferred* mysql-connector-java-8.0.17.jar.

Select the Platform Independent OS:

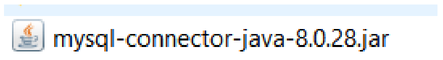


Connector/J 8.0.28

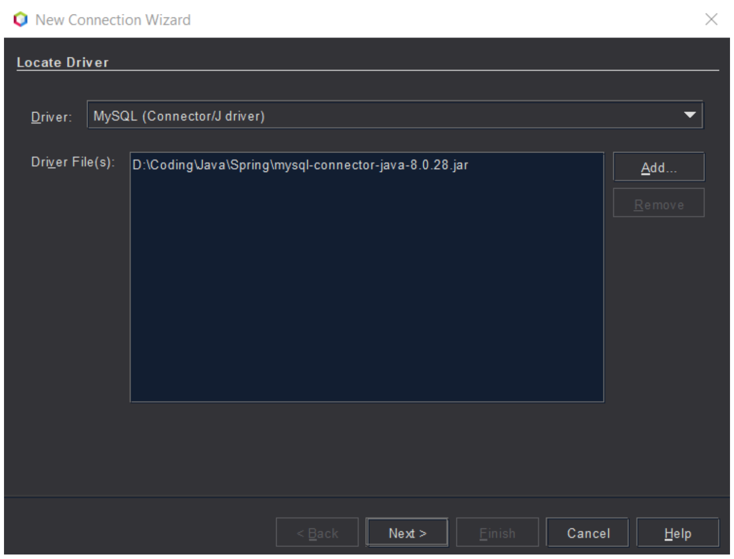
Select Operating System: Platform Independent

Platform Independent (Architecture Independent), Compressed TAR Archive (mysql-connector-java-8.0.28.tar.gz)	8.0.28	4.0M	Download
	MD5: 5f21fdde306a6d643e4c6e8a2ec4b5bd | Signature		

Retrieve the jar file from the zip file and store it in your local:



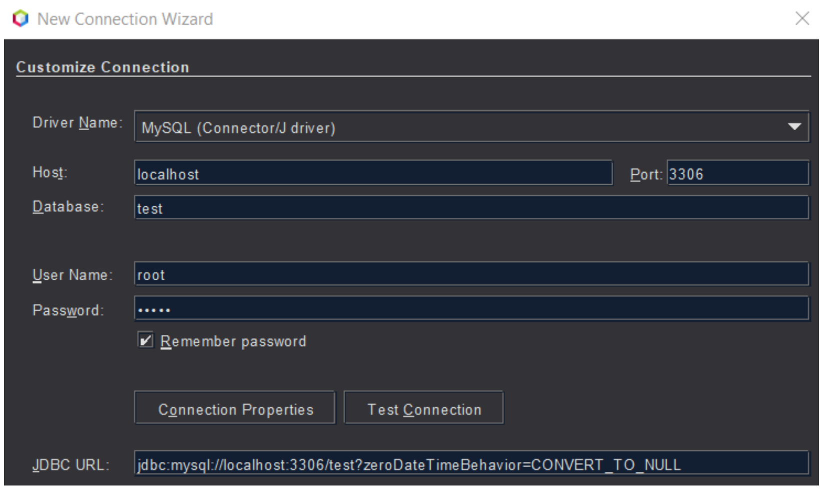
Now Add the driver into the Netbeans Connection Wizard:



Driver: MySQL (Connector/J driver)

Driver File(s): D:\Coding\Java\Spring\mysql-connector-java-8.0.28.jar

Hit Next and enter your DB information:



Driver Name: MySQL (Connector/J driver)

Host: localhost Port: 3306

Database: test

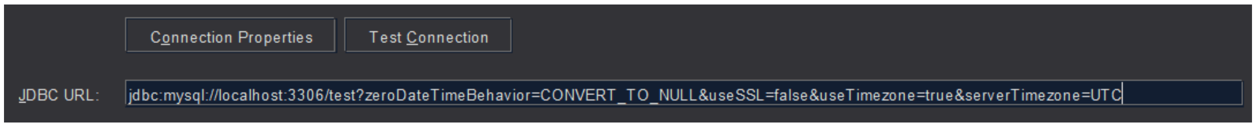
User Name: root

Password: •••••

☑ Remember password

JDBC URL: jdbc:mysql://localhost:3306/test?zeroDateTimeBehavior=CONVERT_TO_NULL

We will modify the JDBC URL a little bit by adding some parameters with &:
jdbc:mysql://localhost:3306/test?zeroDateTimeBehavior=CONVERT_TO_NULL&useSSL=false&useTimezone=true&serverTimezone=UTC

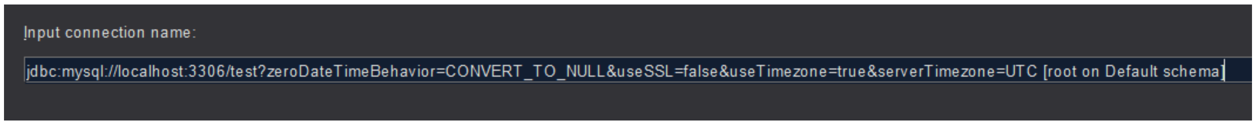


JDBC URL: jdbc:mysql://localhost:3306/test?zeroDateTimeBehavior=CONVERT_TO_NULL&useSSL=false&useTimezone=true&serverTimezone=UTC

Hit Test Connection to make sure everything is ok:

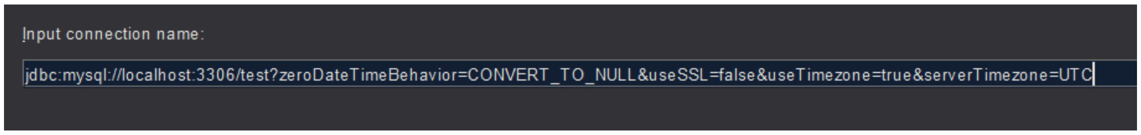
Connection Succeeded.

Hit Next, and set the connection name right:



Input connection name:

```
jdbc:mysql://localhost:3306/test?zeroDateTimeBehavior=CONVERT_TO_NULL&useSSL=false&useTimezone=true&serverTimezone=UTC [root on Default schema]
```

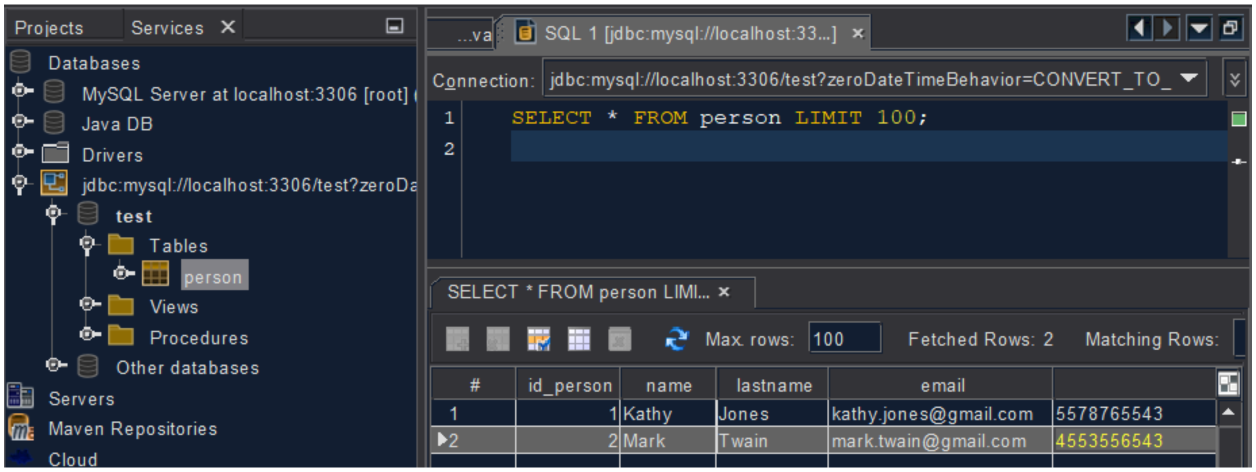


Input connection name:

```
jdbc:mysql://localhost:3306/test?zeroDateTimeBehavior=CONVERT_TO_NULL&useSSL=false&useTimezone=true&serverTimezone=UTC
```

Now hit Finish.

We can now see our DB info from Netbeans:



Connection: jdbc:mysql://localhost:3306/test?zeroDateTimeBehavior=CONVERT_TO_

```
SELECT * FROM person LIMIT 100;
```

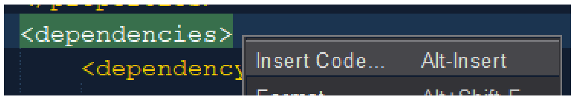
#	id_person	name	lastname	email	
1	1	Kathy	Jones	kathy.jones@gmail.com	5578765543
2	2	Mark	Twain	mark.twain@gmail.com	4553556543

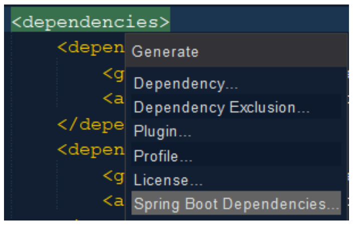
Right click on person table → View Data.

- **Spring Data:**

In order to work with our DB dynamically, the first thing we need to do is to add the missing dependencies (Spring Data and MySQL Driver) into our pom file:

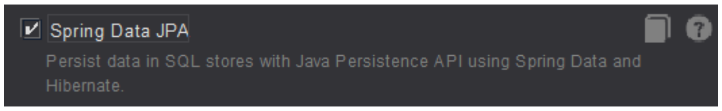
The Spring Wizard can help us with this taks. Go to the dependencies tag in your pom file:



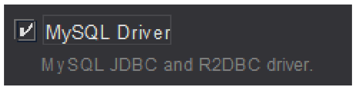


Right click → Insert Code… → Spring Boot Dependencies…

Select Spring Data JPA and MySQL Driver and hit OK:



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



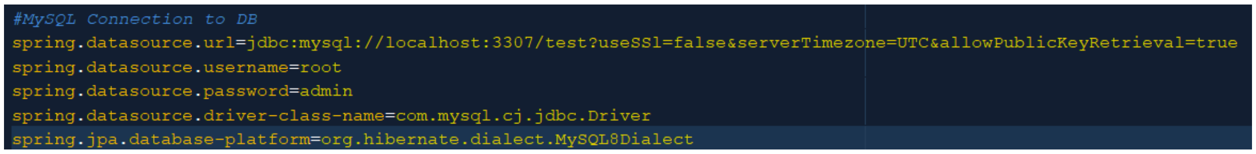
[✓ MySQL Driver]
MySQL JDBC and R2DBC driver.

Both dependencies have been added to our pom file. The java connector version will automatically be managed by Spring. Remember to right click on your project and hit Clean and Build in order for the new dependencies to be downloaded.

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

- **Spring Data – Configure Connection to DB:**

Go to the application.properties file in your Spring project and let's feed it all of the DB information:

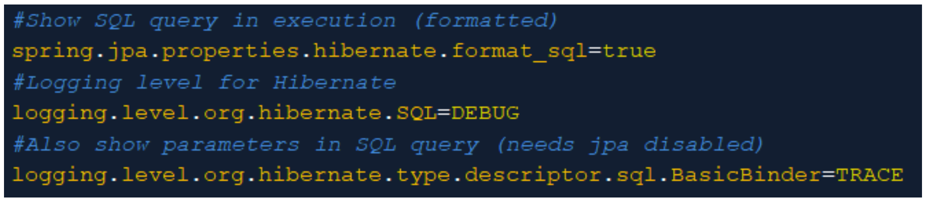


```
#MySQL Connection to DB
spring.datasource.url=jdbc:mysql://localhost:3307/test?useSSl=false&serverTimezone=UTC&allowPublicKeyRetrieval=true
spring.datasource.username=root
spring.datasource.password=admin
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect
```

Note: If your port is 3306 (default) you don't need to write it, otherwise you do.

The JDBC URL built for MySQL8 requires certain parameters that are concatenated with &.

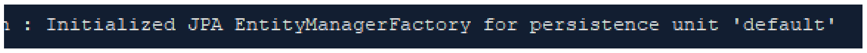
Remember the Spring default for JPA is Hibernate, so if we want our queries in execution to be shown, we need to define a hibernate property:



```
#Show SQL query in execution (formatted)
spring.jpa.properties.hibernate.format_sql=true
#Logging level for Hibernate
logging.level.org.hibernate.SQL=DEBUG
#Also show parameters in SQL query (needs jpa disabled)
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE
```

The first property shows the query in an SQL format. The second property allows us to actually visualize the query. The third property is in TRACE mode, the most basic mode, and it will allow us to see the query parameters (as long as the first property is disabled).

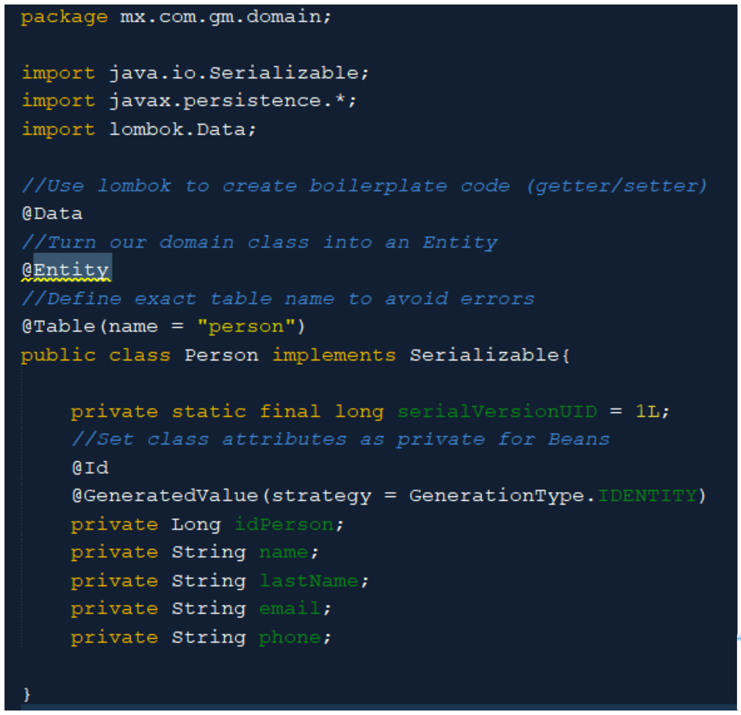
Clean and Build project to check for errors.



```
n : Initialized JPA EntityManagerFactory for persistence unit 'default'
```

We can appreciate that JPA EntityManagerFactory was automatically initiated, and a persistence unit "default" was also automatically created, so a persistence.xml file will not be necessary.

Now let’s turn our domain class into an Entity, use javax.persistence libraries, not hibernate. Everything will be managed via JPA, hibernate itself will only be implemented through JPA.



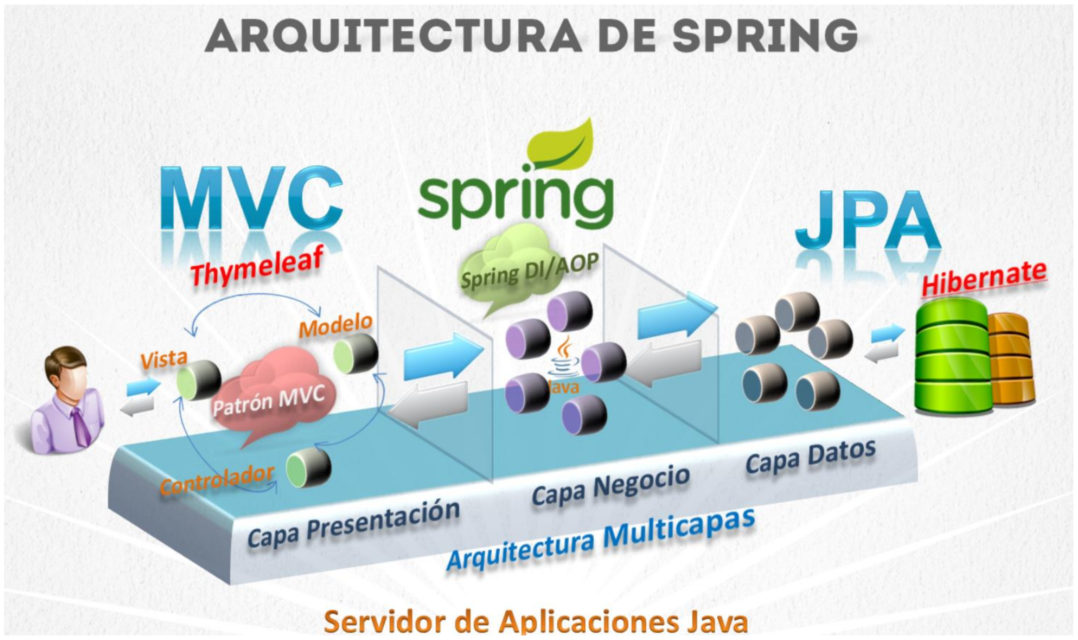
```java
package mx.com.gm.domain;

import java.io.Serializable;
import javax.persistence.*;
import lombok.Data;

//Use lombok to create boilerplate code (getter/setter)
@Data
//Turn our domain class into an Entity
@Entity
//Define exact table name to avoid errors
@Table(name = "person")
public class Person implements Serializable{

    private static final long serialVersionUID = 1L;
    //Set class attributes as private for Beans
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long idPerson;
    private String name;
    private String lastName;
    private String email;
    private String phone;

}
```

Our domain class will now be annotated with @Entity, to be defined as a persistence domain object, which basically represents a table in a DB. Each entity instance will represent a row in it. The @Table annotation is good to clarify the real Table name, since it’s not the exact same name as the one defined in the Schema, it could lead to errors.

Implement Serializable interface for data transfer and storage and define the UID for it.

Finally, we need to provide que class or table id, use the @Id annotation above the variable that will be used as the table Primary Key, in this case: idPerson, and provide a method for its generation via the @GeneratedValue() annotation.

- **Spring Data – DAO Classes:**



There are several ways to define Spring Architecture.

Let's say we have a Multilayer architecture (logical layers) which lives inside an App Server (Tomcat).

First of all, we have our Presentation Layer. We have chosen MVC model and implemented it through Spring. We use Thymeleaf as the display technology. Our View is presented via HTML. Our model is the Person class. Our Controller has also been created to reference our view.

Second, comes the Business Layer. We will interact with the transactional concept here later.

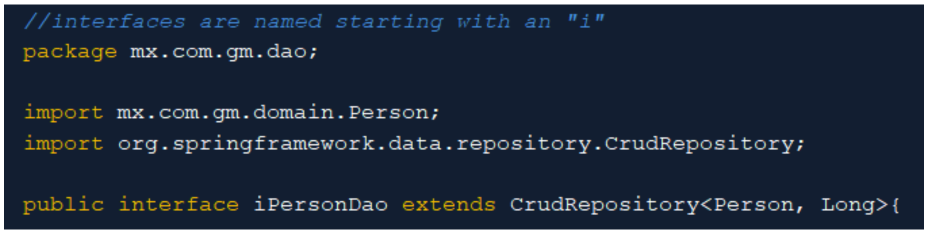
Third and last, comes the Data Layer. We are using JPA to implement hibernate in order to connect to our MySQL Database. We have created Entity classes or tables to withhold our data.

In order to retrieve info directly from the Presentation layer View, we will need Spring Repositories or DAO (Data Access Object) classes.

In the past, it was necessary to create an interface with the CRUD methods and then create a class with the @Repository annotation to have it implement this interface.

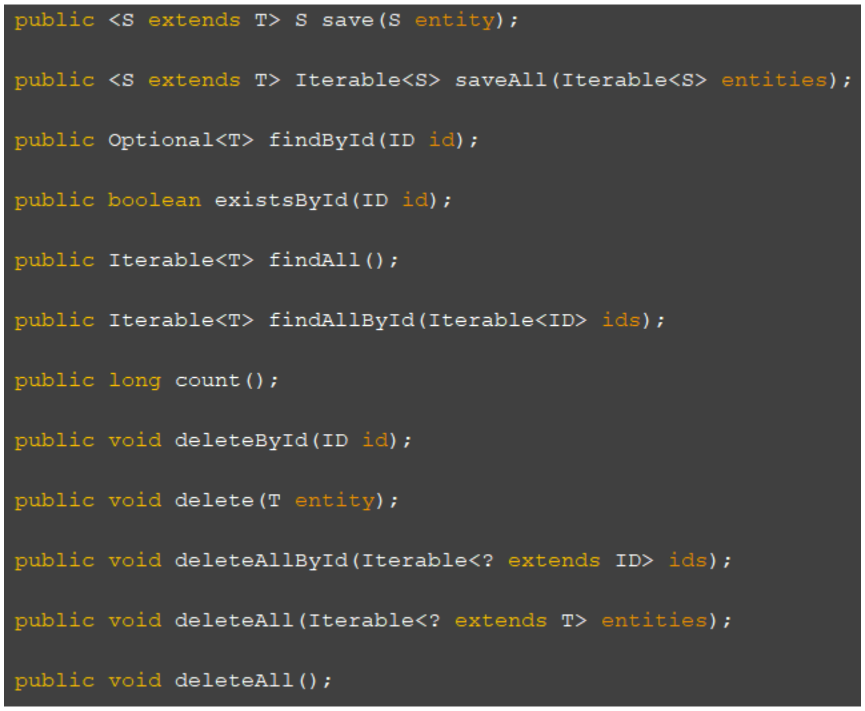
Nowadays, we create an interface which extends from CrudRepository<*Entity Type to manage*, *Primary Key Type*>.

