Gustavo Ponce Follow
Java Developer
Nov 18, 2016 · 5 min read

Spring Boot + Spring MVC + Spring Security + MySQL

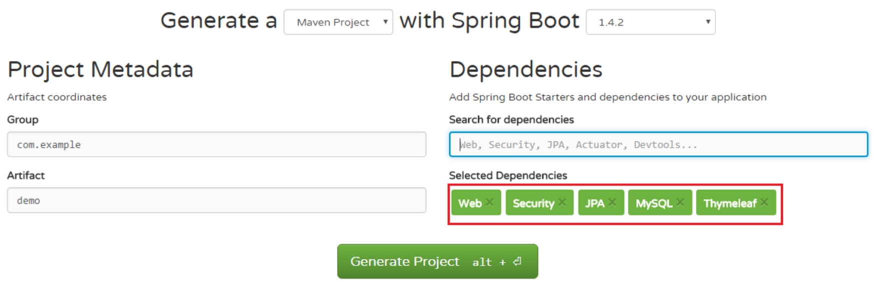
This tutorial will show you how to implement a Login process using the following tech stack:

- Spring Boot(1.4.2)
- Spring Security
- Spring MVC
- JPA
- Thymeleaf
- MySQL (5.7.11)
- Bootstrap (UI Presentation)
- Maven (3.3.9)
- Eclipse (Neon, 4.6.0)
- Java 8
- Packaging (JAR)

Project Creation

First we will use the Spring initializer page to create our maven project with the dependencies listed above.

1. Go to → https://start.spring.io/
2. Leave everything as it is and select the following dependencies: **Web, JPA, Security, MySQL,** and **Thymeleaf**.


Generate a Maven Project with Spring Boot 1.4.2

Project Metadata

Artifact coordinates

Group: com.example

Artifact: demo

Dependencies

Add Spring Boot Starters and dependencies to your application

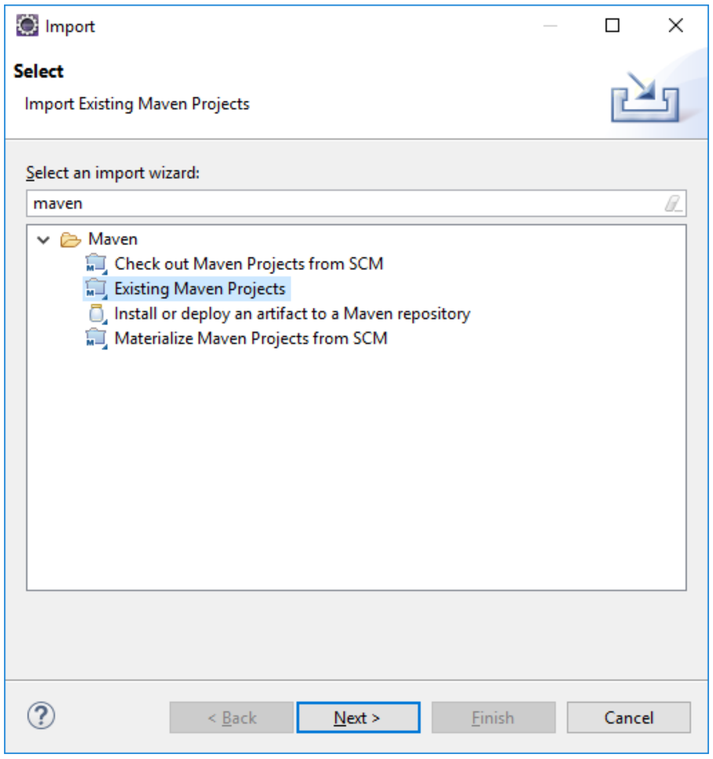
Search for dependencies: Web, Security, JPA, Actuator, Devtools...

Selected Dependencies: Web, Security, JPA, MySQL, Thymeleaf

Click on Generate Project button and it will download a zip file (demo.zip) with our maven project.

