Spring Security Spring Boot Login REST API

In this tutorial, you will learn how to build login or sign-in REST API using Spring boot, Spring Security, Hibernate, and MySQL database.

In this tutorial, we are going to use Spring Boot 3 and Spring Security 6.

Tools and Technologies Used

- Spring Boot 3
- JDK - 1.8 or later
- Spring MVC
- Spring Security
- Hibernate
- Maven
- Spring Data JPA
- IDE - Eclipse or Spring Tool Suite (STS) or Intellij IDEA **// Any IDE works**
- MYSQL

1. Create Spring boot application

Spring Boot provides a web tool called **Spring Initializer** to bootstrap an application quickly. Just go to **https://start.spring.io/** and generate a new spring boot project.

Use the below details in the Spring boot creation:

Project Name: springboot-blog-rest-api

Project Type: Maven

Choose dependencies: Spring Web, Lombok, Spring Data JPA, Spring Security, Dev Tools, and MySQL Driver

Package name: net.javaguides.springboot

Packaging: Jar

Download the Spring Boot project as a zip file, unzip it and import it in your favorite IDE.

2. Maven Dependencies

Here is the pom.xml file for your reference:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>3.0.0</version>
                <relativePath/> <!-- lookup parent from repository -->
        </parent>
        <groupId>com.springboot.blog</groupId>
        <artifactId>springboot-blog-rest-api</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <name>springboot-blog-rest-api</name>
        <description>Spring boot blog application rest api&apos;s</description>
        <properties>
                <java.version>17</java.version>
        </properties>
        <dependencies>
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-data-jpa</artifactId>
                </dependency>
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-web</artifactId>
                </dependency>

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

```
                        <optional>true</optional>
                </dependency>
                <dependency>
                        <groupId>mysql</groupId>
                        <artifactId>mysql-connector-java</artifactId>
                        <scope>runtime</scope>
                </dependency>
                <dependency>
                        <groupId>org.projectlombok</groupId>
                        <artifactId>lombok</artifactId>
                        <optional>true</optional>
                </dependency>
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-validation</artifactId>
                </dependency>
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-security</artifactId>
                </dependency>
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-test</artifactId>
                        <scope>test</scope>
                </dependency>
        </dependencies>

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

        <groupId>org.projectlombok</groupId>

        <artifactId>lombok</artifactId>
                                                </exclude>
                                        </excludes>
                                </configuration>
                        </plugin>
                </plugins>
        </build>

</project>
```

3. Configure MySQL Database

Let's first create a database in MySQL server using the below command:

```
create database myblog
```

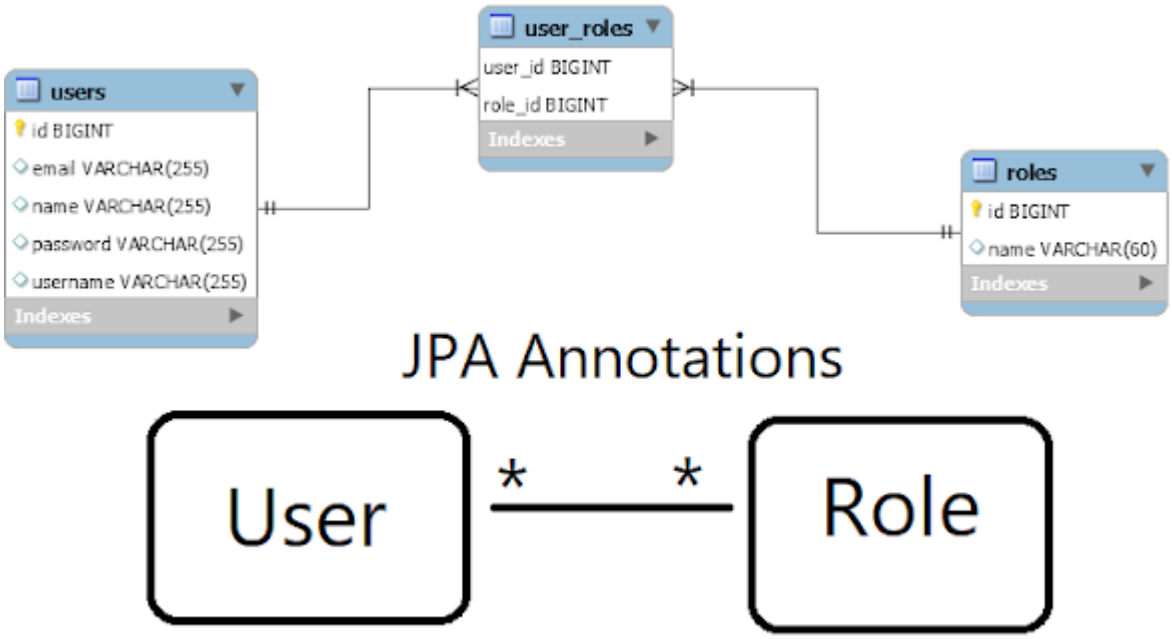
Since we're using MySQL as our database, we need to configure the database **URL**, **username**, and **password** so that Spring can establish a connection with the database on startup.