This way, Spring boot will create a default implementation, which means we no longer have to create a class to implement this interface. Remember it's best practice to start interface names with "i".



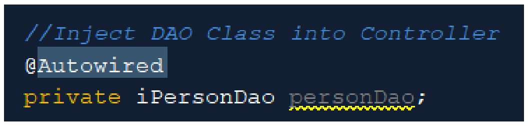
```
//interfaces are named starting with an "i"
package mx.com.gm.dao;

import mx.com.gm.domain.Person;
import org.springframework.data.repository.CrudRepository;

public interface iPersonDao extends CrudRepository<Person, Long>{
```

On the inside, CrudRepository interface contains all the classic methods for Entities:



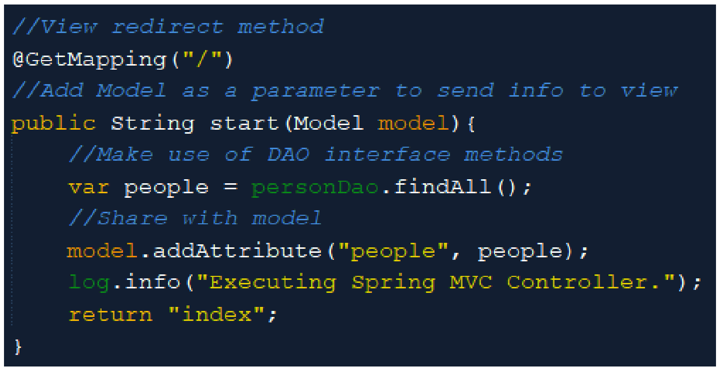
```
public <S extends T> S save(S entity);

public <S extends T> Iterable<S> saveAll(Iterable<S> entities);

public Optional<T> findById(ID id);

public boolean existsById(ID id);

public Iterable<T> findAll();

public Iterable<T> findAllById(Iterable<ID> ids);

public long count();

public void deleteById(ID id);

public void delete(T entity);

public void deleteAllById(Iterable<? extends ID> ids);

public void deleteAll(Iterable<? extends T> entities);

public void deleteAll();
```

Now inject this interface into the Controller by using the @Autowired annotation:



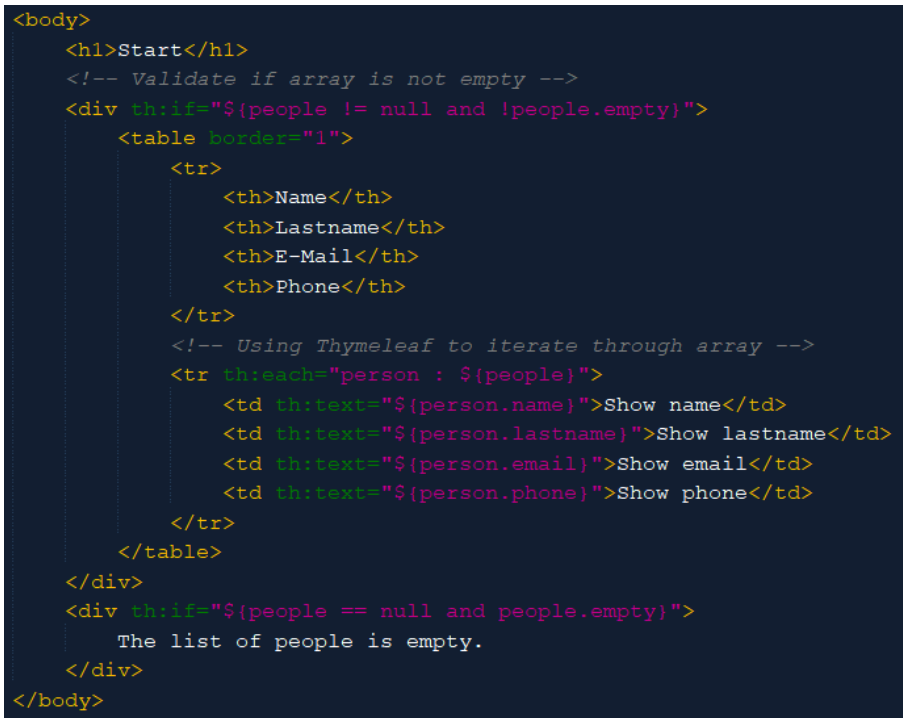
```
//Inject DAO Class into Controller
@Autowired
private iPersonDao personDao;
```

Now let's make use of our iPersonDao available methods:



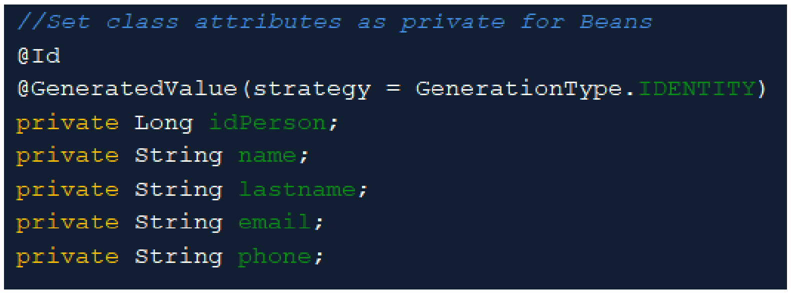
```java
//View redirect method
@GetMapping("/")
//Add Model as a parameter to send info to view
public String start(Model model){
    //Make use of DAO interface methods
    var people = personDao.findAll();
    //Share with model
    model.addAttribute("people", people);
    log.info("Executing Spring MVC Controller.");
    return "index";
}
```

And finally let's take these changes to our View:

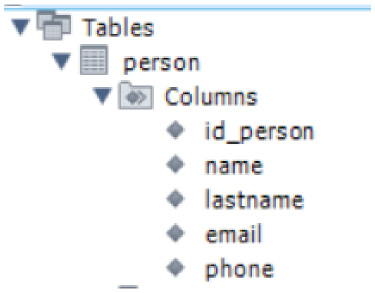


```html
<body>
    <h1>Start</h1>
    <!-- Validate if array is not empty -->
    <div th:if="${people != null and !people.empty}">
        <table border="1">
            <tr>
                <th>Name</th>
                <th>Lastname</th>
                <th>E-Mail</th>
                <th>Phone</th>
            </tr>
            <!-- Using Thymeleaf to iterate through array -->
            <tr th:each="person : ${people}">
                <td th:text="${person.name}">Show name</td>
                <td th:text="${person.lastname}">Show lastname</td>
                <td th:text="${person.email}">Show email</td>
                <td th:text="${person.phone}">Show phone</td>
            </tr>
        </table>
    </div>
    <div th:if="${people == null and people.empty}">
        The list of people is empty.
    </div>
</body>
```

Remember every single column on the Schema (except for PK) should match the domain class attributes with the exact same name:

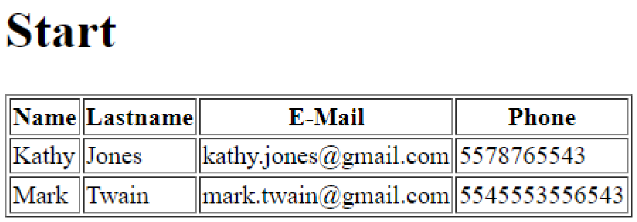


```
//Set class attributes as private for Beans
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long idPerson;
private String name;
private String lastname;
private String email;
private String phone;
```



```
Tables
  person
    Columns
      id_person
      name
      lastname
      email
      phone
```

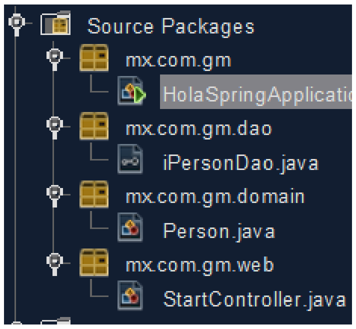
Now run the application to view the results in your browser:



Start

Name	Lastname	E-Mail	Phone
Kathy	Jones	kathy.jones@gmail.com	5578765543
Mark	Twain	mark.twain@gmail.com	5545553556543

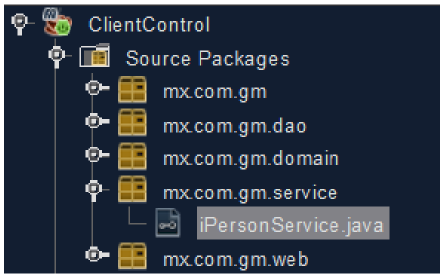
So far, our project is organized into different container packages, like this:



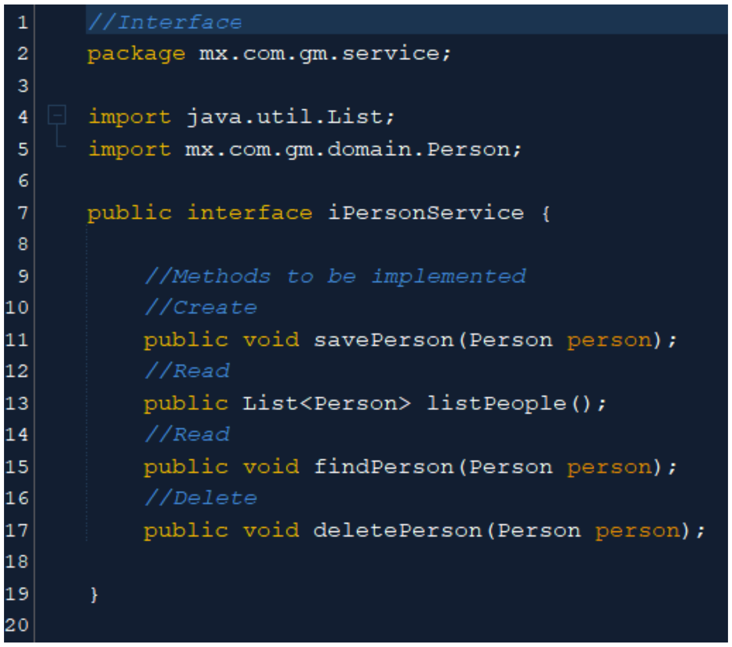
```
Source Packages
  mx.com.gm
    HolaSpringApplicati
  mx.com.gm.dao
    iPersonDao.java
  mx.com.gm.domain
    Person.java
  mx.com.gm.web
    StartController.java
```

- **Spring – Service/Business Layer:**

For the Service Layer, we need to follow the client's requirements or Business Rules. We will create a new package in our project for our services:



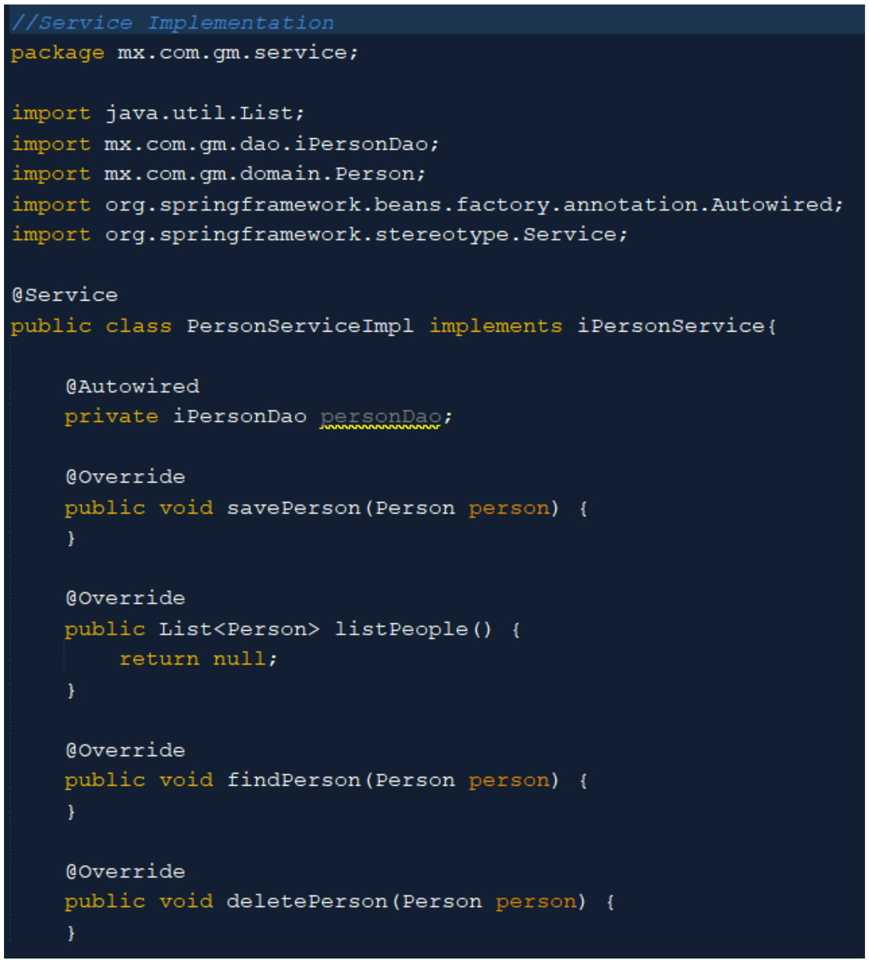
Service classes are not actually classes but interfaces, this interface will contain the contract and the methods to be implemented.



```java
//Interface
package mx.com.gm.service;

import java.util.List;
import mx.com.gm.domain.Person;

public interface iPersonService {

    //Methods to be implemented
    //Create
    public void savePerson(Person person);
    //Read
    public List<Person> listPeople();
    //Read
    public void findPerson(Person person);
    //Delete
    public void deletePerson(Person person);

}

```

Now we will need a class to implement this new interface.

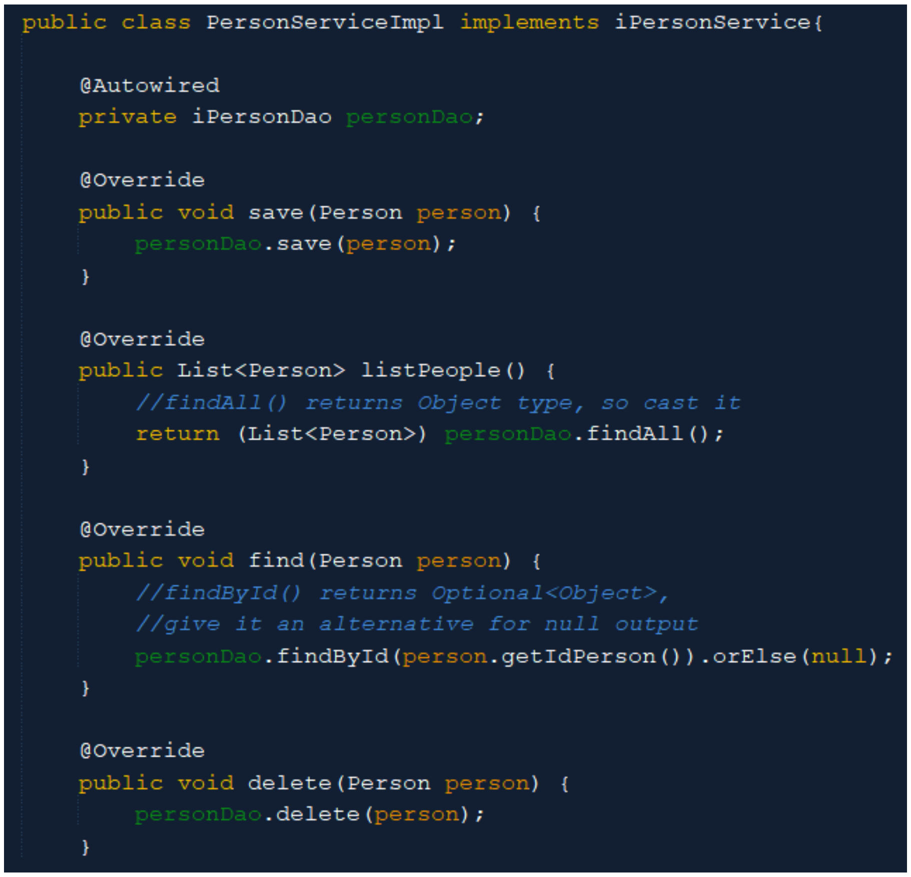
The class that implements the interface will be called the same but with the Impl keyword attached at the end. This class needs to be annotated with @Service so that it can later be injected as the service implementation inside our controller.



```java
//Service Implementation
package mx.com.gm.service;

import java.util.List;
import mx.com.gm.dao.iPersonDao;
import mx.com.gm.domain.Person;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class PersonServiceImpl implements iPersonService{

    @Autowired
    private iPersonDao personDao;

    @Override
    public void savePerson(Person person) {
    }

    @Override
    public List<Person> listPeople() {
        return null;
    }

    @Override
    public void findPerson(Person person) {
    }

    @Override
    public void deletePerson(Person person) {
    }
```

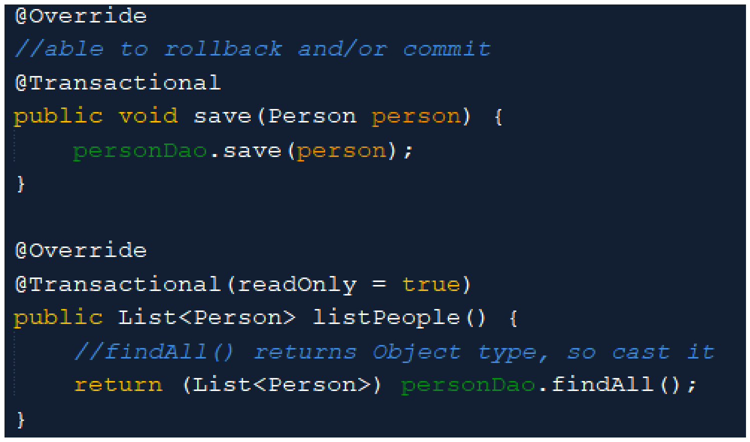
We will inject our Dao class inside this service implementation (@Autowired). This way, our controller won’t have to interact directly with the Data Layer, but instead it will interact with the Business Layer. This same class will connect Business Layer to Data Layer.

Our interface methods have all been defined in our Implementation class.



```java
public class PersonServiceImpl implements iPersonService{

    @Autowired
    private iPersonDao personDao;

    @Override
    public void save(Person person) {
        personDao.save(person);
    }

    @Override
    public List<Person> listPeople() {
        //findAll() returns Object type, so cast it
        return (List<Person>) personDao.findAll();
    }

    @Override
    public void find(Person person) {
        //findById() returns Optional<Object>,
        //give it an alternative for null output
        personDao.findById(person.getIdPerson()).orElse(null);
    }

    @Override
    public void delete(Person person) {
        personDao.delete(person);
    }
```

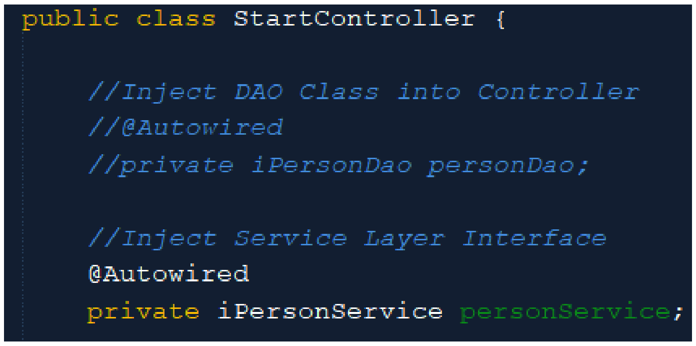
But we are still missing something. All of these methods will be transactional, meaning they will make queries to one or more tables. In case of error, a rollback will have to be applied, and in case of success, a commit will be required to finish the job.



```java
@Override
//able to rollback and/or commit
@Transactional
public void save(Person person) {
    personDao.save(person);
}

@Override
@Transactional(readOnly = true)
public List<Person> listPeople() {
    //findAll() returns Object type, so cast it
    return (List<Person>) personDao.findAll();
}
```

Every method will be annotated with @Transactional annotation. Be careful to import it from springframework. Read only methods will be specified as such, while methods that actually represent changes in the DB, will not be annotated with parameters, meaning a transaction will remain open, being able to rollback and/or commit changes.