Import Project into Eclipse

1. Unzip the zip file.
2. Import into Eclipse as “**Existing Maven Project**”
3. Choose the root directory of the project generated (where the pom.xml file is located) and click on Finish.

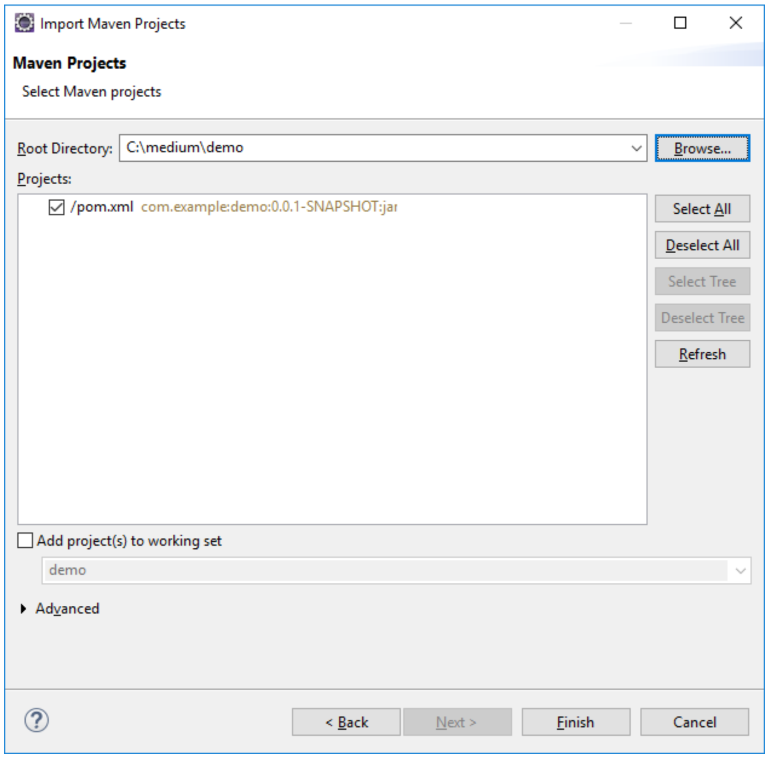

Import

Select

Import Existing Maven Projects

Select an import wizard: maven

- Maven
  - Check out Maven Projects from SCM
  - Existing Maven Projects
  - Install or deploy an artifact to a Maven repository
  - Materialize Maven Projects from SCM


Import Maven Projects

Maven Projects

Select Maven projects

Root Directory: C:\medium\demo

Projects:

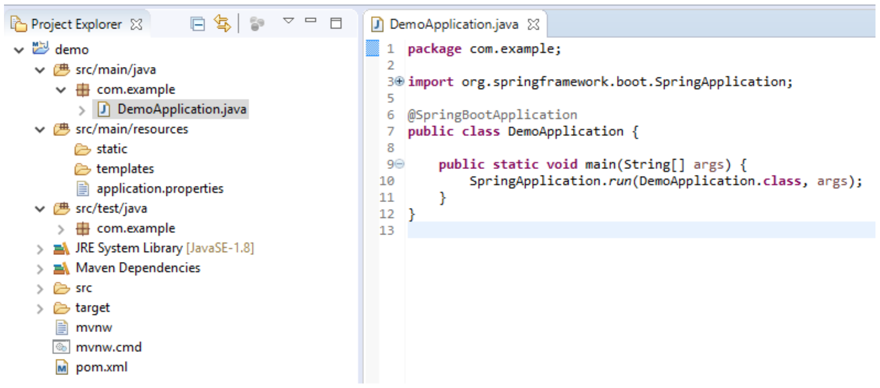
☑ /pom.xml com.example:demo:0.0.1-SNAPSHOT:jar

