In this example, we will learn how to create a form that, through simple requests, obtains a list of employees in a Spring-boot MVC Web application, using Spring MVC 5, maven, thymeleaf and a development IDE such as Netbeans.

In this application, a Java-based configuration will be used, which is implemented through the self-managed configuration of the Spring-boot framework using only Java configuration (not xml).

USED ​​TOOLS AND TECHNOLOGIES

•Spring MVC - 5.1.0 RELEASE

•Spring Boot

•JDK - 1.8 or later

•Maven - 3.5.1

•Apache Tomcat - 8.5

•IDE – Netbeans 8

•Thymeleaf

Here are 11 steps for developing our application in Spring-boot MVC.

1.Create Web Application - Maven

2.Dependencies - pom.xml file

3. Project structure

4.Spring boot configuration – DemoApplication.java

5.Initialization of the data repository. Application.properties

6.Model Class - EmployeeEntity.java

7.Controller class - EmployeeMvcController.java

8.Views – add-edit-employee - list-employees

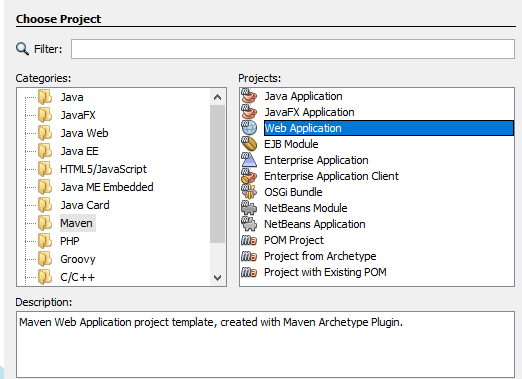
9.Repository and service

10.Build and Run the application

11.Demo

Web application creation – Maven

It can be created either using the command line through maven or from a development IDE, in this case from Netbeans.



Once the web project has been created by maven, the next step is to attach the dependencies to use.

Dependencies - pom.xml file

A project object model or POM is the fundamental unit of work in Maven. It is an XML file that contains information about the project and the configuration details used by Maven to build the project.

Contains default values ​​for most projects.

Examples of this is the build directory, which is the target ; the source directory, which is src/main/java ; the test source directory, which is src/test/java ; and so. When executing a task or target, Maven looks for the POM in the current directory. It reads the POM, gets the necessary configuration information, and then executes the target.

The project dependencies are as follows, found in the pom.xml file:



Note that we are depending on a spring-boot-starter-web library, which is used for a Spring MVC Web application via spring-boot.

We also find spring-boot-starter-data-jpa that is used to make query requests through a relational database in memory h2, and the thymeleaf template framework that accompanies our application for the realization of the visual layer.

The H2 database library is added, which will allow us to manage a relational data repository in memory to store the information that we are going to display in the CRUD application.

In the diagram, as suggested, a Model-View-Controller approach is used.

•Model – EmployeeEntity.java

•Views – add-edit-employee.html – list-employees.html

•Controller – EmployeeMvcController.java

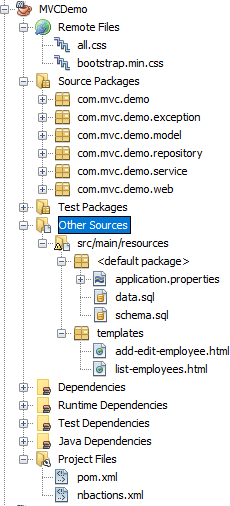
Additional packages are exposed where we can find the configuration of the application and the access classes to the database.

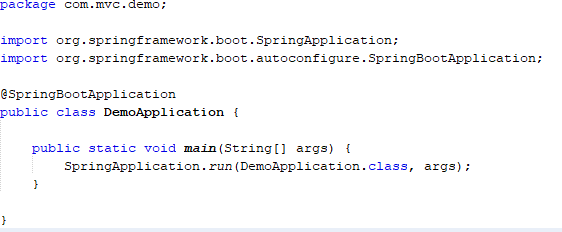
The next step will be configuring Spring Beans using a Java-based configuration.