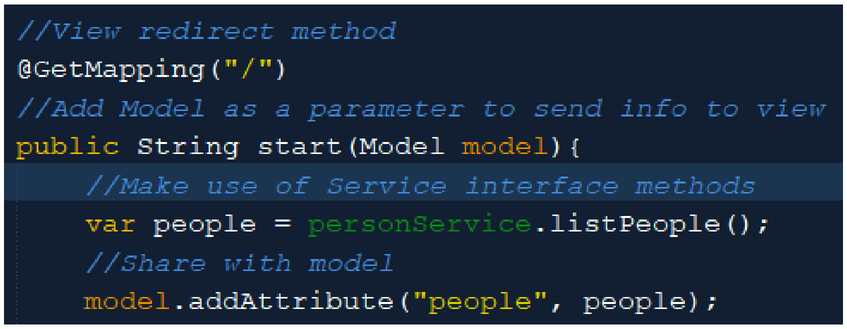
Back at our controller, remember we had previously injected a DAO (Data Layer) Class directly into our controller, which is insecure. We will now inject a Service Layer Interface.



```
public class StartController {

    //Inject DAO Class into Controller
    //@Autowired
    //private iPersonDao personDao;

    //Inject Service Layer Interface
    @Autowired
    private iPersonService personService;
```

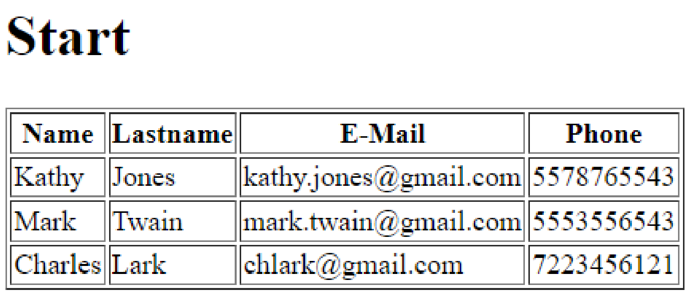
The reason why we inject an interface instead of a Class, is because an interface can be implemented by several classes, each of them will be tagged as a @Service and implement the same interface. This way, we can call upon any of those implementations via this interface and the interface will search for the right object type implementation amongst the tagged classes.

We can now use the interface methods from our controller.



```
//View redirect method
@GetMapping("/")
//Add Model as a parameter to send info to view
public String start(Model model){
    //Make use of Service interface methods
    var people = personService.listPeople();
    //Share with model
    model.addAttribute("people", people);
```

Our application is ready to run:



Start

Name	Lastname	E-Mail	Phone
Kathy	Jones	kathy.jones@gmail.com	5578765543
Mark	Twain	mark.twain@gmail.com	5553556543
Charles	Lark	chlark@gmail.com	7223456121

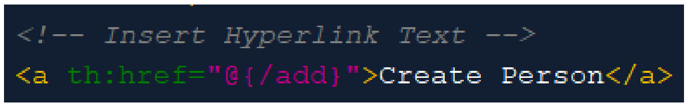
- **Spring Boot and MySQL – CRUD Creation:**

First of all, let's make a quick review of CRUD (Create, Read, Update, Delete) preferred HTTP request methods for a specific resource:

GET → Idempotent (identical request can only be made once) → Get resources from Server. Ideal to **Read** resources.
PUT →Idempotent (identical request can only be made once) → Make an update in Server. Ideal to make an **Update**.
DELETE →Idempotent (identical request can only be made once) → Deletes resources from Server. Ideal to **Delete** resources.
POST → Inserts into Server every time it's processed. Ideal to **Create** even if identical items are being created.

- **Create:**

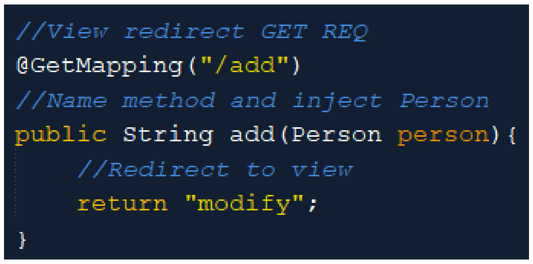
First of all, we will add a hyperlink to a new page in our view that will serve to create an insertion in our database.



```
<!-- Insert Hyperlink Text -->
<a th:href="@{/add}">Create Person</a>
```

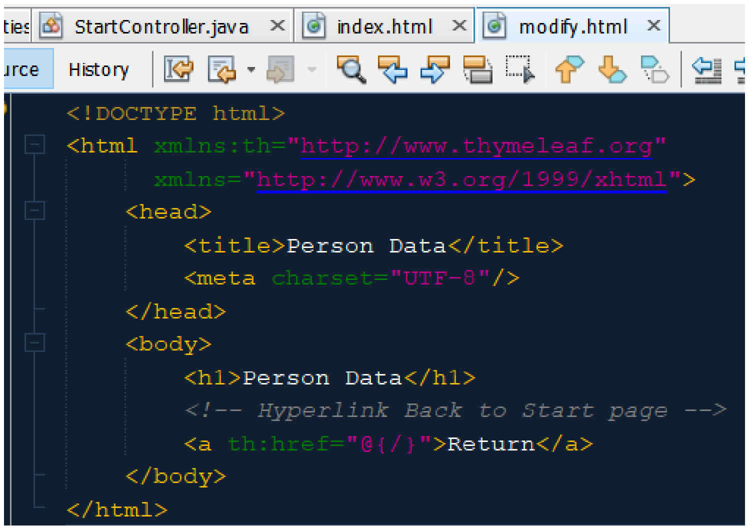
Use <a th:href="@{*path from mapping*}">Hyperlinked text</a> as the standard to insert links.

Now, at our controller, we need a new method for this new operation:



```
//View redirect GET REQ
@GetMapping("/add")
//Name method and inject Person
public String add(Person person){
    //Redirect to view
    return "modify";
}
```

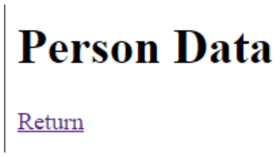
Use @GetMapping() annotation since it's going to use a GET type request. Inject a Person instance by providing it as an argument. Spring will look for a Person type object in the Spring Factory. If it doesn't exist, Spring will create it and inject it. This object will also be automatically available in the Request scope.



```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
      xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <title>Person Data</title>
        <meta charset="UTF-8"/>
    </head>
    <body>
        <h1>Person Data</h1>
        <!-- Hyperlink Back to Start page -->
        <a th:href="@{/}">Return</a>
    </body>
</html>
```

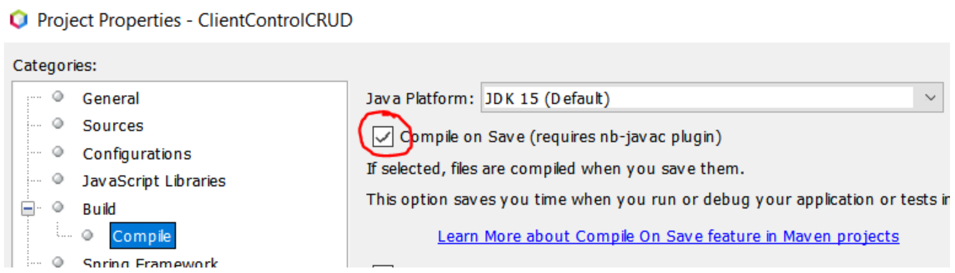
On the view side, we still need to create a new webpage called “modify”. Include the second namespace (w3) in order to remove DOCTYPE errors and apply strict html, meaning you need to close every single < with />.

First thing to do is to create another hyperlink back to the main Start page. Use <a th:href=”@{*mapper text*}”>*Text to be Displayed*</a> syntax to insert a hyperlink on the view side.


Person Data

Return

In order for Tomcat server to automatically reflect changes, and avoid restarting it every time, we need to select the following option:


Project Properties - ClientControlCRUD

Categories:
- General
- Sources
- Configurations
- JavaScript Libraries
- Build
  - Compile
- Spring Framework

Java Platform: JDK 15 (Default)

[✓] Compile on Save (requires nb-javac plugin)

If selected, files are compiled when you save them.

This option saves you time when you run or debug your application or tests i

Learn More about Compile On Save feature in Maven projects

Now we need a form so that the user can fill in the person information. To create a form, use the <form label. The action property, will specify the controller URI that will process the information provided via the form. The method property makes reference to the http method

used. Finally, the object property needs to specify the type of object that will be associated to the form and shared through the model.

```html
<form th:action="@{/save}" method="post" th:object="${person}">

    <label for="name">Name:</label>
    <input type="text" name="name" th:field="*{name}"></input>
    <br/>
    <label for="lastname">Lastname:</label>
    <input type="text" name="lastname" th:field="*{lastname}"></input>
    <br/>
    <label for="email">E-mail:</label>
    <input type="email" name="email" th:field="*{email}"></input>
    <br/>
    <label for="phone">Phone:</label>
    <input type="tel" name="phone" th:field="*{phone}"></input>
    <br/>
    <input type="submit" name="save" value="Add Person"/>

</form>
```

Notice when passing an object's attribute through the th:field property, we will use the *{*class attribute*} syntax to make reference to the particular defined class attribute.

Finally, we add the Submit button in order to go to the mapping provided in the action property and perform the action defined in the URI.

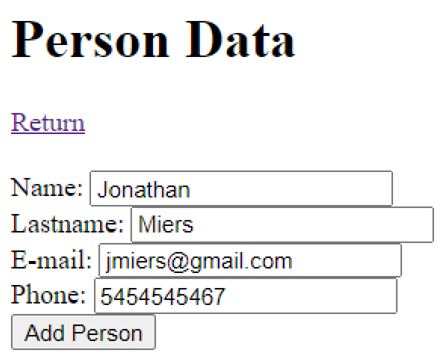
In that sense, we still need close the whole flow for adding a new Person to our DB. So we will provide our Controller with the mapping for this action.

```java
//React to Add Person Form Submit
@PostMapping("/save")
public String save(Person person) {
    //Insert info to DB
    personService.save(person);
    //Back to index page
    return "redirect:/";
}
```

Remember that each time an object is passed as an argument to a function, the Spring factory will look for it and create it if it doesn't exist and retrieve it if it finds that it already exists. In this case it will retrieve it along with the attribute values provided by the user into the form.

Using our personService we will make a straight input into our database.

Then, we want to return to our main page, so we use return "redirect:/".

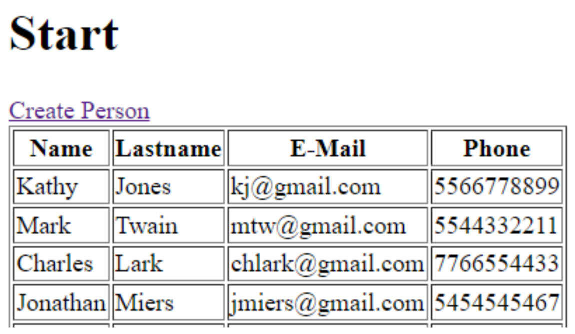


Person Data

Return

Name:	Jonathan
Lastname:	Miers
E-mail:	jmiers@gmail.com
Phone:	5454545467

Add Person



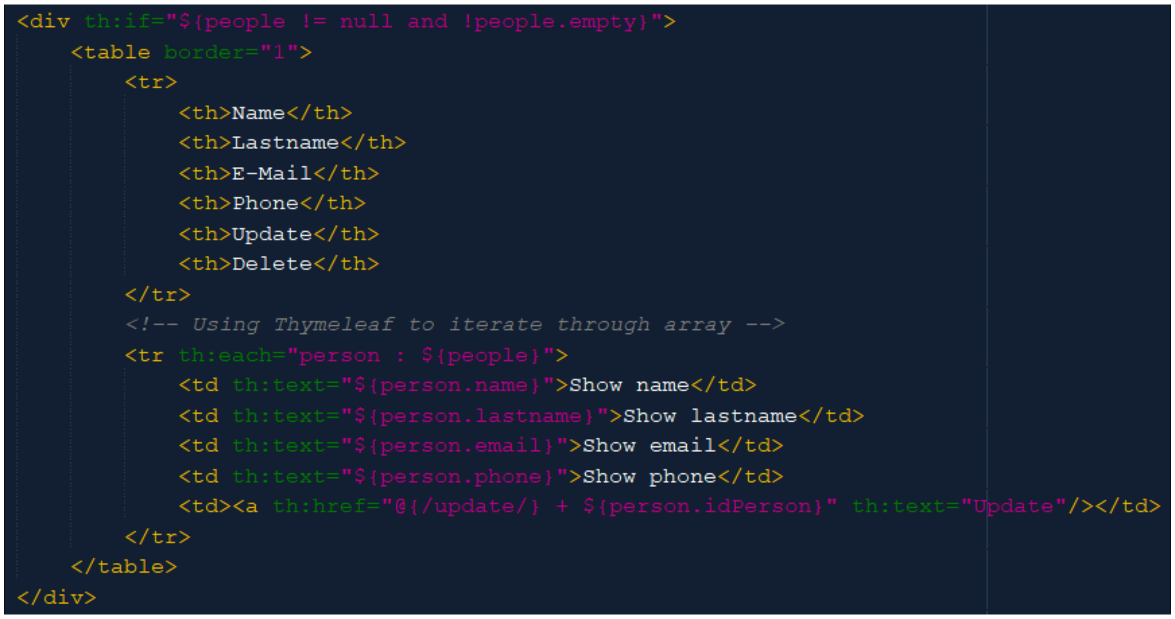
Start

Create Person

Name	Lastname	E-Mail	Phone
Kathy	Jones	kj@gmail.com	5566778899
Mark	Twain	mtw@gmail.com	5544332211
Charles	Lark	chlark@gmail.com	7766554433
Jonathan	Miers	jmiers@gmail.com	5454545467

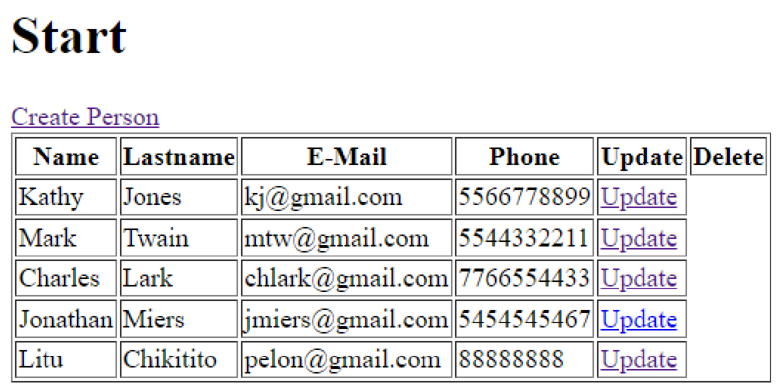
- **Update:**

In our main page, we have created a table. We are going to add the Update and Delete headers.



```html
<div th:if="${people != null and !people.empty}">
    <table border="1">
        <tr>
            <th>Name</th>
            <th>Lastname</th>
            <th>E-Mail</th>
            <th>Phone</th>
            <th>Update</th>
            <th>Delete</th>
        </tr>
        <!-- Using Thymeleaf to iterate through array -->
        <tr th:each="person : ${people}">
            <td th:text="${person.name}">Show name</td>
            <td th:text="${person.lastname}">Show lastname</td>
            <td th:text="${person.email}">Show email</td>
            <td th:text="${person.phone}">Show phone</td>
            <td><a th:href="@{/update/} + ${person.idPerson}" th:text="Update"/></td>
        </tr>
    </table>
</div>
```

As well as table content with a hyperlink formed by the controller mapping concatenated with the person ID. This ${*person.idPerson*} variable is known as a Path Variable.

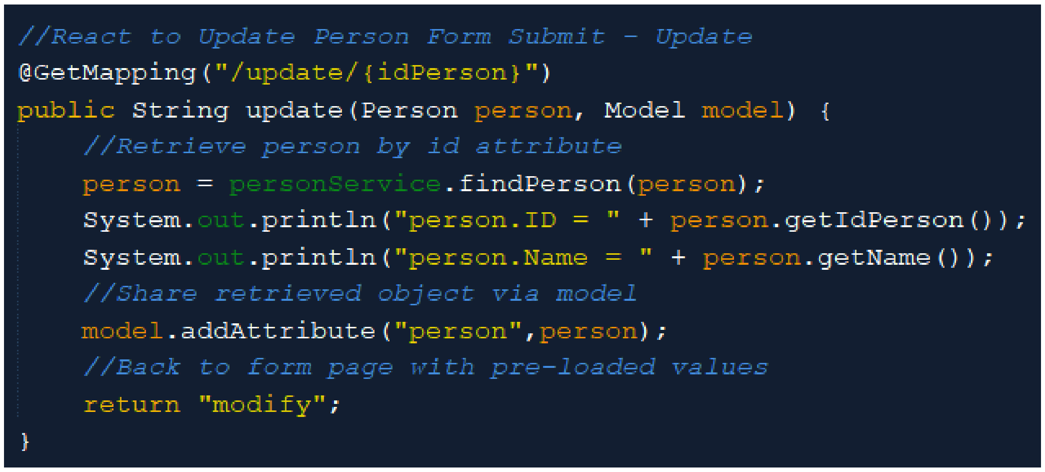


Start

Create Person

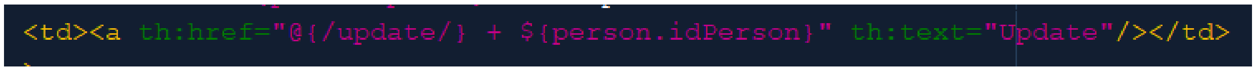
Name	Lastname	E-Mail	Phone	Update	Delete
Kathy	Jones	kj@gmail.com	5566778899	Update	
Mark	Twain	mtw@gmail.com	5544332211	Update	
Charles	Lark	chlark@gmail.com	7766554433	Update	
Jonathan	Miers	jmiers@gmail.com	5454545467	Update	
Litu	Chikitito	pelon@gmail.com	88888888	Update	

Now let's go map this for our controller:



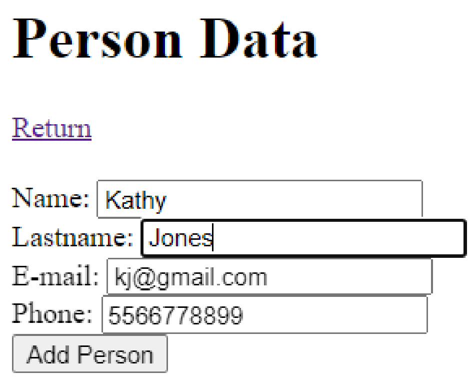
```
//React to Update Person Form Submit - Update
@GetMapping("/update/{idPerson}")
public String update(Person person, Model model) {
    //Retrieve person by id attribute
    person = personService.findPerson(person);
    System.out.println("person.ID = " + person.getIdPerson());
    System.out.println("person.Name = " + person.getName());
    //Share retrieved object via model
    model.addAttribute("person",person);
    //Back to form page with pre-loaded values
    return "modify";
}
```

Use get mapping since this is to retrieve an existing resource from the server. Map it to "/update/{idPerson}", this path variable needs to be called exactly what the attribute is called. The moment Spring factory looks for the injected Person object, it will retrieve the one with this ID, just as retrieved from the view:



```
<td><a th:href="@{/update/} + ${person.idPerson}" th:text="Update"/></td>
```

Use personService to go retrieve the rest of that person's data and assign it to the same person object. Don't forget to share it via model. We will use the same form view so that the user can modify whatever information needed.



Person Data

Return

Name:	Kathy
Lastname:	Jones
E-mail:	kj@gmail.com
Phone:	5566778899

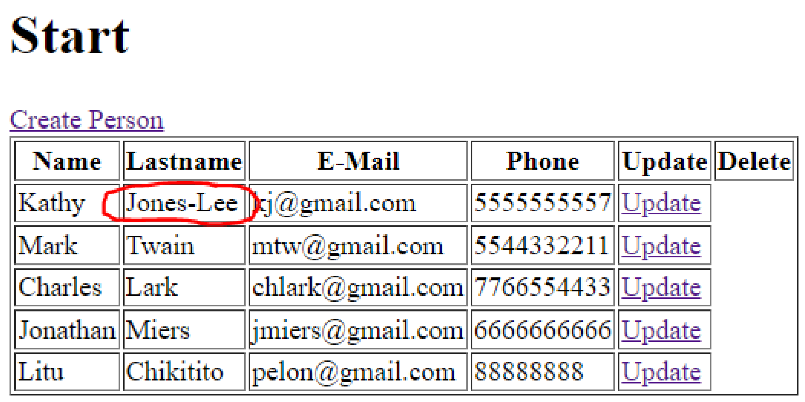
Information will be pre-loaded as the whole person object was shared in the Spring container and the view retrieves these fields by attribute association.

If we were to Submit this form, a new row would be created in our DB instead of modifying this one.

```html
<input type="hidden" name="idPerson" th:field="*{idPerson}">
```

The last thing to do in order to be able to save these changes, is to provide the Person ID on the view side. The best way to achieve this is by creating a hidden input value and associating it directly to the idPerson attribute.