☐ Add project(s) to working set

demo

Advanced

It will display the next project structure.



```java
package com.example;

import org.springframework.boot.SpringApplication;

@SpringBootApplication
public class DemoApplication {

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

Note: In order to execute Thymeleaf in "LEGACYHTML5" mode, we need to add an extra dependency in our pom.xml file → **nekohtml.**

Also we need to add the following properties in the applicaiton.properties file (please refer this file below or in the github repository).

- spring.thymeleaf.mode=LEGACYHTML5
- spring.thymeleaf.cache=false

pom.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xml
        xsi:schemaLocation="http://maven.apache.org/PO
        <modelVersion>4.0.0</modelVersion>

        <groupId>com.example</groupId>
        <artifactId>demo</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <packaging>jar</packaging>

        <name>demo</name>
        <description>Demo project for Spring Boot</des

        <parent>
                <groupId>org.springframework.boot</gro
                <artifactId>spring-boot-starter-parent
                <version>1.4.2.RELEASE</version>
                <relativePath /> <!-- lookup parent fr
        </parent>

        <properties>
                <project.build.sourceEncoding>UTF-8</p
                <project.reporting.outputEncoding>UTF-
                <java.version>1.8</java.version>
        </properties>

        <dependencies>
                <dependency>
                        <groupId>org.springframework.b
                        <artifactId>spring-boot-starte
                </dependency>
                <dependency>
                        <groupId>org.springframework.b
                        <artifactId>spring-boot-starte
                </dependency>
                <dependency>
                        <groupId>org.springframework.b
                        <artifactId>spring-boot-starte
                </dependency>
                <dependency>
                        <groupId>org.springframework.b
```

```
                                <artifactId>spring-boot-starte
                        </dependency>

                        <dependency>
                                <groupId>mysql</groupId>
                                <artifactId>mysql-connector-ja
                                <scope>runtime</scope>
```

Model Creation

Now let´s create our model classes called User and Role(Entity classes).

User

This class includes the fields validations based on the validations provided by Hibernate.

```
package com.example.model;

import java.util.Set;

import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.ManyToMany;
import javax.persistence.Table;

import org.hibernate.validator.constraints.Email;
import org.hibernate.validator.constraints.Length;
import org.hibernate.validator.constraints.NotEmpty;
import org.springframework.data.annotation.Transient;

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

        @Id
        @GeneratedValue(strategy = GenerationType.AUT
        @Column(name = "user_id")
        private int id;
        @Column(name = "email")
        @Email(message = "*Please provide a valid Ema
        @NotEmpty(message = "*Please provide an email
        private String email;
        @Column(name = "password")
        @Length(min = 5, message = "*Your password mu
        @NotEmpty(message = "*Please provide your pas
        @Transient
        private String password;
        @Column(name = "name")
        @NotEmpty(message = "*Please provide your nam
        private String name;
        @Column(name = "last_name")
```

```
	@NotEmpty(message = "*Please provide your las
	private String lastName;
	@Column(name = "active")
	private int active;
	@ManyToMany(cascade = CascadeType.ALL)
	@JoinTable(name = "user_role", joinColumns =
	private Set<Role> roles;

	public int getId() {
		return id;
	}

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

	public String getPassword() {
		return password;
	}

	public void setPassword(String password) {
		this.password = password;
	}

	public String getName() {
		return name;
```

Role

```
package com.example.model;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "role")
public class Role {
        @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
        @Column(name="role_id")
        private int id;
        @Column(name="role")
        private String role;

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

Data Layer (JPA Repositories)

UserRepository

```
package com.example.repository;

import org.springframework.data.jpa.repository.JpaRepo
import org.springframework.stereotype.Repository;

import com.example.model.User;

@Repository("userRepository")
```

RoleRepository

```
package com.example.repository;

import org.springframework.data.jpa.repository.JpaRepo
import org.springframework.stereotype.Repository;

import com.example.model.Role;