# project structure

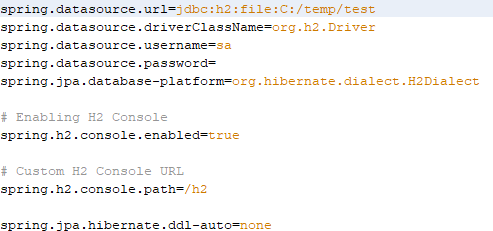
Spring boot configuration – DemoApplication.java

A class called DemoApplication.java is created that gives the entry point to the Spring-boot framework, in said class the famous entry point is created

Main which allows through the SpringApplication.run instruction to start the application. In this class we also find the @SpringBootApplication annotation, annotation that allows us to scan the class through CDI and start the application



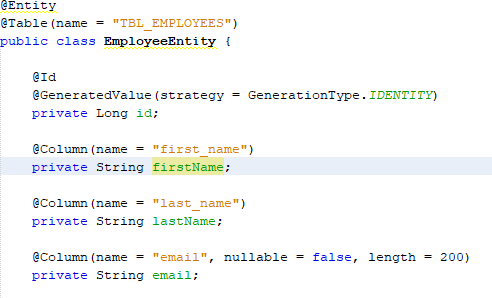
Initialization of the data repository. Application.properties

In this file, the connection to the information repository is configured, in this case, a relational database in memory H2. This file describes the URL to the database, the driver used, the user, the password, the Orm, in this case Hibernate and other configuration.

Next step, we will create an EmployeeEntity class model.

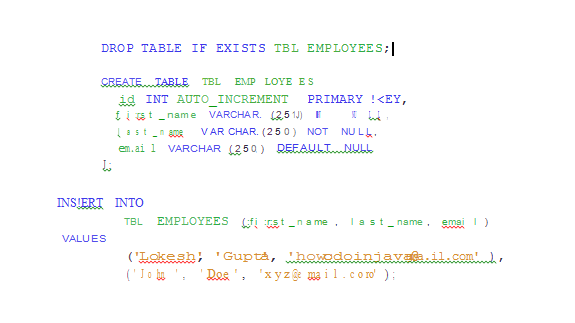
Model Class – EmployeeEntity.java

We create the Java EmployeeEntity class to bind data to the model and render model data in views. It is worth noting the addition of JPA's own annotations to link our ORM (Hibernate) to the application.



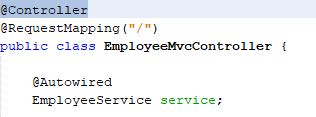
To highlight the use of the @Entity annotation, which represents the table to which we are going to connect to consult the information, @Id and @GenerateValue that indicate the main key of the table and the generation strategy, as indicated which is an antonumeric column. And the @Column annotation, which indicates the columns of the table and their characteristics. All this so that the ORM knows how to map the information from the database to our application through the java JPA specification.

In the resources folder, along with the application.properties file, there are two sql artifacts, data.sql and schema.sql; which allow you to create the table used and add base information for the query.



Controller class – EmployeeMvcController.java

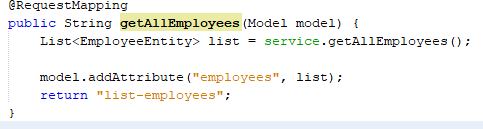
Let's create the EmployeeMvcController controller class annotated with the @Controller annotation as follows:



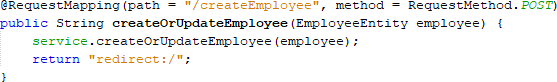
The @Controller annotation, as its name indicates, allows us to establish that it is a front controller class towards our view. The @RequestMapping annotation allows you to map the URL from the view to the methods of the controller class.

Additionally, a service is injected through CDI, which is the artifact that allows the business logic to be carried out and calls the relational information repository H2.

In this controller class, the calling methods are made with a particular mapping, for example, when you need to fetch all the employees from the database, the getAllEmployees method is implemented, which receives a model and returns a list of employees, which is injects into the returned view, which is list-employees.html



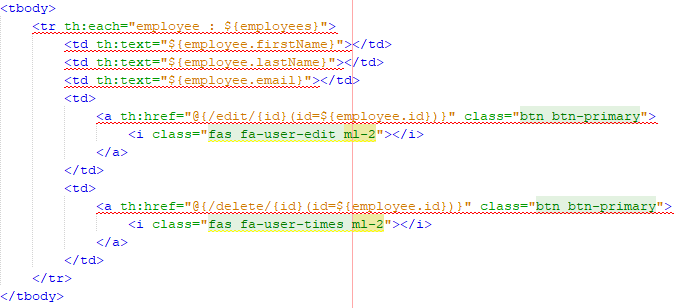
For example, for one of the Crud methods, the implementation of createOrUpdateEmployee is made, in said method the model with the employee's data is received as a parameter; so that later, through the use of the service artifact, it is consulted if the employee exists, then the data is updated, otherwise one more employee would be created.