Now we are able to update existing resources:

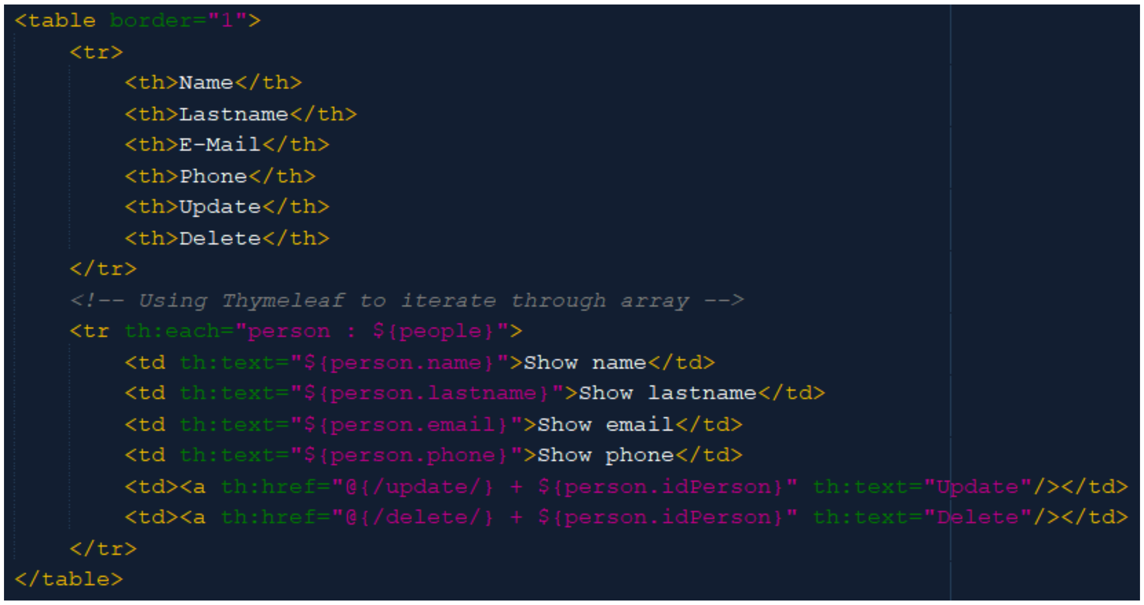


Start

Create Person

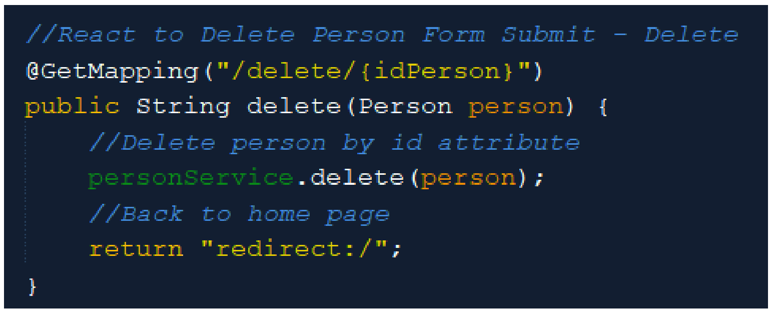
Name	Lastname	E-Mail	Phone	Update	Delete
Kathy	Jones-Lee	kj@gmail.com	5555555557	Update	
Mark	Twain	mtw@gmail.com	5544332211	Update	
Charles	Lark	chlark@gmail.com	7766554433	Update	
Jonathan	Miers	jmiers@gmail.com	6666666666	Update	
Litu	Chikitito	pelon@gmail.com	88888888	Update	

- **Delete:**



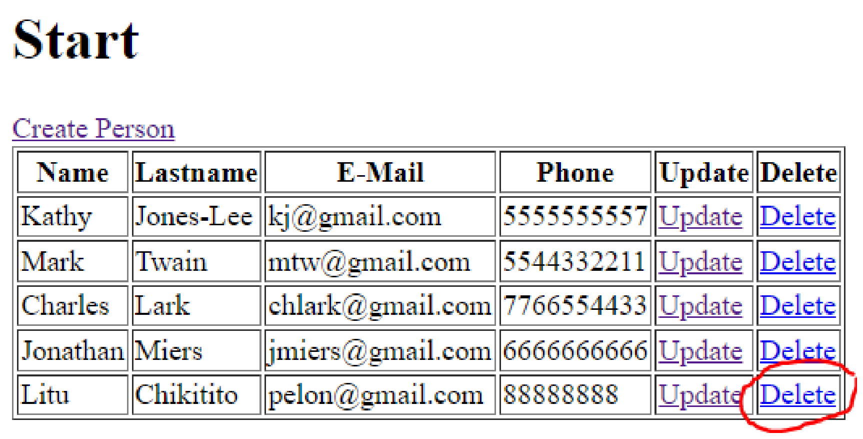
```html
<table border="1">
    <tr>
        <th>Name</th>
        <th>Lastname</th>
        <th>E-Mail</th>
        <th>Phone</th>
        <th>Update</th>
        <th>Delete</th>
    </tr>
    <!-- Using Thymeleaf to iterate through array -->
    <tr th:each="person : ${people}">
        <td th:text="${person.name}">Show name</td>
        <td th:text="${person.lastname}">Show lastname</td>
        <td th:text="${person.email}">Show email</td>
        <td th:text="${person.phone}">Show phone</td>
        <td><a th:href="@{/update/} + ${person.idPerson}" th:text="Update"/></td>
        <td><a th:href="@{/delete/} + ${person.idPerson}" th:text="Delete"/></td>
    </tr>
</table>
```

Introduce new table data with the new delete hyperlink and Delete text.



```java
//React to Delete Person Form Submit - Delete
@GetMapping("/delete/{idPerson}")
public String delete(Person person) {
    //Delete person by id attribute
    personService.delete(person);
    //Back to home page
    return "redirect:/";
}
```

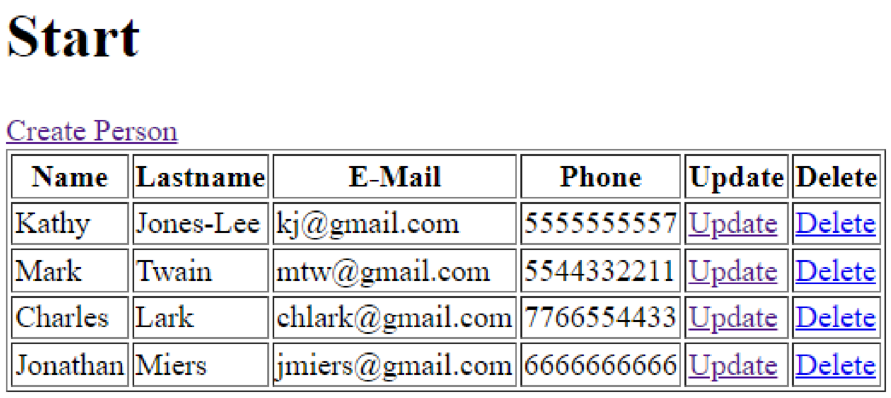
Map and define method for this action.



Start

Create Person

Name	Lastname	E-Mail	Phone	Update	Delete
Kathy	Jones-Lee	kj@gmail.com	5555555557	Update	Delete
Mark	Twain	mtw@gmail.com	5544332211	Update	Delete
Charles	Lark	chlark@gmail.com	7766554433	Update	Delete
Jonathan	Miers	jmiers@gmail.com	6666666666	Update	Delete
Litu	Chikitito	pelon@gmail.com	88888888	Update	Delete



Start

Create Person

Name	Lastname	E-Mail	Phone	Update	Delete
Kathy	Jones-Lee	kj@gmail.com	5555555557	Update	Delete
Mark	Twain	mtw@gmail.com	5544332211	Update	Delete
Charles	Lark	chlark@gmail.com	7766554433	Update	Delete
Jonathan	Miers	jmiers@gmail.com	6666666666	Update	Delete

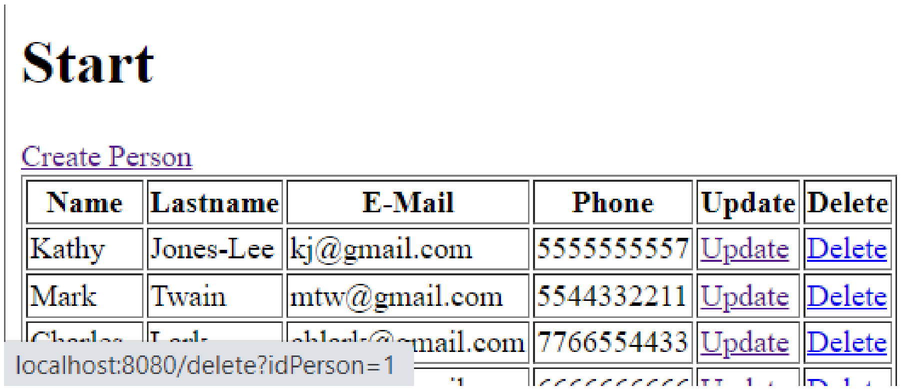
We will see the actual row disappear and since we are redirected to the same page, no other change is appreciated.

There's yet another way to accomplish this. Instead of building an URI with a path variable, we could use a query parameter:

```
<td><a th:href="@{/delete(idPerson=${person.idPerson})}" th:text="Delete"/></td>
```

Now the URI will be builded as such:

localhost:8080/delete?idPerson=1



Start

Create Person

Name	Lastname	E-Mail	Phone	Update	Delete
Kathy	Jones-Lee	kj@gmail.com	5555555557	Update	Delete
Mark	Twain	mtw@gmail.com	5544332211	Update	Delete
[illegible]	[illegible]	mail.com	7766554433	Update	Delete

localhost:8080/delete?idPerson=1

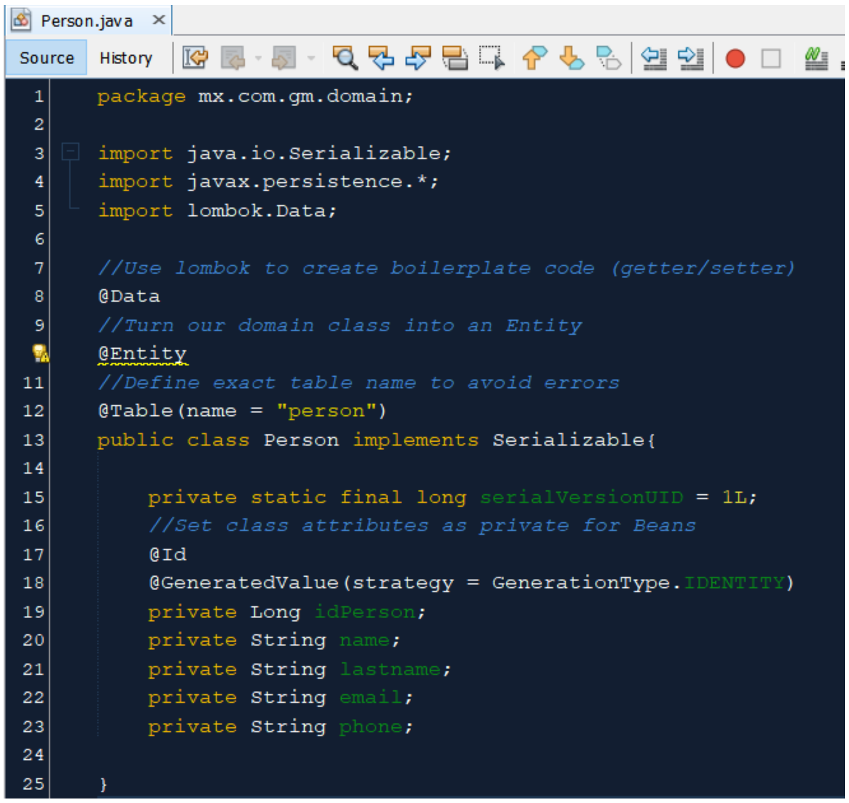
Last change will be on the controller side:

```java
//React to Delete Person Form Submit - Delete
@GetMapping("/delete")
public String delete(Person person) {
    //Delete person by id attribute
    personService.delete(person);
    //Back to home page
    return "redirect:/";
}
```

Since we are no longer using a path variable, remove it from the Mapping. We will receive the Person ID through the view, by defining the query parameter.

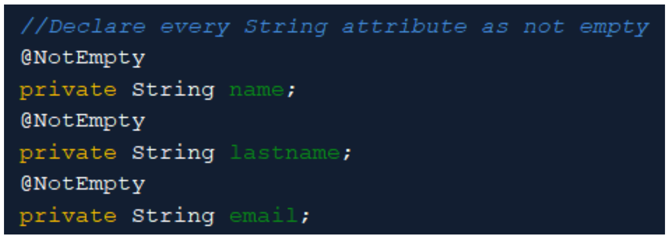
- **Spring - Validations:**

When working with forms, it is necessary to validate the input fields, meaning restrictions have to be made so that we receive only the kind of input allowed.



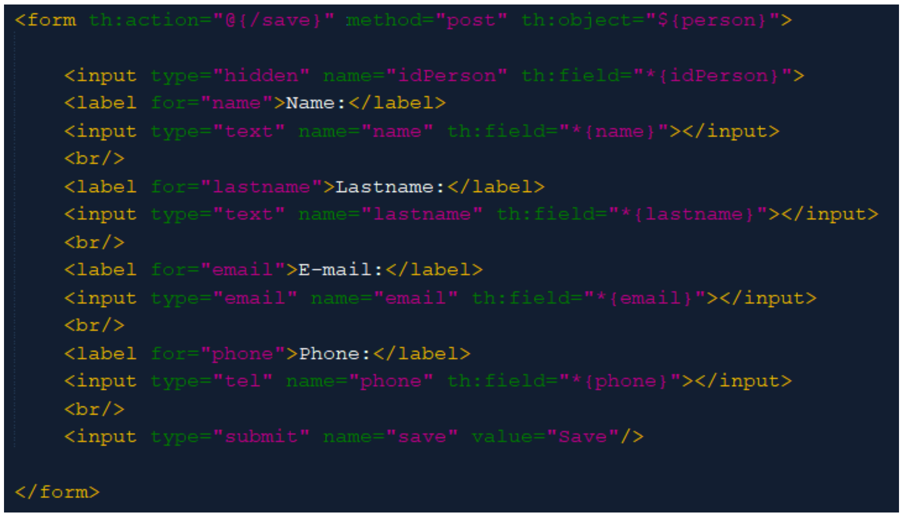
```java
package mx.com.gm.domain;

import java.io.Serializable;
import javax.persistence.*;
import lombok.Data;

//Use lombok to create boilerplate code (getter/setter)
@Data
//Turn our domain class into an Entity
@Entity
//Define exact table name to avoid errors
@Table(name = "person")
public class Person implements Serializable{

    private static final long serialVersionUID = 1L;
    //Set class attributes as private for Beans
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long idPerson;
    private String name;
    private String lastname;
    private String email;
    private String phone;

}
```

First off, we need to make some changes in our domain class in order to define business layer restrictions.



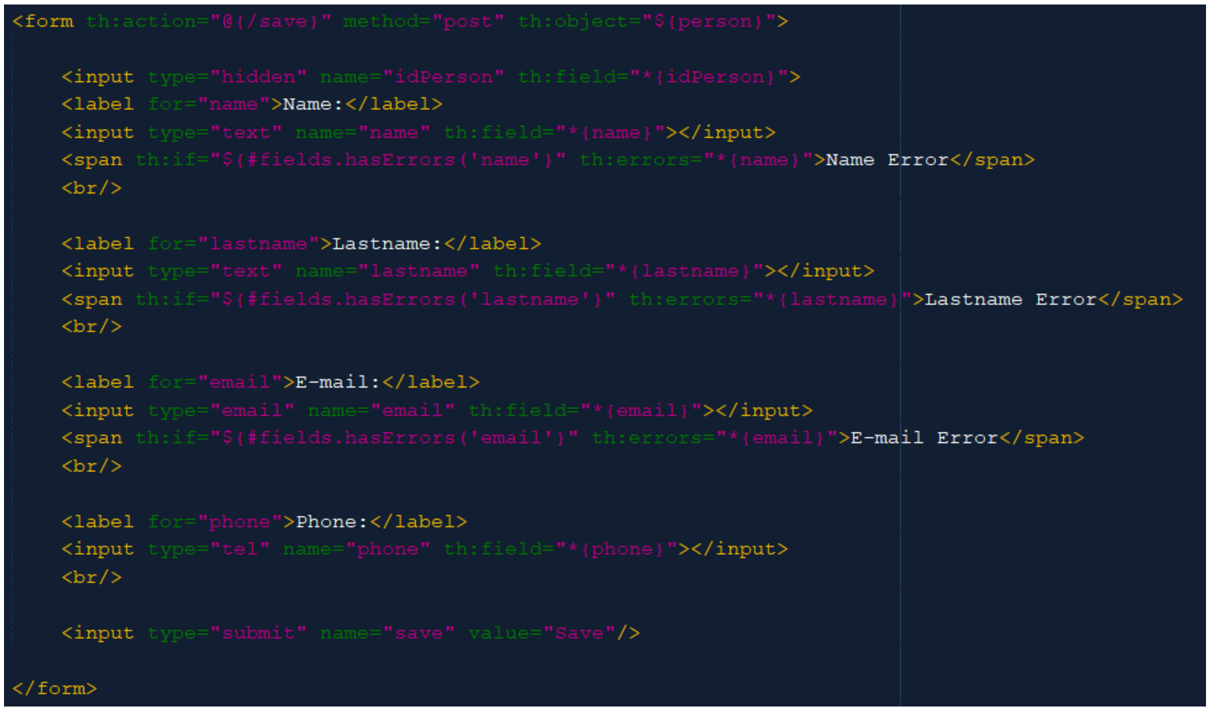
```java
//Declare every String attribute as not empty
@NotEmpty
private String name;
@NotEmpty
private String lastname;
@NotEmpty
private String email;
```

Every String type attribute can be annotated with the org.hibernate.validator.constraints.NotEmpty; @NotEmpty tag, this is to avoid user from submitting the form without filling one of these fields, checks that it’s not null nor empty.



```
<form th:action="@{/save}" method="post" th:object="${person}">

    <input type="hidden" name="idPerson" th:field="*{idPerson}">
    <label for="name">Name:</label>
    <input type="text" name="name" th:field="*{name}"></input>
    <br/>
    <label for="lastname">Lastname:</label>
    <input type="text" name="lastname" th:field="*{lastname}"></input>
    <br/>
    <label for="email">E-mail:</label>
    <input type="email" name="email" th:field="*{email}"></input>
    <br/>
    <label for="phone">Phone:</label>
    <input type="tel" name="phone" th:field="*{phone}"></input>
    <br/>
    <input type="submit" name="save" value="Save"/>

</form>
```

Now, our view side is not adapted to display the user with error messages. We need to tell the user what he’s missing in a clear way:



```
<form th:action="@{/save}" method="post" th:object="${person}">

    <input type="hidden" name="idPerson" th:field="*{idPerson}">
    <label for="name">Name:</label>
    <input type="text" name="name" th:field="*{name}"></input>
    <span th:if="${#fields.hasErrors('name')}" th:errors="*{name}">Name Error</span>
    <br/>

    <label for="lastname">Lastname:</label>
    <input type="text" name="lastname" th:field="*{lastname}"></input>
    <span th:if="${#fields.hasErrors('lastname')}" th:errors="*{lastname}">Lastname Error</span>
    <br/>

    <label for="email">E-mail:</label>
    <input type="email" name="email" th:field="*{email}"></input>
    <span th:if="${#fields.hasErrors('email')}" th:errors="*{email}">E-mail Error</span>
    <br/>

    <label for="phone">Phone:</label>
    <input type="tel" name="phone" th:field="*{phone}"></input>
    <br/>

    <input type="submit" name="save" value="Save"/>

</form>
```

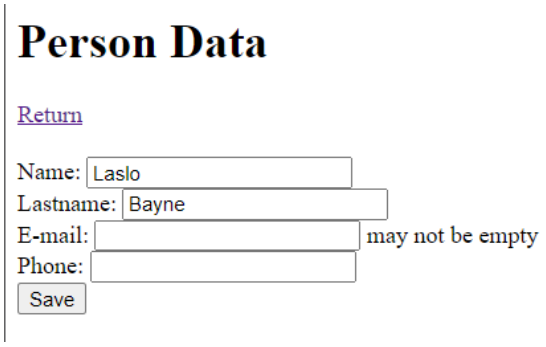
The <span> tag allows us to show text in case an if validation turns out true. In this case, we are using the #fields joker to call a function depending on the field input. Validate if it has errors and react with th:errors label by showing the errors in the given attribute.

Last but not least, comes the task of recovering said errors. From our Controller class, we are going to retrieve them. The action that will detonate such errors will be the pressing of the Save button, so the relevant mapped function will be the one that "saves".

```
//React to Save Person Form Submit - Create
@PostMapping("/save")
public String save(@Valid Person person, Errors errors) {
    //Validate for errors
    if(errors.hasErrors()) {
        return "modify";
    }
    //Insert info to DB
    personService.save(person);
    //Back to index page
    return "redirect:/";
}
```

The first thing to do is to annotate Person with the @Valid annotation from javax.validation.Valid library. The second thing to do, is to inject an Errors class type object from the org.springframework.validation.Errors library so that we can use its methods. It's important that these are the first two arguments received in the function, after them, we can have any other argument.

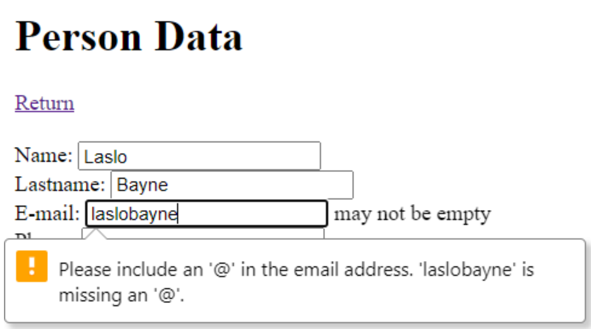
Next, we will validate for errors and if so, we'll be returned to the same view, never allowing for a redirect to happen. The errors deployed in the view, will be defined in modify.html view.