@Repository("roleRepository")
public interface RoleRepository extends JpaRepository<
```

Service Layer

Now let´s create our user service layer(interface and implementation). We will inject the UserRepository, RoleRepository and the BCryptPasswordEncoder into our service implementation.

Interface

```
package com.example.service;

import com.example.model.User;

public interface UserService {
        public User findUserByEmail(String email);
```

Implementation

```
package com.example.service;

import java.util.Arrays;
import java.util.HashSet;

import org.springframework.beans.factory.annotation.Au
import org.springframework.security.crypto.bcrypt.BCry
import org.springframework.stereotype.Service;

import com.example.model.Role;
import com.example.model.User;
import com.example.repository.RoleRepository;
import com.example.repository.UserRepository;

@Service("userService")
public class UserServiceImpl implements UserService{

        @Autowired
        private UserRepository userRepository;
        @Autowired
    private RoleRepository roleRepository;
    @Autowired
    private BCryptPasswordEncoder bCryptPasswordEncode

        @Override
        public User findUserByEmail(String email) {
```

Configuration Files

WebMvcConfig.java

This class defines the password encoder that we just injected in the service layer.

```
package com.example.configuration;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configur
import org.springframework.security.crypto.bcrypt.BCry
import org.springframework.web.servlet.config.annotati

@Configuration
public class WebMvcConfig extends WebMvcConfigurerAdap

        @Bean
        public BCryptPasswordEncoder passwordEncoder()
```

SecurityConfiguration.java

```
package com.example.configuration;

import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Au
import org.springframework.beans.factory.annotation.Va
import org.springframework.context.annotation.Configur
import org.springframework.security.config.annotation.
import org.springframework.security.config.annotation.
import org.springframework.security.config.annotation.
import org.springframework.security.config.annotation.
import org.springframework.security.config.annotation.
import org.springframework.security.crypto.bcrypt.BCry
import org.springframework.security.web.util.matcher.A

@Configuration
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurity

        @Autowired
        private BCryptPasswordEncoder bCryptPasswordEn

        @Autowired
        private DataSource dataSource;

        @Value("${spring.queries.users-query}")
        private String usersQuery;

        @Value("${spring.queries.roles-query}")
        private String rolesQuery;

        @Override
        protected void configure(AuthenticationManager
                        throws Exception {
                auth.
                        jdbcAuthentication()
                                .usersByUsernameQuery(
                                .authoritiesByUsername
                                .dataSource(dataSource
                                .passwordEncoder(bCryp
        }
```

```
42
43          @Override
44          protect
45
```

This class is where the security logic is implemented, let´s analyze the code.

- **Line 21** → password encoder reference implemented in WebMvcConfig.java
- **Line 24** → data source implemented out of the box by Spring Boot. We only need to provide the database information in the application.properties file (please see the reference below).
- **Lines 27 and 30** → Reference to user and role queries stored in application.properties file (please see the reference below).
- **Lines from 33 to 41** → AuthenticationManagerBuilder provides a mechanism to get a user based on the password encoder, data source, user query and role query.
- **Lines from 44 to 61** → Here we define the antMatchers to provide access based on the role(s) (lines 48 to 51), the parameters for the login process (lines 55 to 56), the success login page(line 53), the failure login page(line 53), and the logout page (line 58).
- **Lines from 64 to 68** → Due we have implemented Spring Security we need to let Spring knows that our resources folder can be served skipping the antMatchers defined.

Note: There is an alternative to implement the AuthenticationManagerBuilder implementing the UserDetailsService interface in your User Repository.

Evaluate your necessities and implement based on your requirements.

Interface:
org.springframework.security.core.userdetails.UserDetailsService

Here the code for the UserDetailsService authentication strategy.

gustavoponce7/SpringSecurityUserDetailsSe

rvice

Contribute to SpringSecurityUserDetailsService development by creating an account on GitHub.

github.com

application.properties file

Basically the idea of this file is to setup the configurations in a property file instead of a xml file or a java configuration class.