Open `src/main/resources/application.properties` file and add the following properties to it:

```
spring.datasource.url =
jdbc:mysql://localhost:3306/myblog?useSSL=false&serverTimezone=UTC
spring.datasource.username = root
spring.datasource.password = root

# hibernate properties
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQLDialect

# Hibernate ddl auto (create, create-drop, validate, update)
spring.jpa.hibernate.ddl-auto = update

logging.level.org.springframework.security=DEBUG
```

4. Model Layer - Create JPA Entities

In this step, we will create *`User`* and *`Role`* JPA entities and establish **MANY-to-MANY** relationships between them. Let's use JPA annotations to establish **MANY-to-MANY** relationships between *`User`* and *`Role`* entities.



User JPA Entity

```
package com.springboot.blog.entity;

import lombok.Data;

import jakarta.persistence.*;
import java.util.Set;

@Data
@Entity
@Table(name = "users", uniqueConstraints = {
        @UniqueConstraint(columnNames = {"username"}),
        @UniqueConstraint(columnNames = {"email"})
})
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    private String name;
    private String username;
    private String email;
    private String password;

    @ManyToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
    @JoinTable(name = "user_roles",
        joinColumns = @JoinColumn(name = "user_id", referencedColumnName = "id"),
        inverseJoinColumns = @JoinColumn(name = "role_id", referencedColumnName =
"id"))
    private Set<Role> roles;
}
```

Role JPA Entity

```
package com.springboot.blog.entity;

import lombok.Getter;
import lombok.Setter;

import jakarta.persistence.*;

@Setter
@Getter
@Entity
@Table(name = "roles")
public class Role {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    @Column(length = 60)
    private String name;
}
```

5. Repository Layer

UserRepository

```
package com.springboot.blog.repository;

import com.springboot.blog.entity.User;
import org.springframework.data.domain.Example;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.Optional;

public interface UserRepository extends JpaRepository<User, Long> {
    Optional<User> findByEmail(String email);
    Optional<User> findByUsernameOrEmail(String username, String email);
    Optional<User> findByUsername(String username);
    Boolean existsByUsername(String username);
    Boolean existsByEmail(String email);
}
```

RoleRepository

```
package com.springboot.blog.repository;

import com.springboot.blog.entity.Role;
```

```
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.Optional;

public interface RoleRepository extends JpaRepository<Role, Long> {
    Optional<Role> findByName(String name);
}
```

6. Service Layer - CustomUserDetailsService

Let's write a logic to load user details by name or email from the database.

Let's create a *CustomUserDetailsService* which implements

the *UserDetailsService* interface (Spring security in-build interface) and provides

an implementation for the *loadUserByUsernamе()* method:

```
import com.springboot.blog.entity.User;
import com.springboot.blog.repository.UserRepository;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

import java.util.Set;
import java.util.stream.Collectors;

@Service
public class CustomUserDetailsService implements UserDetailsService {

    private UserRepository userRepository;

    public CustomUserDetailsService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @Override
    public UserDetails loadUserByUsername(String usernameOrEmail) throws
UsernameNotFoundException {
          User user = userRepository.findByUsernameOrEmail(usernameOrEmail,
usernameOrEmail)
                 .orElseThrow(() ->
                         new UsernameNotFoundException("User not found with
username or email: "+ usernameOrEmail));

        Set<GrantedAuthority> authorities = user
                .getRoles()
                .stream()
```

```
                .map((role) -> new
SimpleGrantedAuthority(role.getName())).collect(Collectors.toSet());

        return new
org.springframework.security.core.userdetails.User(user.getEmail(),
                user.getPassword(),
                authorities);
    }
}
```

Spring Security uses the `UserDetailsService` interface, which contains the `loadUserByUsername(String username)` method to lookup `UserDetails` for a given `username`. The `UserDetails` interface represents an authenticated user object and Spring Security provides an out-of-the-box implementation of `org.springframework.security.core.userdetails.User`.