Person Data

Return

Name: Laslo
Lastname: Bayne
E-mail: may not be empty
Phone:

A different format from the type expected will not be allowed either:


Person Data

Return

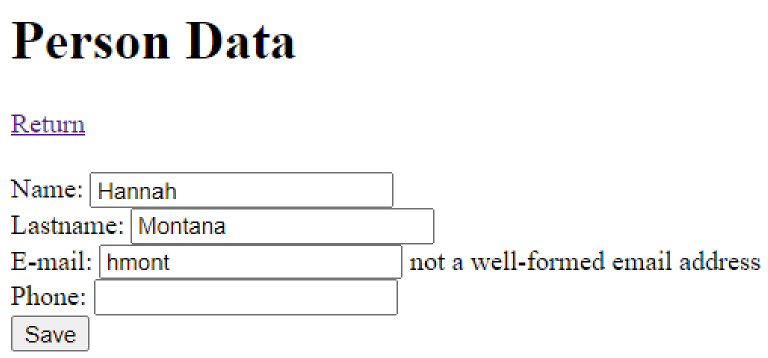
Name: Laslo
Lastname: Bayne
E-mail: laslobayne may not be empty

Please include an '@' in the email address. 'laslobayne' is missing an '@'.

There is though, a standard e-mail validation in the javax library, so we are going to declare it as standard text in the front end (no HTML validation) and validate it instead, from the backend.

```
@NotEmpty
@Email
private String email;
```

The E-Mail attribute should be not empty nor a simple text. Import the @Email annotation from org.hibernate.validator.constraints.Email;.



Person Data

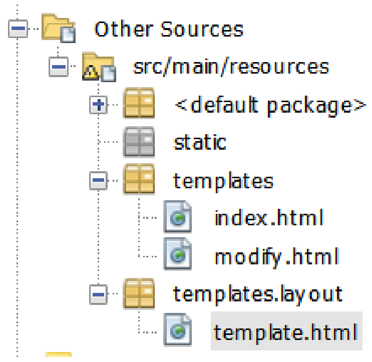
Return

Name: Hannah
Lastname: Montana
E-mail: hmont not a well-formed email address
Phone:

Now until a well formed e-mail address is inserted, the user will not be able to submit.

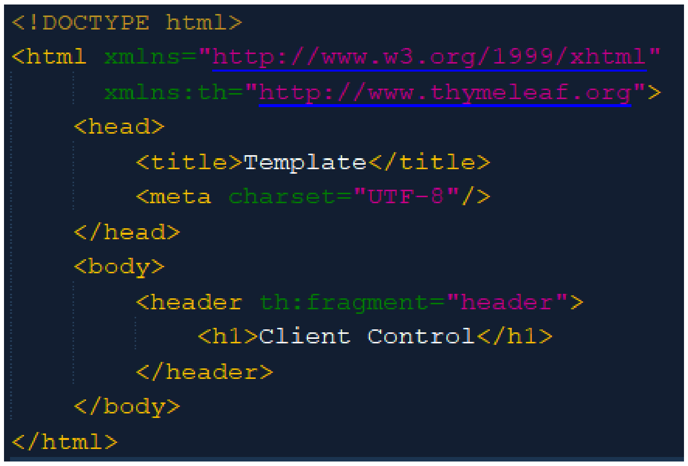
- **Spring – Thymeleaf Templates:**

We will create a new folder for our thymeleaf templates:



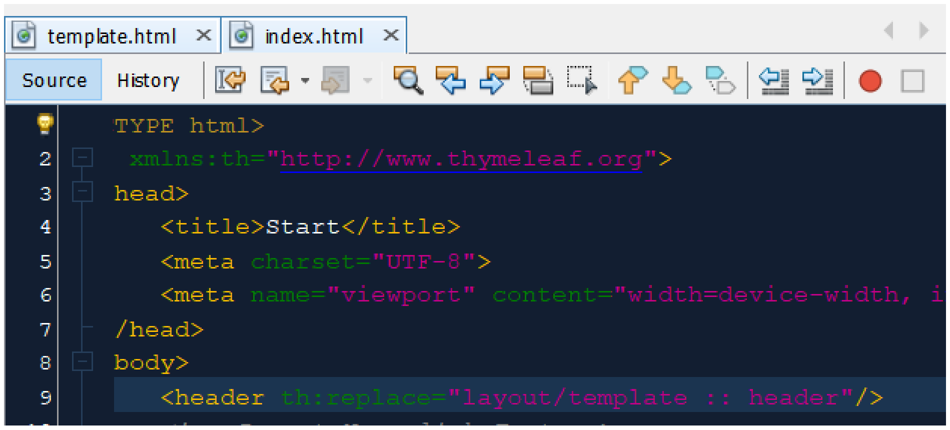
```
Other Sources
  src/main/resources
    <default package>
    static
    templates
      index.html
      modify.html
    templates.layout
      template.html
```

Our new template will have the same name spaces we've been working with so far. Add a simple title and header. This header will be a declared as a fragment so that we can reuse it for other pages. All of the code inside the header tags will be recyclable:

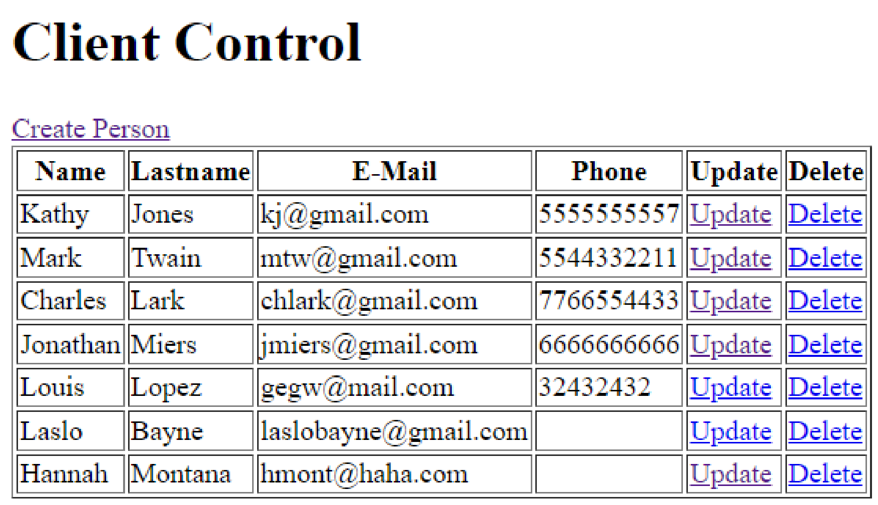


```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
    <head>
        <title>Template</title>
        <meta charset="UTF-8"/>
    </head>
    <body>
        <header th:fragment="header">
            <h1>Client Control</h1>
        </header>
    </body>
</html>
```

Now in our index.html view, we will call this fragment to display it:



```
TYPE html>
 xmlns:th="http://www.thymeleaf.org">
head>
    <title>Start</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, i
/head>
body>
    <header th:replace="layout/template :: header"/>
```



Client Control

Create Person

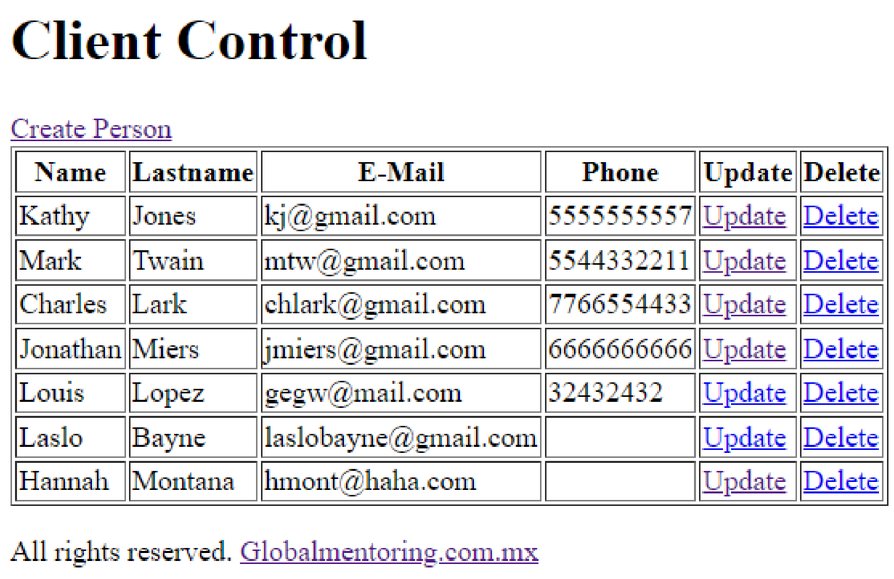
Name	Lastname	E-Mail	Phone	Update	Delete
Kathy	Jones	kj@gmail.com	5555555557	Update	Delete
Mark	Twain	mtw@gmail.com	5544332211	Update	Delete
Charles	Lark	chlark@gmail.com	7766554433	Update	Delete
Jonathan	Miers	jmiers@gmail.com	6666666666	Update	Delete
Louis	Lopez	gegw@mail.com	32432432	Update	Delete
Laslo	Bayne	laslobayne@gmail.com		Update	Delete
Hannah	Montana	hmont@haha.com		Update	Delete

Client Control has now substituted the previous Start header.

Same goes for a footer fragment:

```
<footer th:fragment="footer">
    <p>All rights reserved. <a href="http://www.globalmentoring.com.mx" target="_blank">Globalmentoring.com.mx</a></p>
</footer>
```

```
<footer th:replace="layout/template :: footer"></footer>
```



Client Control

Create Person

Name	Lastname	E-Mail	Phone	Update	Delete
Kathy	Jones	kj@gmail.com	5555555557	Update	Delete
Mark	Twain	mtw@gmail.com	5544332211	Update	Delete
Charles	Lark	chlark@gmail.com	7766554433	Update	Delete
Jonathan	Miers	jmiers@gmail.com	6666666666	Update	Delete
Louis	Lopez	gegw@mail.com	32432432	Update	Delete
Laslo	Bayne	laslobayne@gmail.com		Update	Delete
Hannah	Montana	hmont@haha.com		Update	Delete

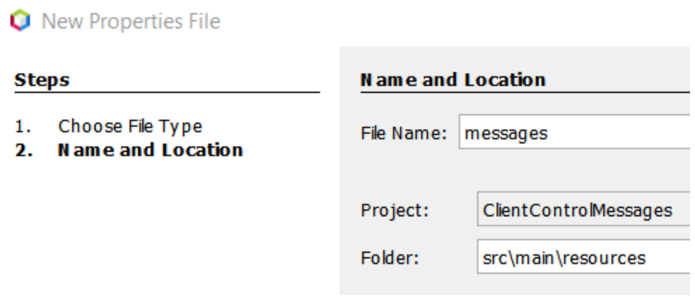
All rights reserved. Globalmentoring.com.mx

By using fragments, many other elements may be shared through view pages, such as a dashboard, menus, side bars, etc.

- **Spring – Messages:**

Instead of mapping html messages inside the view page, now we are going to retrieve them from a properties file. By doing this, we will be able to handle several languages, one properties file per language.

Create a new properties file:



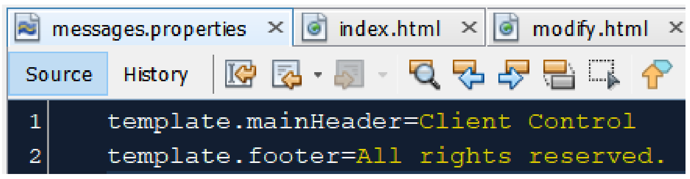
File Name: messages

Project: ClientControlMessages

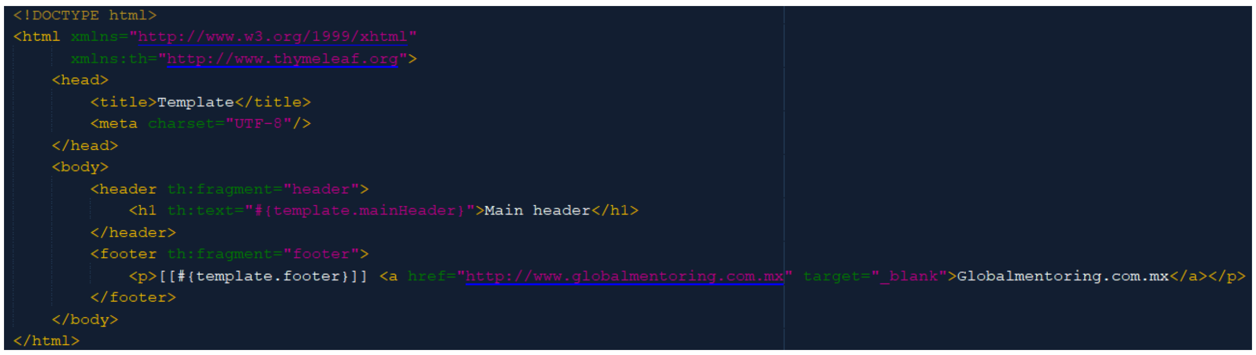
Folder: src\main\resources

This file must be named messages in order to follow the default configuration mode, and be inside of the resources folder.

The primary goal of this file, is to store all of the plain text that we display in our html pages and assign it to a labels. For instance:

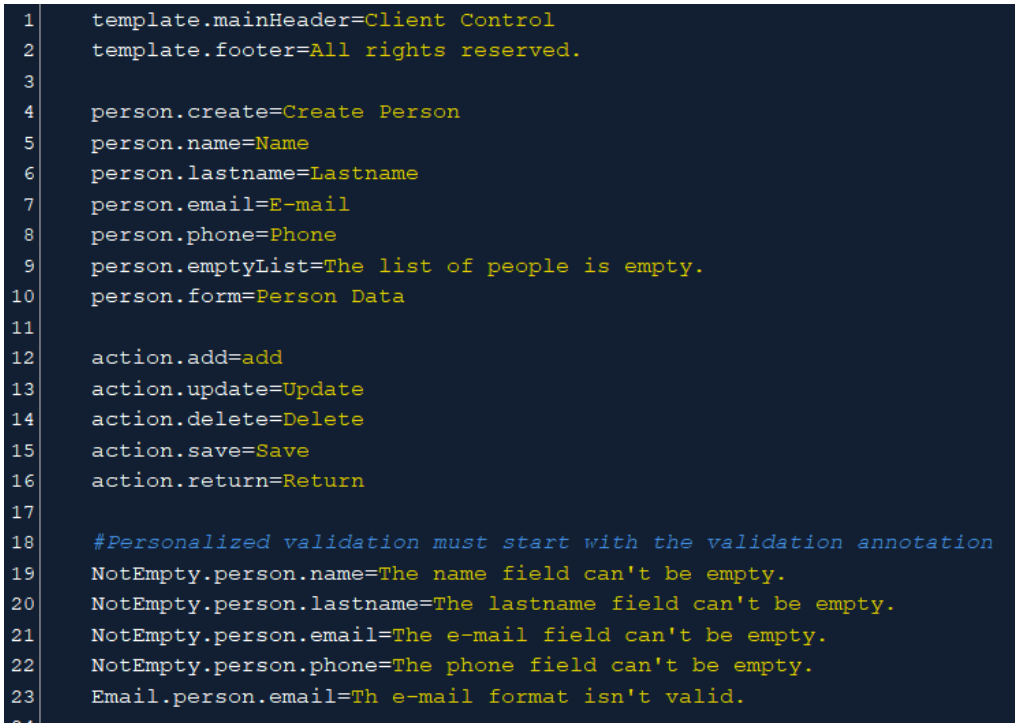


```
template.mainHeader=Client Control
template.footer=All rights reserved.
```



```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
    <head>
        <title>Template</title>
        <meta charset="UTF-8"/>
    </head>
    <body>
        <header th:fragment="header">
            <h1 th:text="#{template.mainHeader}">Main header</h1>
        </header>
        <footer th:fragment="footer">
            <p>[[#{template.footer}]] <a href="http://www.globalmentoring.com.mx" target="_blank">Globalmentoring.com.mx</a></p>
        </footer>
    </body>
</html>
```

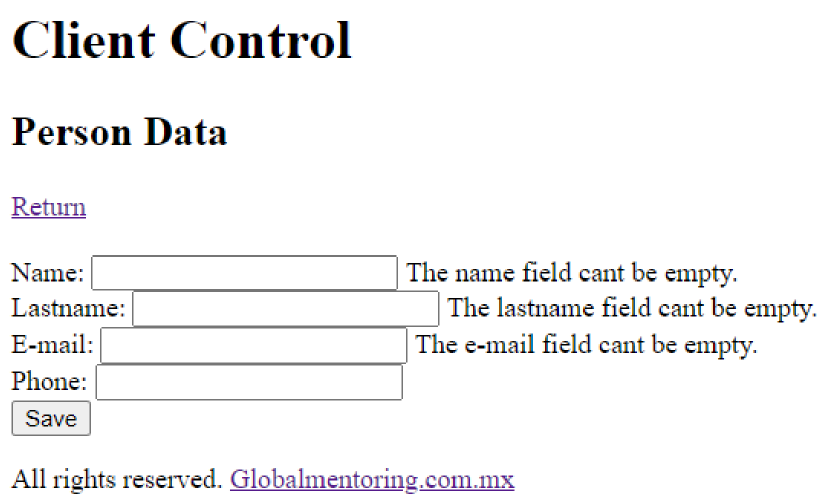
Use the th:text="#{*properties.label*}" syntax to invoke the properties label from inside thymeleaf text property. If you wanted to call them from a plain text field, use the [[#{*properties.label*}]] syntax, in this case, done not to lose the hyperlink.



```
template.mainHeader=Client Control
template.footer=All rights reserved.

person.create=Create Person
person.name=Name
person.lastname=Lastname
person.email=E-mail
person.phone=Phone
person.emptyList=The list of people is empty.
person.form=Person Data

action.add=add
action.update=Update
action.delete=Delete
action.save=Save
action.return=Return

#Personalized validation must start with the validation annotation
NotEmpty.person.name=The name field can't be empty.
NotEmpty.person.lastname=The lastname field can't be empty.
NotEmpty.person.email=The e-mail field can't be empty.
NotEmpty.person.phone=The phone field can't be empty.
Email.person.email=Th e-mail format isn't valid.
```

We can personalize every single text used in our view pages, plain text, object attributes, actions, error messages, etc.

For the field validation case, variable names must start with the annotation name. While using this syntax, the message we have defined for every specific error will automatically be retrieved from this file and deployed in the front view.



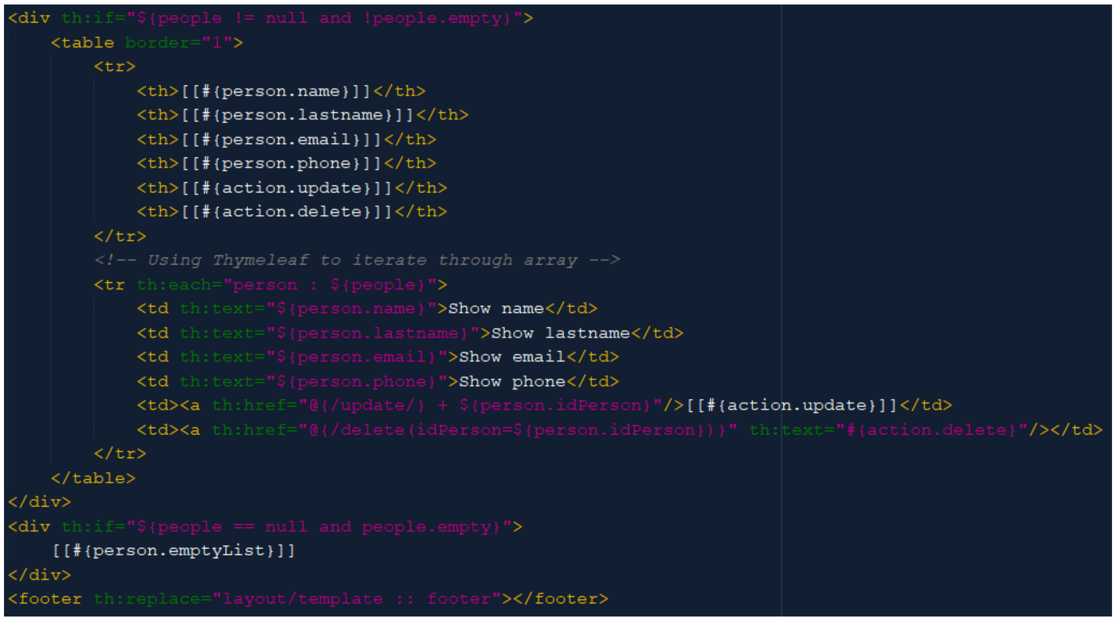
Client Control

Person Data

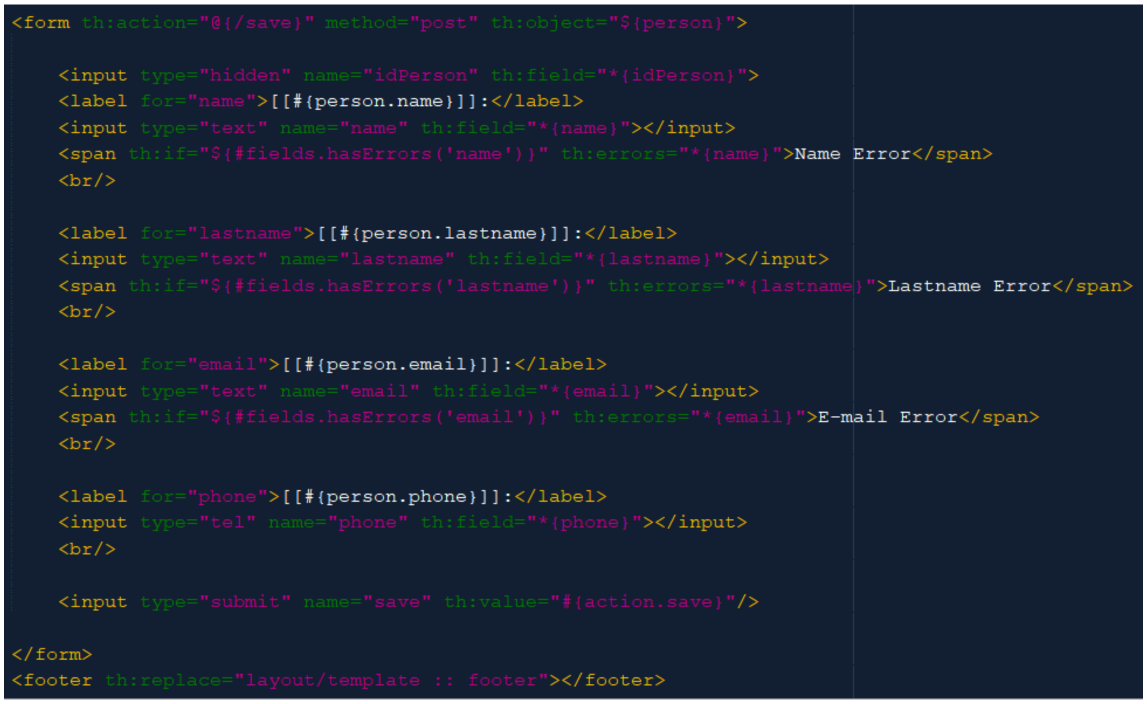
Return

Name: The name field cant be empty.
Lastname: The lastname field cant be empty.
E-mail: The e-mail field cant be empty.
Phone:
Save