```
# ===============================
# = DATA SOURCE
# ===============================
spring.datasource.url = jdbc:mysql://localhost:3306/sp
spring.datasource.username = root
spring.datasource.password = admin
spring.datasource.testWhileIdle = true
spring.datasource.validationQuery = SELECT 1

# ===============================
# = JPA / HIBERNATE
# ===============================
spring.jpa.show-sql = true
spring.jpa.hibernate.ddl-auto = update
spring.jpa.hibernate.naming-strategy = org.hibernate.c
spring.jpa.properties.hibernate.dialect = org.hibernat

# ===============================
# = Thymeleaf configurations
```

Note: Update with your Database credentials.

If you want to see the complete reference of the application.properties file, please refer the next page.

Appendix A. Common application properties

banner.charset=UTF-8
banner.location=classpath:banner.txt
banner.image.location=classpath:banner.gif
banner.image.width...

docs.spring.io

Controller Layer

MVC Logic

```
package com.example.controller;

import javax.validation.Valid;

import org.springframework.beans.factory.annotation.Au
import org.springframework.security.core.Authenticatic
import org.springframework.security.core.context.Secur
import org.springframework.stereotype.Controller;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.Request
import org.springframework.web.bind.annotation.Request
import org.springframework.web.servlet.ModelAndView;

import com.example.model.User;
import com.example.service.UserService;

@Controller
public class LoginController {

        @Autowired
        private UserService userService;

        @RequestMapping(value={"/", "/login"}, method
        public ModelAndView login(){
                ModelAndView modelAndView = new ModelA
                modelAndView.setViewName("login");
                return modelAndView;
        }


        @RequestMapping(value="/registration", method
        public ModelAndView registration(){
                ModelAndView modelAndView = new ModelA
                User user = new User();
                modelAndView.addObject("user", user);
                modelAndView.setViewName("registration
                return modelAndView;
        }

        @RequestMapping(value = "/registration", metho
        public ModelAndView createNewUser(@Valid User
                ModelAndView modelAndView = new Mode
                User userExists = userSer
                if (userExist
```

By default Spring Boot defines the view resolver in the next way.

- **Prefix** → resources/templates
- **Suffix** → html

Note: if you want to implement a custom view resolver you can do it using the application.properties file or the a java configuration file.

View Layer

login.html

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
        xmlns:th="http://www.thymeleaf.org">

<head>
        <title>Spring Security Tutorial</title>
        <link rel="stylesheet" type="text/css" th:href
        <link rel="stylesheet" href="https://maxcdn.bo
        <script src="https://ajax.googleapis.com/ajax/
        <script src="https://maxcdn.bootstrapcdn.com/b
</head>

<body>
        <form th:action="@{/registration}" method="get
                <button class="btn btn-md btn-warning 
        </form>

        <div class="container">
                <img th:src="@{/images/login.jpg}" cla
                <form th:action="@{/login}" method="PO
                        <h3 class="form-signin-heading
                        <br/>

                        <input type="text" id="email" 
```

registration.html

```
<!DOCTYPE html>
<html lang="en" xmlns="http://www.w3.org/1999/xhtml"
        xmlns:th="http://www.thymeleaf.org">
<head>
        <title>Registration Form</title>
        <link rel="stylesheet" type="text/css" th:href
        <link rel="stylesheet" href="https://maxcdn.bo
        <script src="https://ajax.googleapis.com/ajax/
        <script src="https://maxcdn.bootstrapcdn.com/b
</head>
<body>
        <form th:action="@{/}" method="get">
                <button class="btn btn-md btn-warning
        </form>

        <div class="container">
                <div class="row">
                        <div class="col-md-6 col-md-of
                                <form autocomplete="of
                                        th:object="${u
                                        role="form">
                                        <h2>Registrati
                                        <div class="fo
                                                <div c
                                                <label

                                                <input

                                                </div>
                                        </div>

                                        <div class="fo
                                                <div c
                                                <label



                                                </div>
                                        </div>
                                        <div class="fo
                                                <div c
```