So far we have created the model and the controller, now in the next step, we will create views.

Views – list-employees.html – add-edit-employee.html

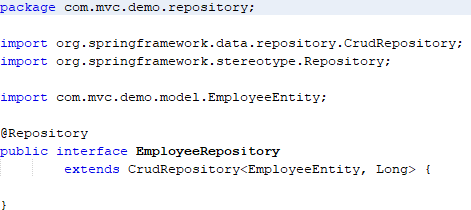
The view is a simple page that shows the information of the employees when making the request from the view.



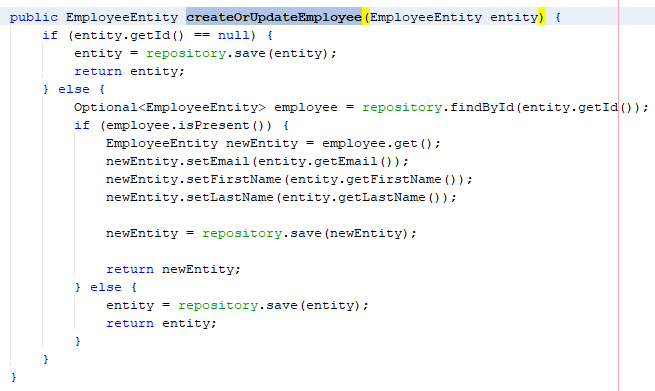
In the case of the initial query of our application, the information of all the employees that exist in the database is extracted. As the image shows, through the use of thymeleaf templates, the employees are brought in the employees variable, iterates through them, and the information of each of them is placed in the table that is shown as result in view. Two buttons are also implemented, one for navigation when editing an employee, which goes to the next view, and the other to delete the employee, which makes a direct request to our controller.

Repository and Service

The EmployeeRepository.java class extends CrudRepository, which is a framework master class with the base implementation of crud. To do this we pass the entity type to persist. In this EmployeeEntity case, the type of the primary key that makes a record in the table unique is also added.



And the EmployeeService.java class that is used to carry out our business logic, in this case calling the repository class and making access requests to the database used. For example, the implementation of a createOrUpdateEmployee method is made, which within its logic allows creating or updating a specific employee that is passed as a parameter in the method. This logic first determines if the employee already exists, for its respective update or otherwise allows a new one to persist with the information sent.



Build and Run the application

Since we are using the maven build tool, we will first need to build this application using the following maven command:

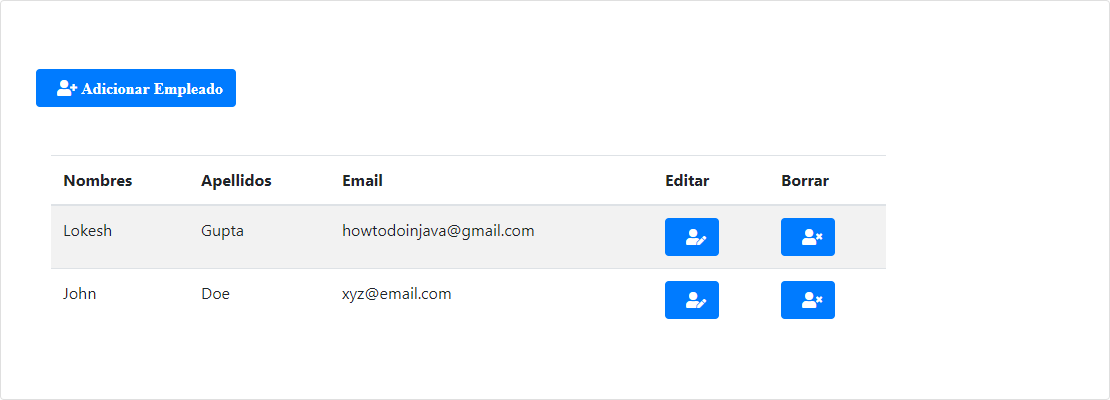
clean install

Once the compilation is successful, we will run this application on the Tomcat 8.5 server in the IDE or we can also simply click on the resulting jar file. Let us remember that it is a Spring-boot application, which allows self-deployment with everything necessary for its execution, including the web server.

show

Once an application is up and running on the Tomcat server or by clicking on the jar, hit this link in the browser: http://localhost:8080

By entering the URL, you will see the following page.



If the edit or add employee button is pressed, the request is made to the controller of our application, and there the following implemented view is returned, showing us the information of the employee to update or create



respectively.

ADDITIONAL

Add a functionality that they choose freely.