All rights reserved. Globalmentoring.com.mx



```
<div th:if="${people != null and !people.empty}">
    <table border="1">
        <tr>
            <th>[[#{person.name}]]</th>
            <th>[[#{person.lastname}]]</th>
            <th>[[#{person.email}]]</th>
            <th>[[#{person.phone}]]</th>
            <th>[[#{action.update}]]</th>
            <th>[[#{action.delete}]]</th>
        </tr>
        <!-- Using Thymeleaf to iterate through array -->
        <tr th:each="person : ${people}">
            <td th:text="${person.name}">Show name</td>
            <td th:text="${person.lastname}">Show lastname</td>
            <td th:text="${person.email}">Show email</td>
            <td th:text="${person.phone}">Show phone</td>
            <td><a th:href="@{/update/} + ${person.idPerson}"/>[[#{action.update}]]</td>
            <td><a th:href="@{/delete(idPerson=${person.idPerson})}" th:text="#{action.delete}"/></td>
        </tr>
    </table>
</div>
<div th:if="${people == null and people.empty}">
    [[#{person.emptyList}]]
</div>
<footer th:replace="layout/template :: footer"></footer>
```



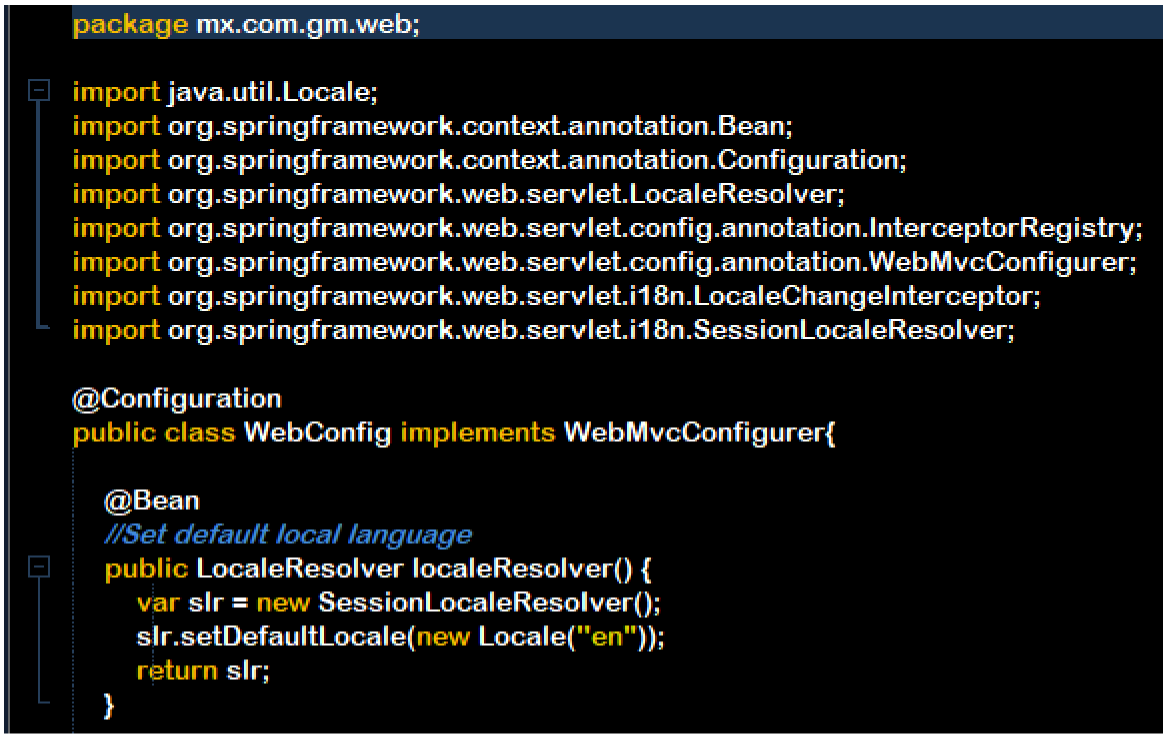
```
<form th:action="@{/save}" method="post" th:object="${person}">

    <input type="hidden" name="idPerson" th:field="*{idPerson}">
    <label for="name">[[#{person.name}]]:</label>
    <input type="text" name="name" th:field="*{name}"></input>
    <span th:if="${#fields.hasErrors('name')}" th:errors="*{name}">Name Error</span>
    <br/>

    <label for="lastname">[[#{person.lastname}]]:</label>
    <input type="text" name="lastname" th:field="*{lastname}"></input>
    <span th:if="${#fields.hasErrors('lastname')}" th:errors="*{lastname}">Lastname Error</span>
    <br/>

    <label for="email">[[#{person.email}]]:</label>
    <input type="text" name="email" th:field="*{email}"></input>
    <span th:if="${#fields.hasErrors('email')}" th:errors="*{email}">E-mail Error</span>
    <br/>

    <label for="phone">[[#{person.phone}]]:</label>
    <input type="tel" name="phone" th:field="*{phone}"></input>
    <br/>

    <input type="submit" name="save" th:value="#{action.save}"/>

</form>
<footer th:replace="layout/template :: footer"></footer>
```

This was useful to replace all of the plain text in our Thymeleaf pages. Now, this is especially resourceful when making websites international.

- **Spring – Internationalization (i18n):**

First of all, we will create a new class → WebConfig, which will be inside the web package. This class is part of the MVC pattern and will implement a corresponding interface. This class needs to be annotation with the @Configuration annotation from Spring Framework Context.

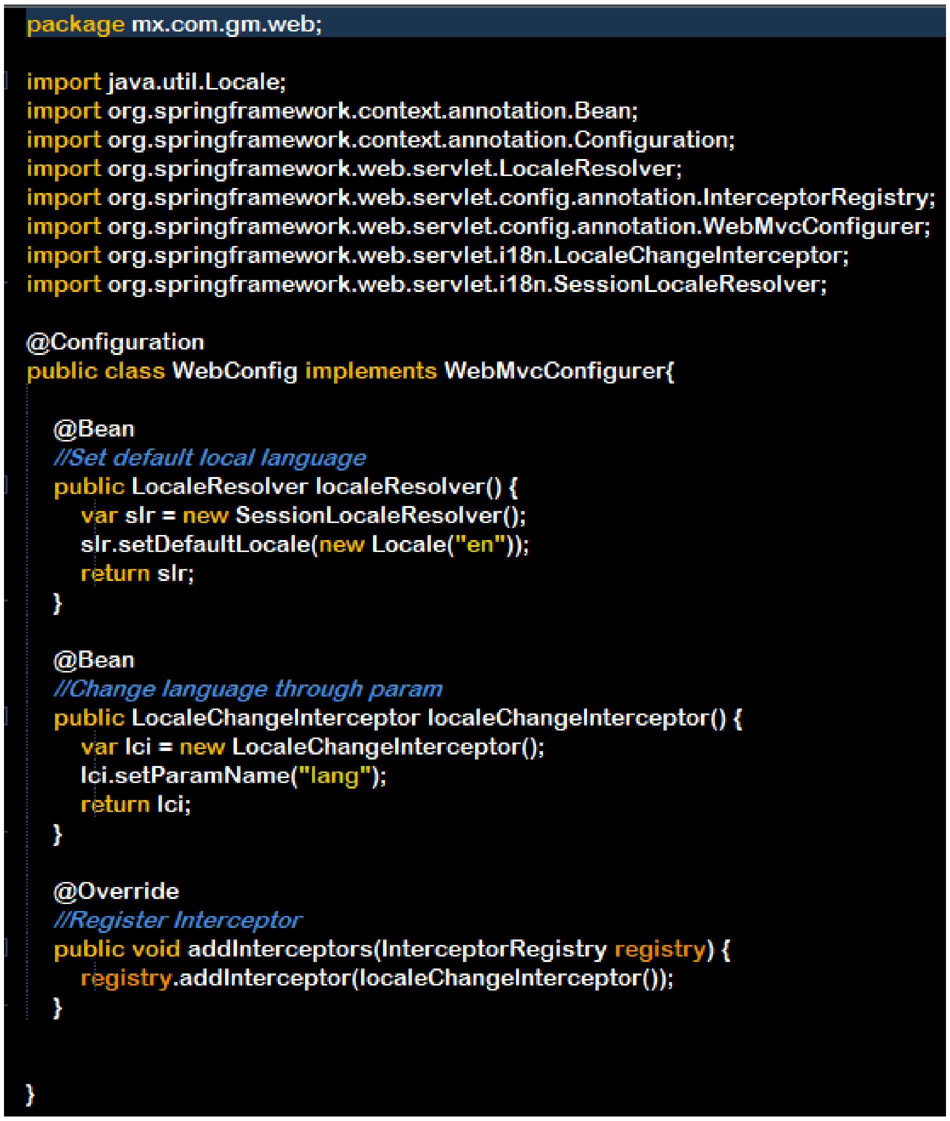
This inferface has several methods that we will implement, such as addInterceptors, useful for Internationalization. Use the @Bean annotation to create a bean class instance that will be automatically added to the Spring container context.



```java
package mx.com.gm.web;

import java.util.Locale;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.LocaleResolver;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
import org.springframework.web.servlet.i18n.LocaleChangeInterceptor;
import org.springframework.web.servlet.i18n.SessionLocaleResolver;

@Configuration
public class WebConfig implements WebMvcConfigurer{

    @Bean
    //Set default local language
    public LocaleResolver localeResolver() {
        var slr = new SessionLocaleResolver();
        slr.setDefaultLocale(new Locale("en"));
        return slr;
    }
```

This class will have several imports related to the servlet behavior.

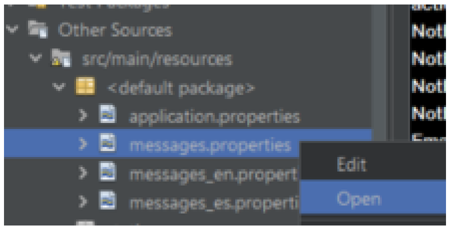
First thing to do is to create an automatic instance of a SessionLocaleResolver and add it to the Spring context. This will give us a default language to work with.



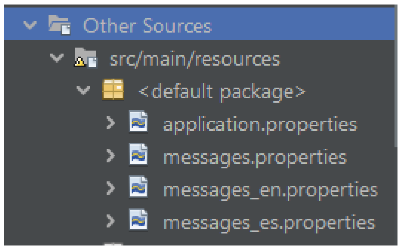
```java
package mx.com.gm.web;

import java.util.Locale;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.LocaleResolver;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
import org.springframework.web.servlet.i18n.LocaleChangeInterceptor;
import org.springframework.web.servlet.i18n.SessionLocaleResolver;

@Configuration
public class WebConfig implements WebMvcConfigurer{

    @Bean
    //Set default local language
    public LocaleResolver localeResolver() {
        var slr = new SessionLocaleResolver();
        slr.setDefaultLocale(new Locale("en"));
        return slr;
    }

    @Bean
    //Change language through param
    public LocaleChangeInterceptor localeChangeInterceptor() {
        var lci = new LocaleChangeInterceptor();
        lci.setParamName("lang");
        return lci;
    }

    @Override
    //Register Interceptor
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(localeChangeInterceptor());
    }


}
```

Add an addInterceptors method implementation so that lifecycle interceptors for controller invocations and handling before and after processing are managed. Our class is now ready to dynamically change languages. Language management will be controlled through a properties file.

Right click in the messages.properties file and select Open to visualize the file in a Key → Default language pair mode.

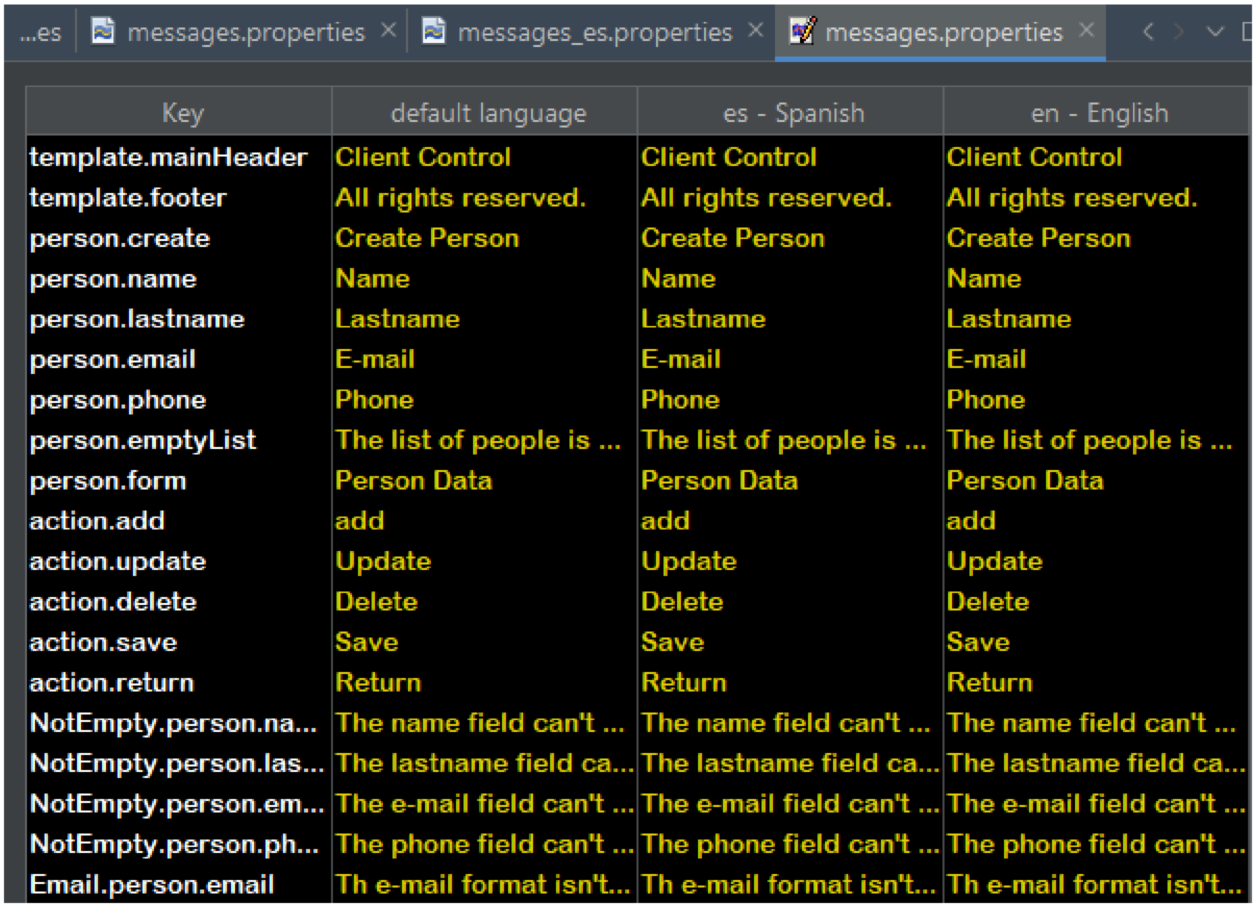


Now copy this file multiple times, one for every language you want to manage.



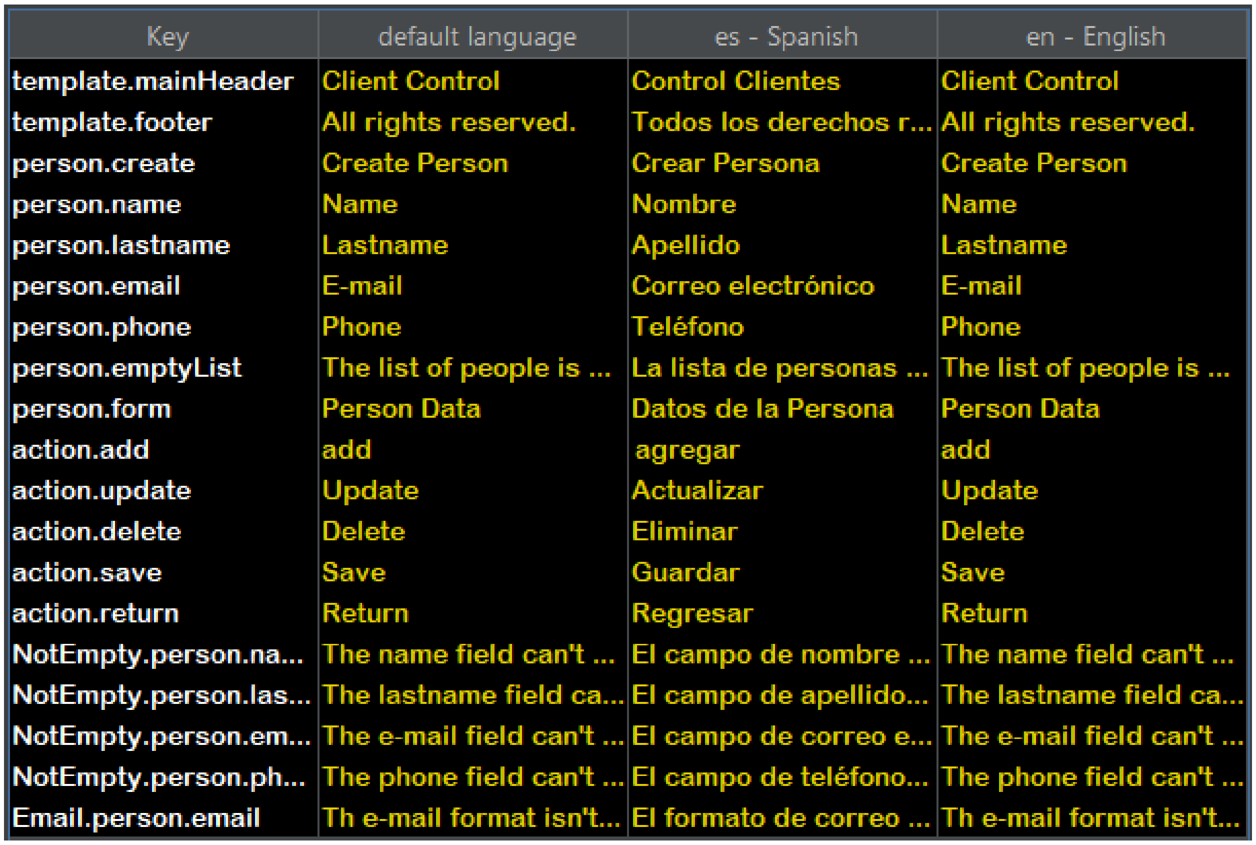
```
Other Sources
  src/main/resources
    <default package>
      application.properties
      messages.properties
      messages_en.properties
      messages_es.properties
```

Rename them using the corresponding language suffix. Now messages.properties will contain all of the Key – message equivalencies for every language case.