SQL Scripts

Database Schema

```
--
-- Table structure for table `role`
--

DROP TABLE IF EXISTS `role`;
/*!40101 SET @saved_cs_client     = @@character_set_cl
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE `role` (
  `role_id` int(11) NOT NULL AUTO_INCREMENT,
  `role` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`role_id`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;
/*!40101 SET character_set_client = @saved_cs_client *


--
-- Table structure for table `user`
--

DROP TABLE IF EXISTS `user`;
/*!40101 SET @saved_cs_client     = @@character_set_cl
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE `user` (
  `user_id` int(11) NOT NULL AUTO_INCREMENT,
  `active` int(11) DEFAULT NULL,
  `email` varchar(255) NOT NULL,
  `last_name` varchar(255) NOT NULL,
  `name` varchar(255) NOT NULL,
  `password` varchar(255) NOT NULL,
  PRIMARY KEY (`user_id`)
```

Role insert

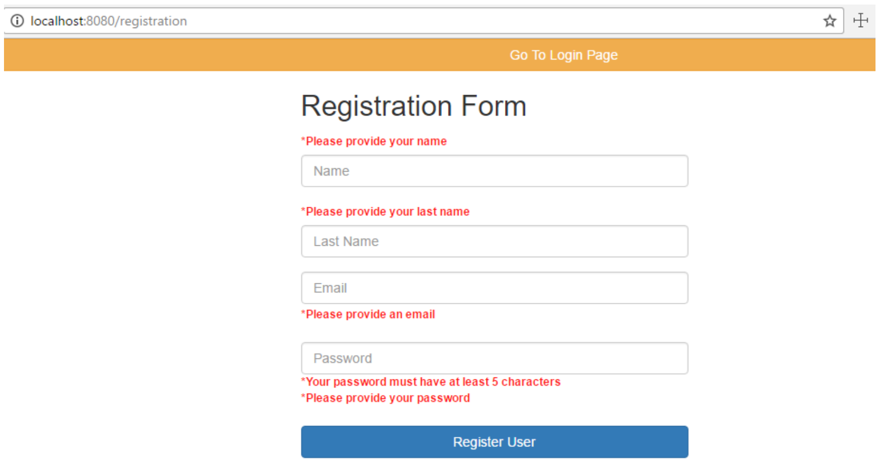
```sql
INSERT INTO `role` VALUES (1,'ADMIN');
```

Note: By default Spring Boot will create the database structure if you have provided in the right way your MySQL credentials in the application.properties file, basically you only need to insert the admin role manually.