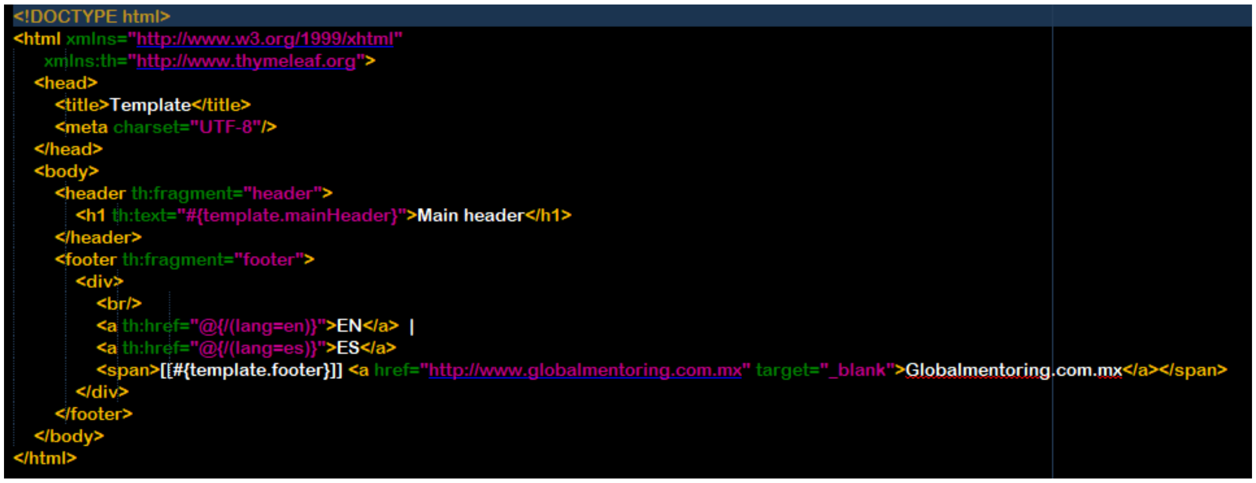
Key	default language	es - Spanish	en - English
template.mainHeader	Client Control	Client Control	Client Control
template.footer	All rights reserved.	All rights reserved.	All rights reserved.
person.create	Create Person	Create Person	Create Person
person.name	Name	Name	Name
person.lastname	Lastname	Lastname	Lastname
person.email	E-mail	E-mail	E-mail
person.phone	Phone	Phone	Phone
person.emptyList	The list of people is ...	The list of people is ...	The list of people is ...
person.form	Person Data	Person Data	Person Data
action.add	add	add	add
action.update	Update	Update	Update
action.delete	Delete	Delete	Delete
action.save	Save	Save	Save
action.return	Return	Return	Return
NotEmpty.person.na...	The name field can't ...	The name field can't ...	The name field can't ...
NotEmpty.person.las...	The lastname field ca...	The lastname field ca...	The lastname field ca...
NotEmpty.person.em...	The e-mail field can't ...	The e-mail field can't ...	The e-mail field can't ...
NotEmpty.person.ph...	The phone field can't ...	The phone field can't ...	The phone field can't ...
Email.person.email	Th e-mail format isn't...	Th e-mail format isn't...	Th e-mail format isn't...

Without modifying the Key column, we will translate the es – Spanish column to the desired equivalent value:



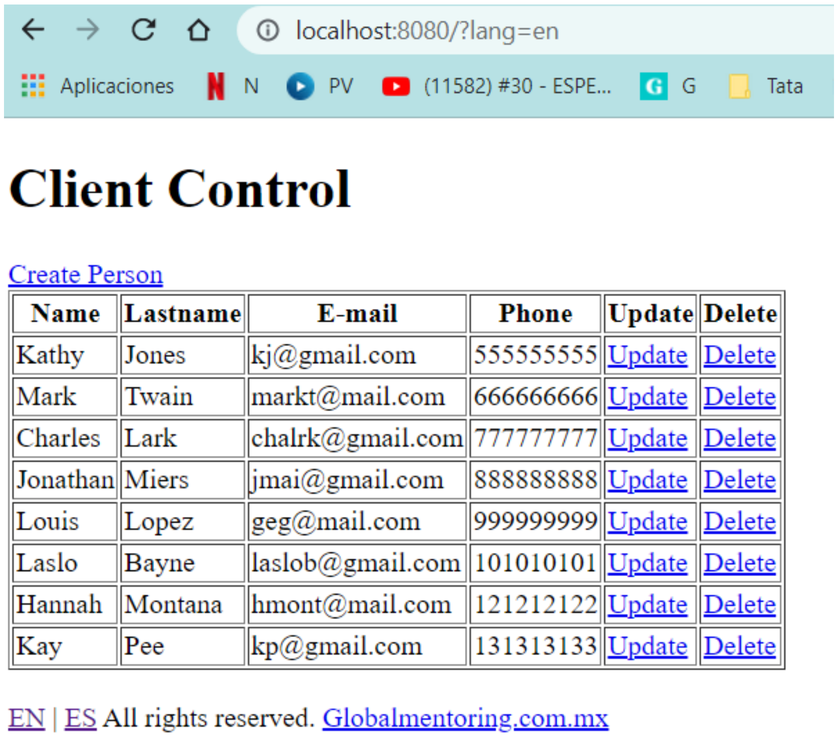
Key	default language	es - Spanish	en - English
template.mainHeader	Client Control	Control Clientes	Client Control
template.footer	All rights reserved.	Todos los derechos r...	All rights reserved.
person.create	Create Person	Crear Persona	Create Person
person.name	Name	Nombre	Name
person.lastname	Lastname	Apellido	Lastname
person.email	E-mail	Correo electrónico	E-mail
person.phone	Phone	Teléfono	Phone
person.emptyList	The list of people is ...	La lista de personas ...	The list of people is ...
person.form	Person Data	Datos de la Persona	Person Data
action.add	add	agregar	add
action.update	Update	Actualizar	Update
action.delete	Delete	Eliminar	Delete
action.save	Save	Guardar	Save
action.return	Return	Regresar	Return
NotEmpty.person.na...	The name field can't ...	El campo de nombre ...	The name field can't ...
NotEmpty.person.las...	The lastname field ca...	El campo de apellido...	The lastname field ca...
NotEmpty.person.em...	The e-mail field can't ...	El campo de correo e...	The e-mail field can't ...
NotEmpty.person.ph...	The phone field can't ...	El campo de teléfono...	The phone field can't ...
Email.person.email	Th e-mail format isn't...	El formato de correo ...	Th e-mail format isn't...

Last, we will implement this somehow from the view so that the user can select another language:



```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
     xmlns:th="http://www.thymeleaf.org">
   <head>
      <title>Template</title>
      <meta charset="UTF-8"/>
   </head>
   <body>
      <header th:fragment="header">
         <h1 th:text="#{template.mainHeader}">Main header</h1>
      </header>
      <footer th:fragment="footer">
         <div>
            <br/>
            <a th:href="@{/(lang=en)}">EN</a>  |
            <a th:href="@{/(lang=es)}">ES</a>
            <span>[[#{template.footer}]] <a href="http://www.globalmentoring.com.mx" target="_blank">Globalmentoring.com.mx</a></span>
         </div>
      </footer>
   </body>
</html>
```

This is our index page now:


localhost:8080/?lang=en

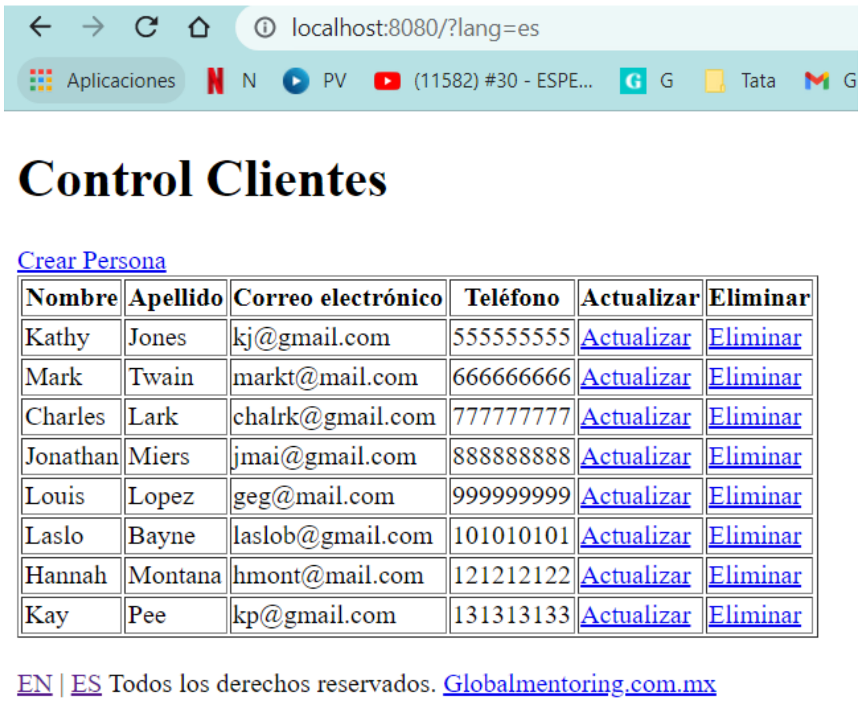
Client Control

Create Person

Name	Lastname	E-mail	Phone	Update	Delete
Kathy	Jones	kj@gmail.com	555555555	Update	Delete
Mark	Twain	markt@mail.com	666666666	Update	Delete
Charles	Lark	chalrk@gmail.com	777777777	Update	Delete
Jonathan	Miers	jmai@gmail.com	888888888	Update	Delete
Louis	Lopez	geg@mail.com	999999999	Update	Delete
Laslo	Bayne	laslob@gmail.com	101010101	Update	Delete
Hannah	Montana	hmont@mail.com	121212122	Update	Delete
Kay	Pee	kp@gmail.com	131313133	Update	Delete

EN | ES All rights reserved. Globalmentoring.com.mx

Spanish:


localhost:8080/?lang=es

Control Clientes

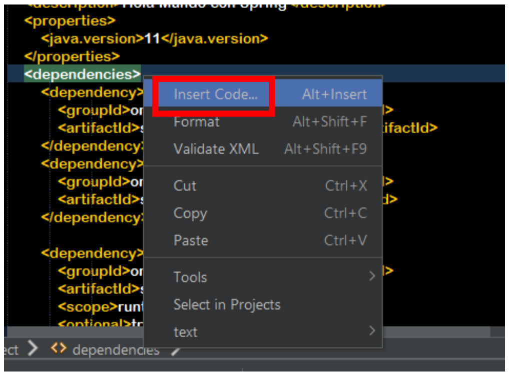
Crear Persona

Nombre	Apellido	Correo electrónico	Teléfono	Actualizar	Eliminar
Kathy	Jones	kj@gmail.com	555555555	Actualizar	Eliminar
Mark	Twain	markt@mail.com	666666666	Actualizar	Eliminar
Charles	Lark	chalrk@gmail.com	777777777	Actualizar	Eliminar
Jonathan	Miers	jmai@gmail.com	888888888	Actualizar	Eliminar
Louis	Lopez	geg@mail.com	999999999	Actualizar	Eliminar
Laslo	Bayne	laslob@gmail.com	101010101	Actualizar	Eliminar
Hannah	Montana	hmont@mail.com	121212122	Actualizar	Eliminar
Kay	Pee	kp@gmail.com	131313133	Actualizar	Eliminar

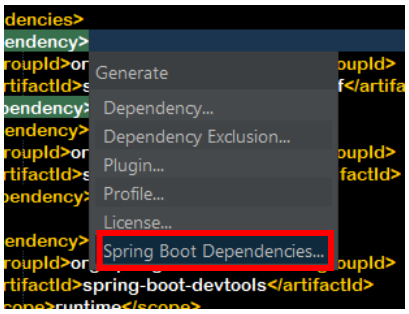
EN | ES Todos los derechos reservados. Globalmentoring.com.mx

- **Spring – Security:**

Before doing anything else, we need to add the Spring Security Dependency to our project:

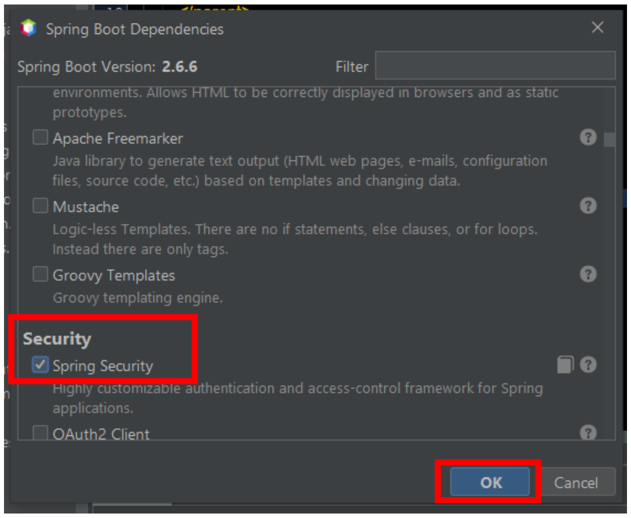


```
<properties>
    <java.version>11</java.version>
</properties>
<dependencies>
    <dependency>
        <groupId>or
        <artifactId>s
    </dependency>
    <dependency>
        <groupId>or
        <artifactId>s
    </dependency>

    <dependency>
        <groupId>or
        <artifactId>s
        <scope>runt
```



```
tifactId>spring-boot-devtools</artifactId>
```

Right click in the pom.xml file and select Insert Code → Spring Boot Dependencies…



Spring Boot Version: **2.6.6**

environments. Allows HTML to be correctly displayed in browsers and as static prototypes.

☐ Apache Freemarker
Java library to generate text output (HTML web pages, e-mails, configuration files, source code, etc.) based on templates and changing data.

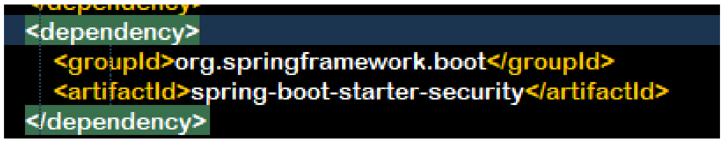
☐ Mustache
Logic-less Templates. There are no if statements, else clauses, or for loops. Instead there are only tags.

☐ Groovy Templates
Groovy templating engine.

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

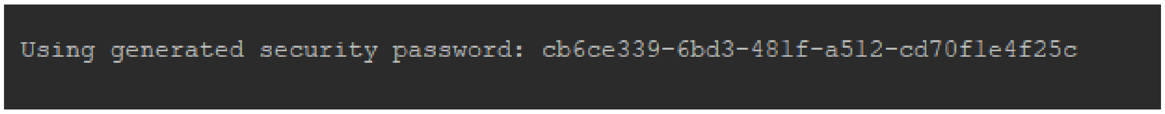
☐ OAuth2 Client

Tick Spring Security and then OK.



```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

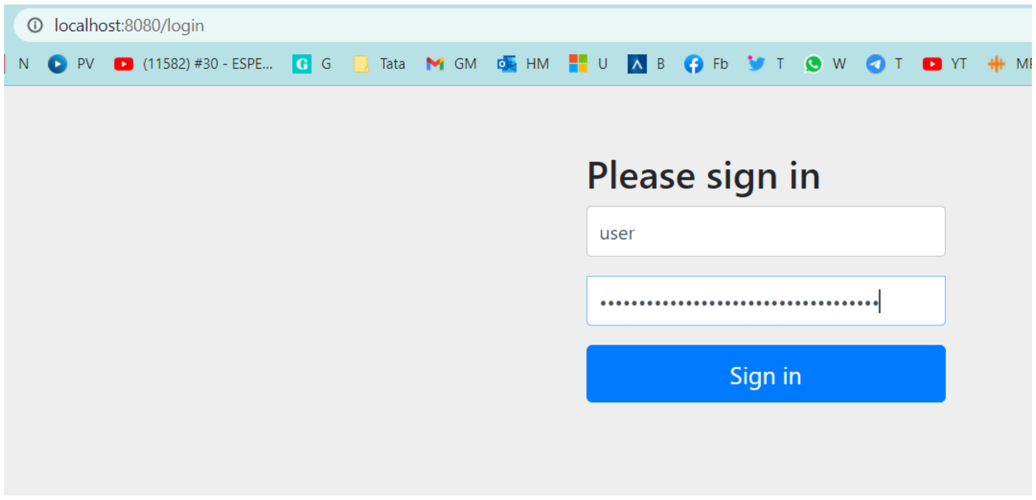
Just by adding this dependency, a password was automatically generated when the project ran:



```
Using generated security password: cb6ce339-6bd3-481f-a512-cd70f1e4f25c
```

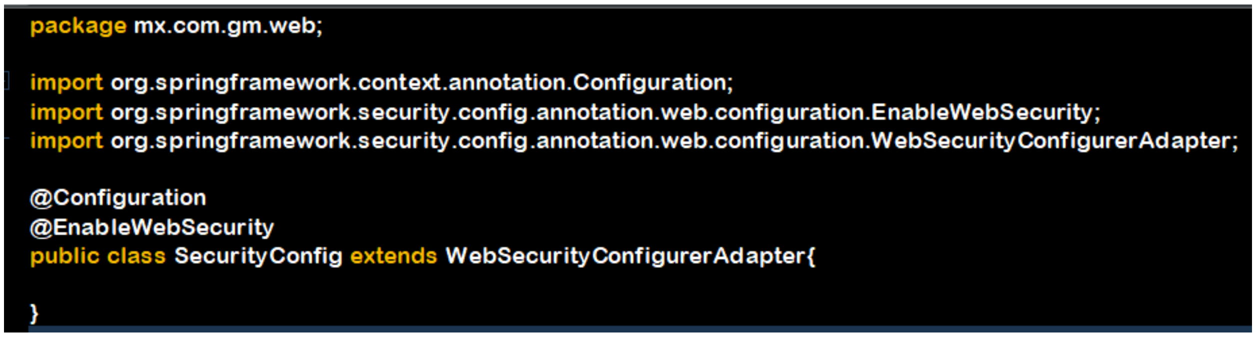
Using generated security password: 5aec68d0-e8c6-4cb4-842a-fee2cea2e952.

Also by default, Spring Security created a generic login page and is asking us for credentials. The default username is “user” and the password will be visible in the Application deployment log.



Though we have a first barrier, we still need to configure other aspects of Spring Security.

Create a new class in the web package:

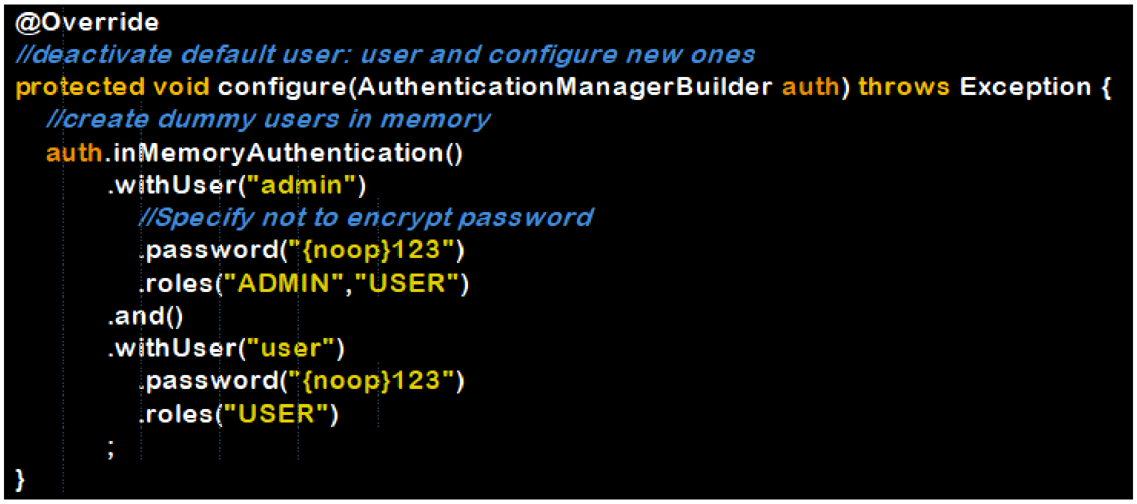


```
package mx.com.gm.web;

import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter{

}
```

This class will manage the aspects of Spring Security and must have a related name. Annotate if with @Configuration so that configurations herein defined are included in Spring’s Application Context. Also use @EnableWebSecurity as a marker so that Spring can add this class to the framework context. Last, extend from WebSecurityConfigurerAdapter. These 3 steps will form up the applicationContext.xml file indirectly.

The first method, known as *AUTHENTICATION*, will be used to restrict users allowed into our App. User must present its credentials in this step.

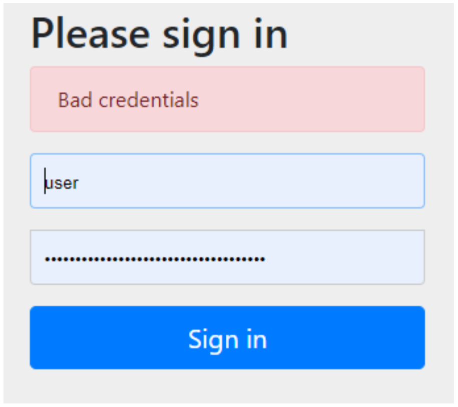
We will create two dummy users authorized to access our application and give an example of the syntax:



```
@Override
//deactivate default user: user and configure new ones
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
   //create dummy users in memory
   auth.inMemoryAuthentication()
         .withUser("admin")
            //Specify not to encrypt password
            .password("{noop}123")
            .roles("ADMIN","USER")
         .and()
         .withUser("user")
            .password("{noop}123")
            .roles("USER")
         ;
}
```

By using the Authentication Manager Builder instance, we are able to call the methods in it. Use withUser() method to set a username, call the rest of the methods concatenated to define the user's password and roles. Behind the scenes, Spring will automatically change the role label syntax, for instance: role_ADMIN.

Spring is now expecting one of the recently defined credentials to allow login. Any other combination will be denied access:



Please sign in

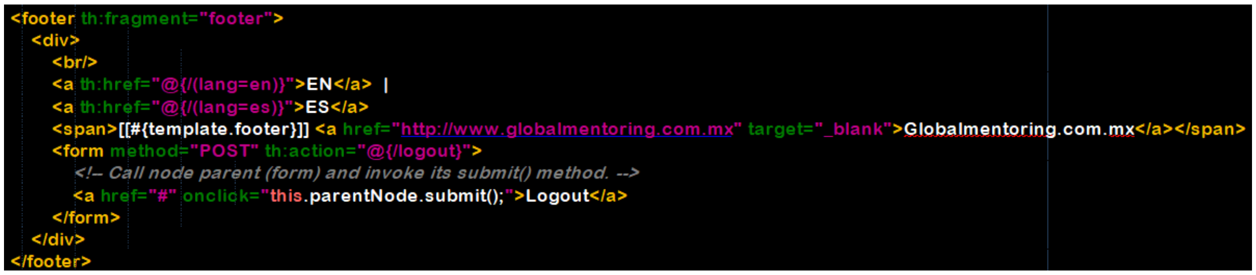
Bad credentials

user

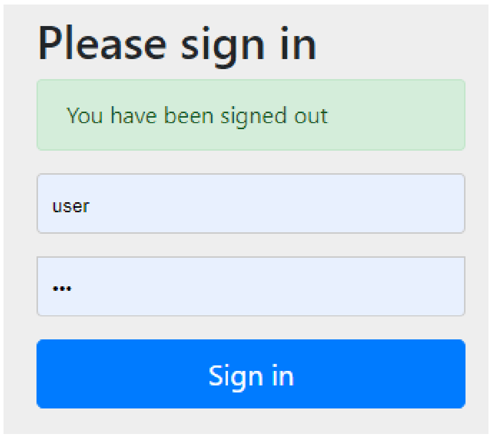
••••••••••••••••••••••••••••••••••••••

Let’s add a Logout function in our web page so that we can try out both users. Remember logout’s http method must be of type POST due to the kind of submit (link in a form) we are going to send.

To tell Spring Security that we want to logout, refer the action to @{/logout} and direct the link’s href to “#”. This doesn’t need to be specified in any Controller.



```
<footer th:fragment="footer">
  <div>
    <br/>
    <a th:href="@{/(lang=en)}">EN</a>  |
    <a th:href="@{/(lang=es)}">ES</a>
    <span>[[#{template.footer}]] <a href="http://www.globalmentoring.com.mx" target="_blank">Globalmentoring.com.mx</a></span>
    <form method="POST" th:action="@{/logout}">
      <!-- Call node parent (form) and invoke its submit() method. -->
      <a href="#" onclick="this.parentNode.submit();">Logout</a>
    </form>
  </div>
</footer>
```



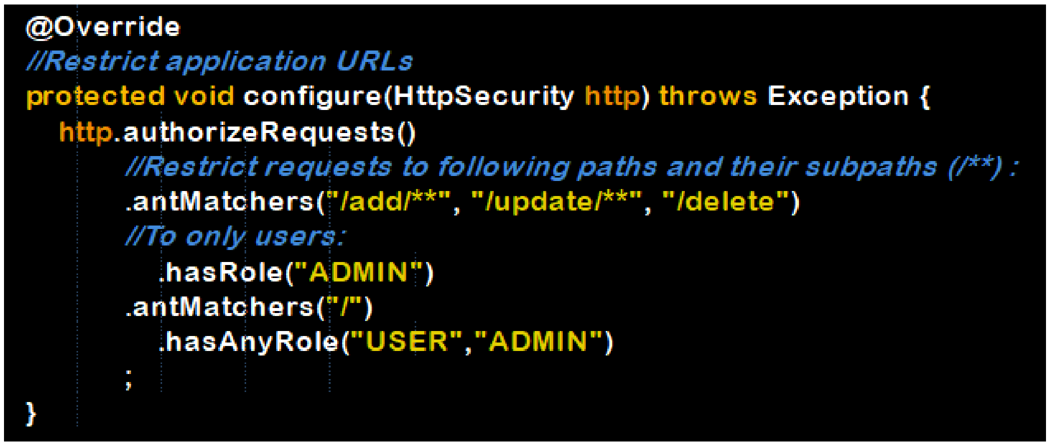
Please sign in

You have been signed out

user

•••

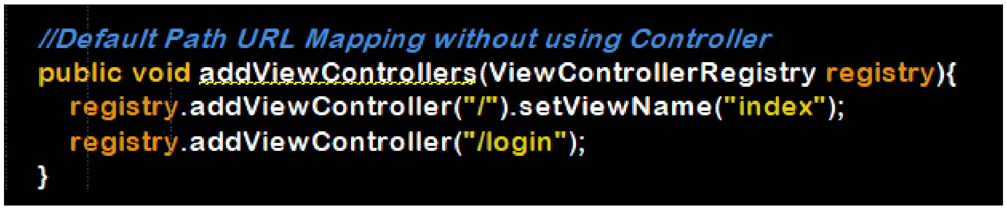
The second method, known as *AUTHORIZATION*, is a method override of the first one. It will allow us to determine which users will be allowed into our application per path and restrict http requests according to controller mapping by user role:



```
@Override
//Restrict application URLs
protected void configure(HttpSecurity http) throws Exception {
   http.authorizeRequests()
         //Restrict requests to following paths and their subpaths (/**) :
         .antMatchers("/add/**", "/update/**", "/delete")
         //To only users:
            .hasRole("ADMIN")
         .antMatchers("/")
            .hasAnyRole("USER","ADMIN")
         ;
}
```

We will restrict authorized requests to be made only by a user with the role ADMIN for the /add, /update and /delete mappings and their submappings. All of the other roles are allowed to access the index page.

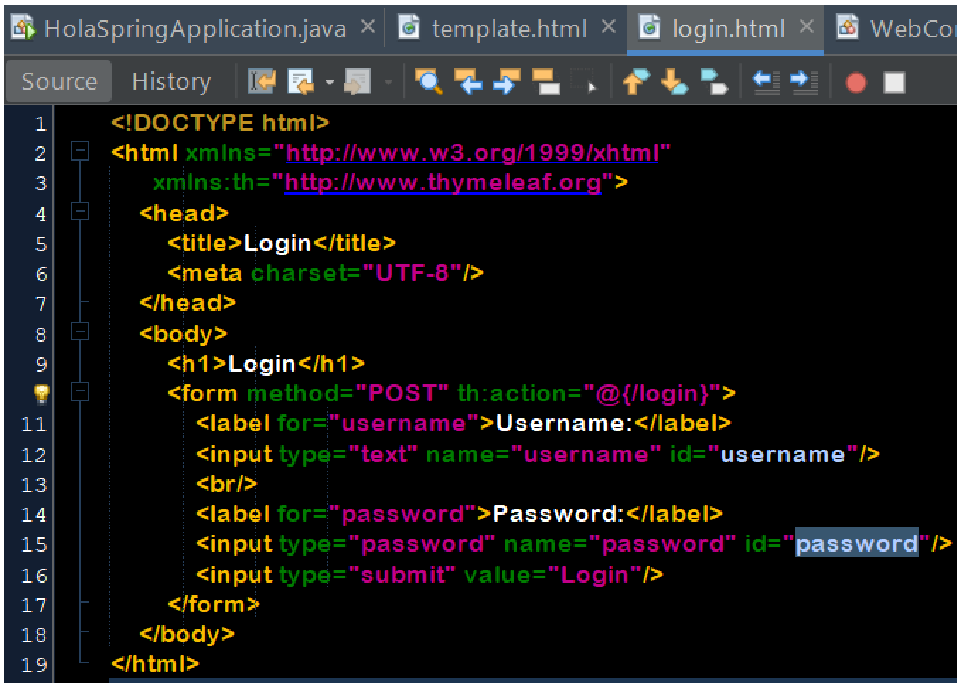
To achieve this, we must first create a default url mapping that doesn't go through our Controller, but uses an implementation of addViewControllers method from WebMvcConfigurer interface (Our Webconfig class).



```
//Default Path URL Mapping without using Controller
public void addViewControllers(ViewControllerRegistry registry){
   registry.addViewController("/").setViewName("index");
   registry.addViewController("/login");
}
```

The first view we get is the index page, which won't be dislayed until we login. So we need to add another view called login, for this action to be performed in correct order. This view will override the previous default login page provided by Spring.

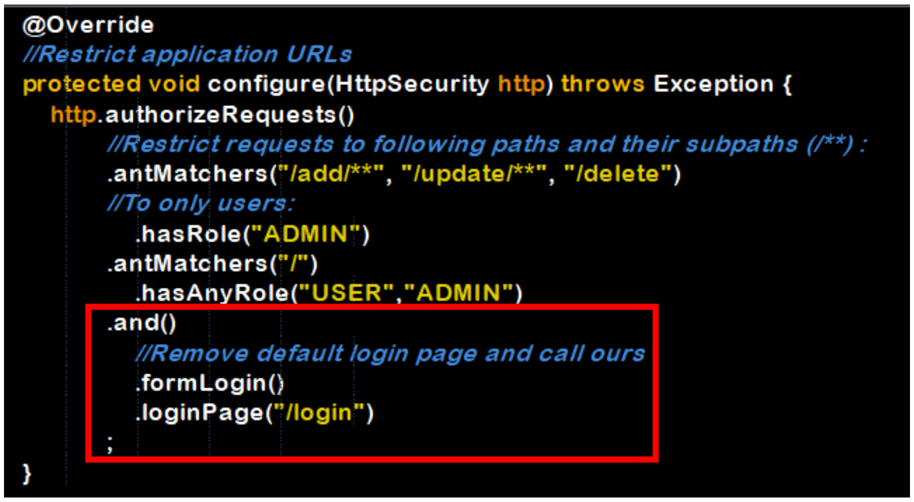
Create a simple login form page:



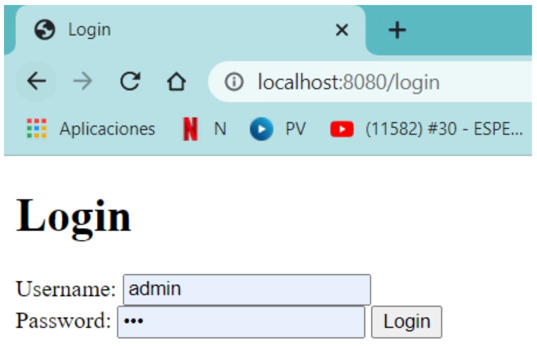
```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
    <head>
        <title>Login</title>
        <meta charset="UTF-8"/>
    </head>
    <body>
        <h1>Login</h1>
        <form method="POST" th:action="@{/login}">
            <label for="username">Username:</label>
            <input type="text" name="username" id="username"/>
            <br/>
            <label for="password">Password:</label>
            <input type="password" name="password" id="password"/>
            <input type="submit" value="Login"/>
        </form>
    </body>
</html>
```

Remember login and logout actions must be performed through POST method and the action mapped on this side has to be @{/login} or @{/logout}.

Of course, Spring Security should also agree with this. Provide the new login page in the list of authorized http requests:



```
@Override
//Restrict application URLs
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
            //Restrict requests to following paths and their subpaths (/**) :
            .antMatchers("/add/**", "/update/**", "/delete")
            //To only users:
                .hasRole("ADMIN")
            .antMatchers("/")
                .hasAnyRole("USER","ADMIN")
            .and()
                //Remove default login page and call ours
                .formLogin()
                .loginPage("/login")
            ;
}
```

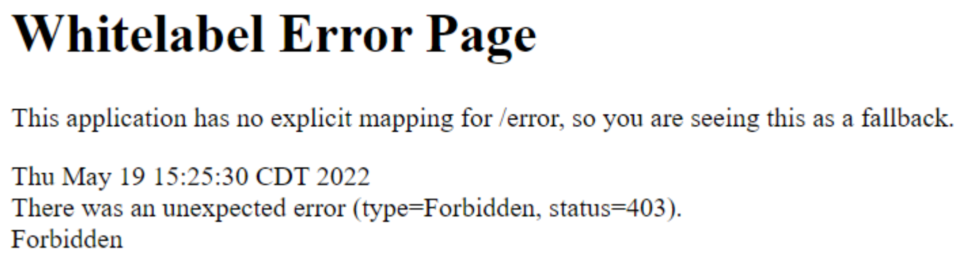


Login

Username: admin
Password: •••
Login

Login using credentials to test the rest of the configurations given.

For example, a regular user must not be able to create a new person and insert it into DB:

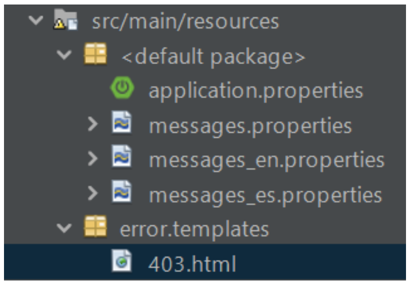


Whitelabel Error Page

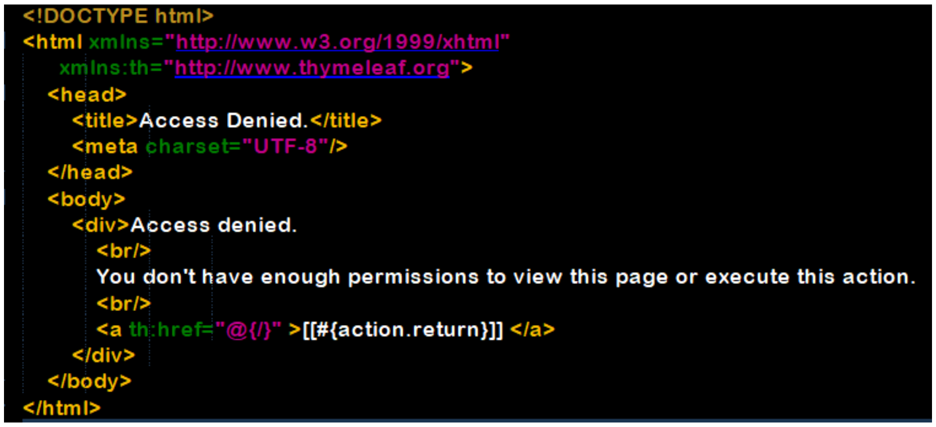
This application has no explicit mapping for /error, so you are seeing this as a fallback.

Thu May 19 15:25:30 CDT 2022
There was an unexpected error (type=Forbidden, status=403).
Forbidden

Let's create a custom error page for this kind of situation. Create a package since many errors may arise and we'll need to address them all:



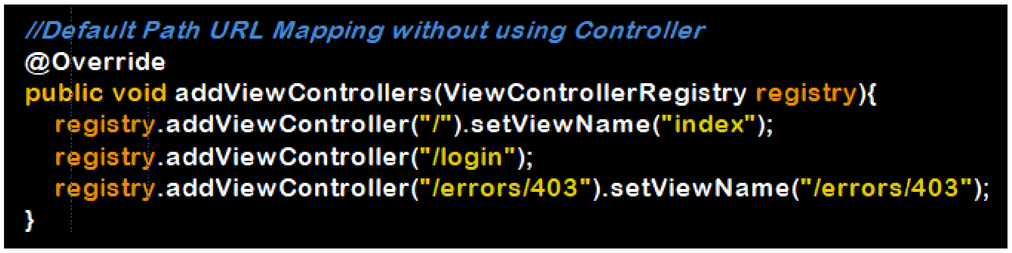
403.html:



```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:th="http://www.thymeleaf.org">
  <head>
    <title>Access Denied.</title>
    <meta charset="UTF-8"/>
  </head>
  <body>
    <div>Access denied.
      <br/>
      You don't have enough permissions to view this page or execute this action.
      <br/>
      <a th:href="@{/}" >[[#{action.return}]] </a>
    </div>
  </body>
</html>
```

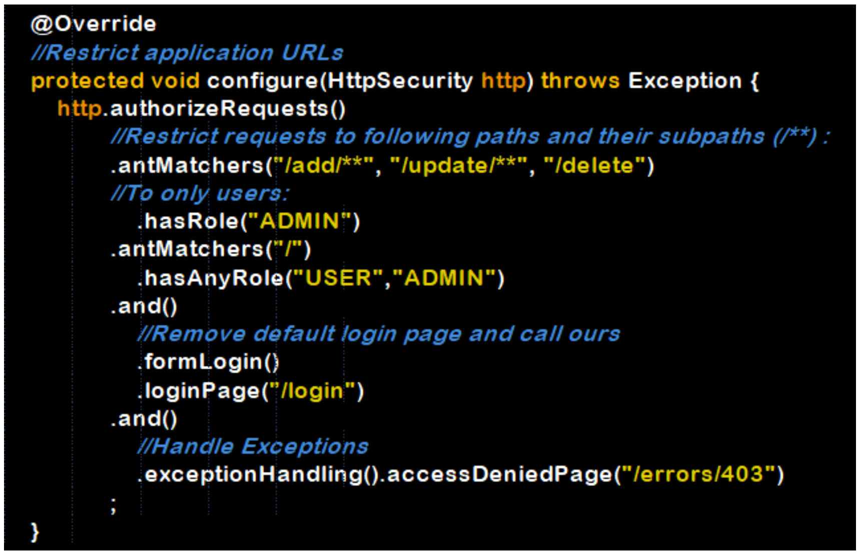
Remember this error page will be displayed without going through any controller. We need to add it to WebConfig's addViewControllers method to map it and authorize it from SecurityConfig's authorizerRequests method as well:

WebConfig.java:



```
//Default Path URL Mapping without using Controller
@Override
public void addViewControllers(ViewControllerRegistry registry){
   registry.addViewController("/").setViewName("index");
   registry.addViewController("/login");
   registry.addViewController("/errors/403").setViewName("/errors/403");
}
```

SecurityConfig.java:

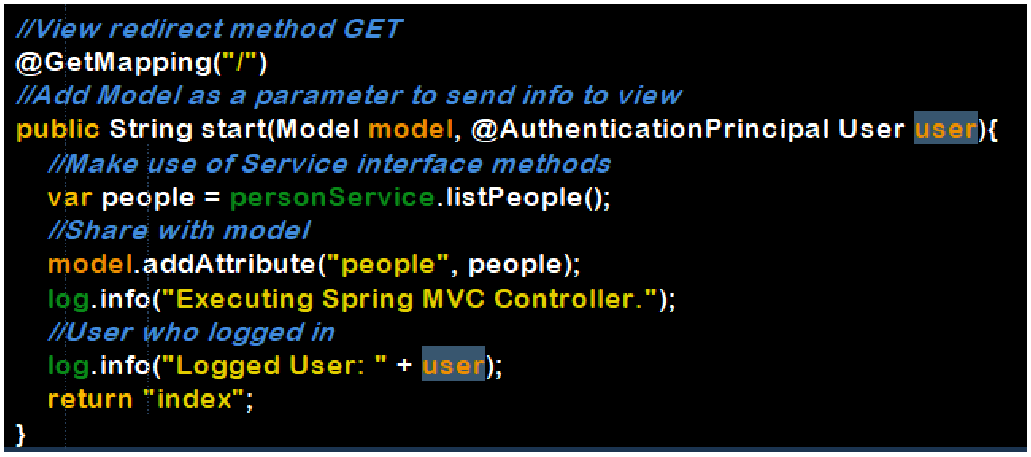


```
@Override
//Restrict application URLs
protected void configure(HttpSecurity http) throws Exception {
   http.authorizeRequests()
         //Restrict requests to following paths and their subpaths (/**) :
         .antMatchers("/add/**", "/update/**", "/delete")
         //To only users:
            .hasRole("ADMIN")
         .antMatchers("/")
            .hasAnyRole("USER","ADMIN")
         .and()
            //Remove default login page and call ours
            .formLogin()
            .loginPage("/login")
         .and()
            //Handle Exceptions
            .exceptionHandling().accessDeniedPage("/errors/403")
         ;
}
```

Now that we have achieved our login and restricted access by user to several view pages, it's time to give back control to our actual Controller class.

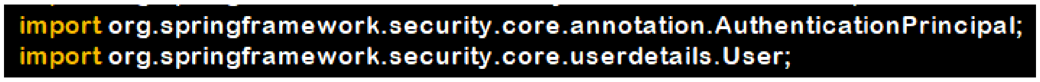
First of all, we need to retrieve the user back from Spring Security.

Annotate the User with @AuthenticationPrincipal to retrieve user from Spring Security login.



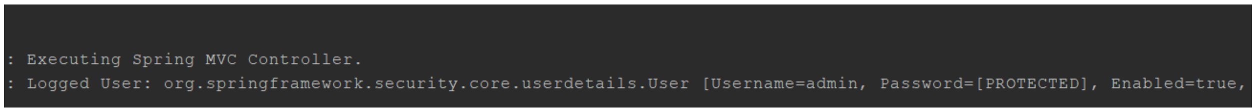
```java
//View redirect method GET
@GetMapping("/")
//Add Model as a parameter to send info to view
public String start(Model model, @AuthenticationPrincipal User user){
   //Make use of Service interface methods
   var people = personService.listPeople();
   //Share with model
   model.addAttribute("people", people);
   log.info("Executing Spring MVC Controller.");
   //User who logged in
   log.info("Logged User: " + user);
   return "index";
}
```

Be careful to import from proper package:



```java
import org.springframework.security.core.annotation.AuthenticationPrincipal;
import org.springframework.security.core.userdetails.User;
```

We can now see in our log, the logged user credentials:



```
: Executing Spring MVC Controller.
: Logged User: org.springframework.security.core.userdetails.User [Username=admin, Password=[PROTECTED], Enabled=true,
```