Register new user

http://localhost:8080/registration

Validations


localhost:8080/registration

Go To Login Page

Registration Form

*Please provide your name

Name

*Please provide your last name

Last Name

Email

*Please provide an email

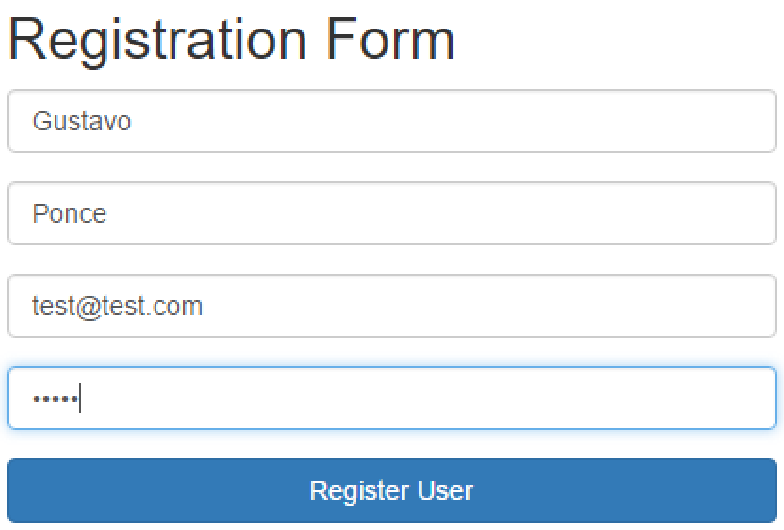
Password

*Your password must have at least 5 characters

*Please provide your password

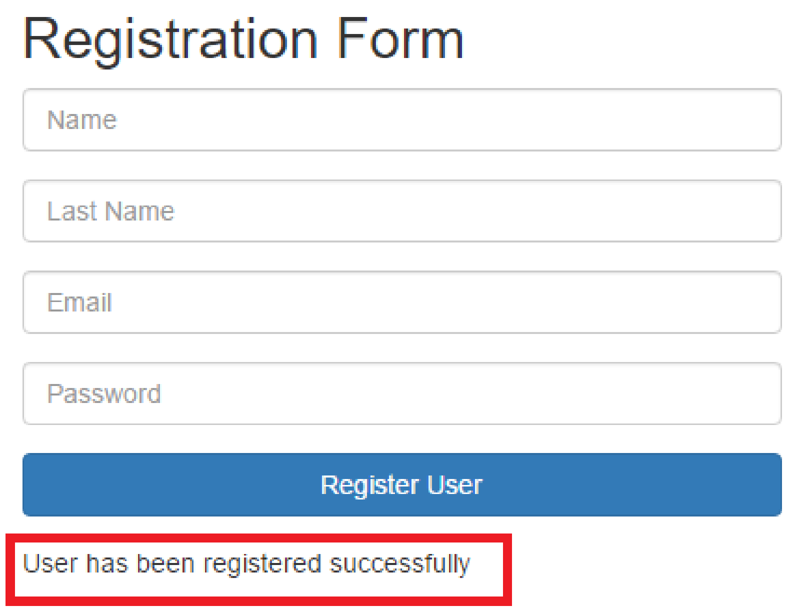
Register User

User Registration



Registration Form

Gustavo
Ponce
test@test.com
•••••

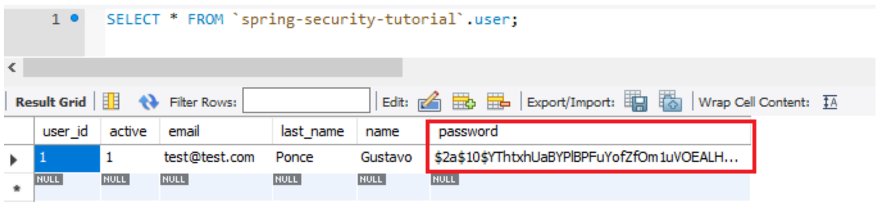


Registration Form

Name
Last Name
Email
Password

User has been registered successfully

As you can see the password has been stored with a **Hash algorithm** due we have implemented the BCryptPasswordEncoder in our AuthenticationManagerBuilder.



```
SELECT * FROM `spring-security-tutorial`.user;
```

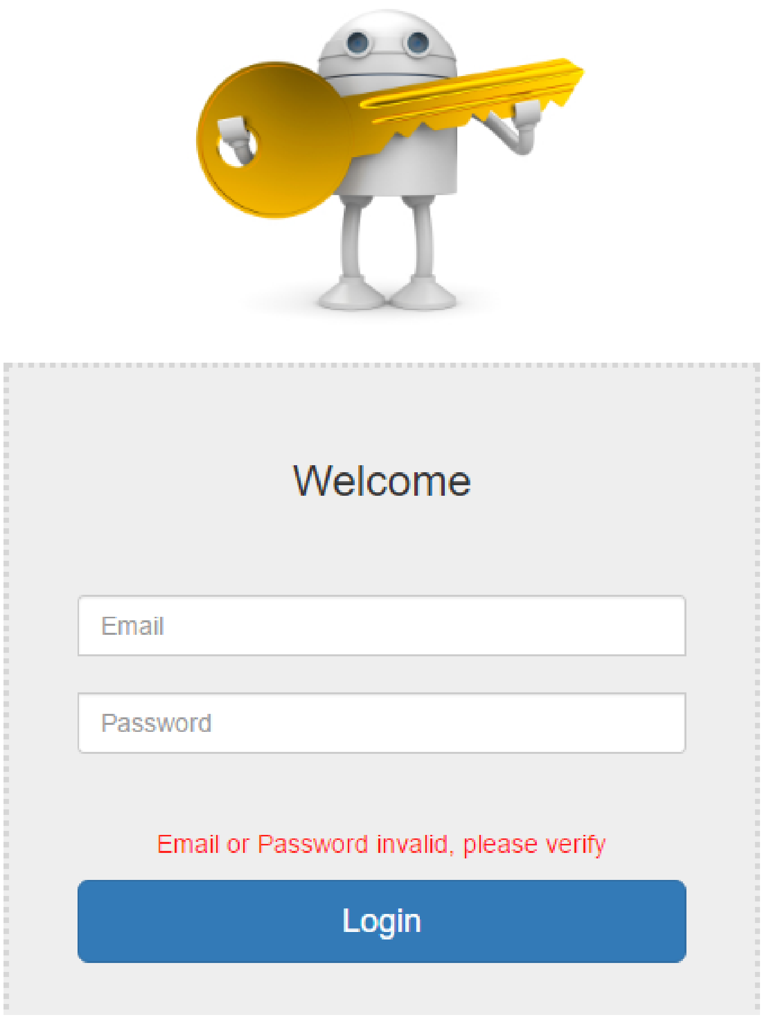
user_id	active	email	last_name	name	password
1	1	test@test.com	Ponce	Gustavo	$2a$10$YThtxhUaBYPlBPFuYofZfOm1uVOEALH...
NULL	NULL	NULL	NULL	NULL	NULL

Note: Don´t forget to insert the ADMIN role into the database, otherwise you will get an exception.

Login Process

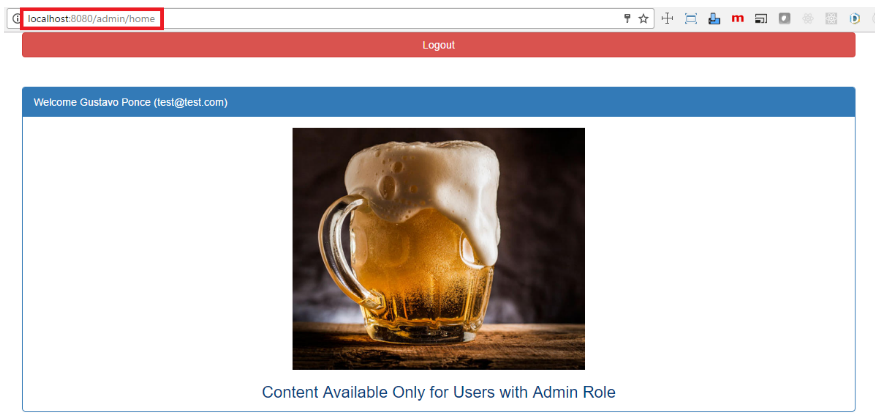
http://localhost:8080/login

Login Fail



Email or Password invalid, please verify

Login Success



Welcome Gustavo Ponce (test@test.com)

Content Available Only for Users with Admin Role

That´s all folks, as you can see we have implemented a Login process from scratch including password hash strategy. BTW never store passwords in a plain text.

If you have any question or feedback don't hesitate to write your thoughts in the responses section.

Github Repository

gustavoponce7/SpringSecurityLoginTutorial

Contribute to SpringSecurityLoginTutorial development by creating an account on GitHub.

github.com