7. Spring Security Configuration

Let's create a class *`SecurityConfig`* and add the following configuration to it:

```
package com.springboot.blog.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpMethod;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.config.Customizer;
import
org.springframework.security.config.annotation.authentication.configuration.Authen
ticationConfiguration;
import
org.springframework.security.config.annotation.method.configuration.EnableMethodSe
curity;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.provisioning.InMemoryUserDetailsManager;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableMethodSecurity
public class SecurityConfig {

    private UserDetailsService userDetailsService;

    public SecurityConfig(UserDetailsService userDetailsService){
        this.userDetailsService = userDetailsService;
```

```
    }

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

    @Bean
    public AuthenticationManager authenticationManager(
                                    AuthenticationConfiguration configuration) throws
Exception {
        return configuration.getAuthenticationManager();
    }

    @Bean
    SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

        http.csrf().disable()
                .authorizeHttpRequests((authorize) ->
                        //authorize.anyRequest().authenticated()
                        authorize.requestMatchers(HttpMethod.GET,
"/api/**").permitAll()
                                .requestMatchers("/api/auth/**").permitAll()
                                .anyRequest().authenticated()

                );

        return http.build();
    }
}
```

In Spring Security 5.6, we can enable annotation-based security using the *@EnableMethodSecurity* annotation on any @Configuration instance. *@EnableMethodSecurity* enables @PreAuthorize, @PostAuthorize, @PreFilter, and @PostFilter by default.

We are allowing anyone to access login REST API with the below security configuration:

```
authorize.requestMatchers(HttpMethod.GET, "/api/**").permitAll()
```

We are using the Spring security provided *BCryptPasswordEncoder* class to encrypt the passwords.

8. DTO or Payload Classes

Let's create DTO classes to transfer data or payload between client and server and vice-versa.

LoginDto

```
package com.springboot.blog.payload;

import lombok.Data;

@Data
public class LoginDto {
    private String usernameOrEmail;
    private String password;
}
```

9. Controller Layer - Login/Sign-in and Register/SignUp REST API's

Now it's time to code Login/Sign-in and Register/SignUp REST APIs. Let's create a class AuthController and add the following code to it:

```
package com.springboot.blog.controller;

import com.springboot.blog.payload.LoginDto;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.security.authentication.AuthenticationManager;
import
org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.Collections;

@RestController
@RequestMapping("/api/auth")
public class AuthController {

    @Autowired
    private AuthenticationManager authenticationManager;

    @PostMapping("/signin")
```

```
    public ResponseEntity<String> authenticateUser(@RequestBody LoginDto
loginDto){
        Authentication authentication = authenticationManager.authenticate(new
UsernamePasswordAuthenticationToken(
                loginDto.getUsernameOrEmail(), loginDto.getPassword()));

        SecurityContextHolder.getContext().setAuthentication(authentication);
        return new ResponseEntity<>("User signed-in successfully!.",
HttpStatus.OK);
    }
}
```

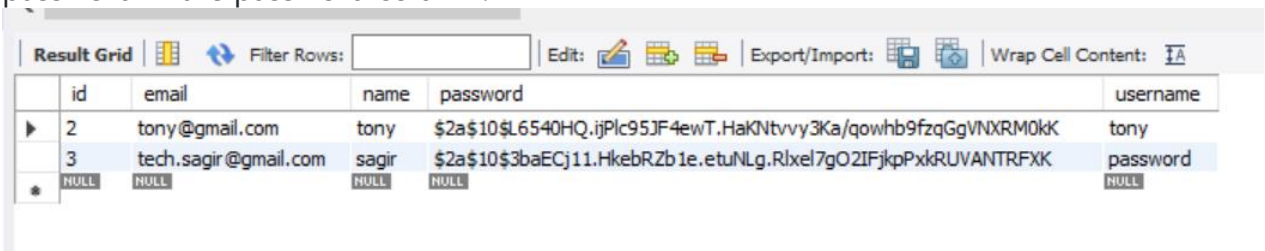
In the above code, we're using the *@RestController* annotation to indicate that this is a controller that handles HTTP requests and returns data directly to the client. The *@RequestMapping* annotation specifies the base URL for all requests to this controller.

10. Generate Encrypted Password

Use the below code snippet to create an encrypted password:

```
package com.springboot.blog.utils;

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

public class PasswordEncoderGenerator {
    public static void main(String[] args) {
        PasswordEncoder passwordEncoder = new BCryptPasswordEncoder();
        System.out.println(passwordEncoder.encode("admin"));
    }
}
```

Add an entry in the **users** table and make sure that you will add an encrypted password in the password column:



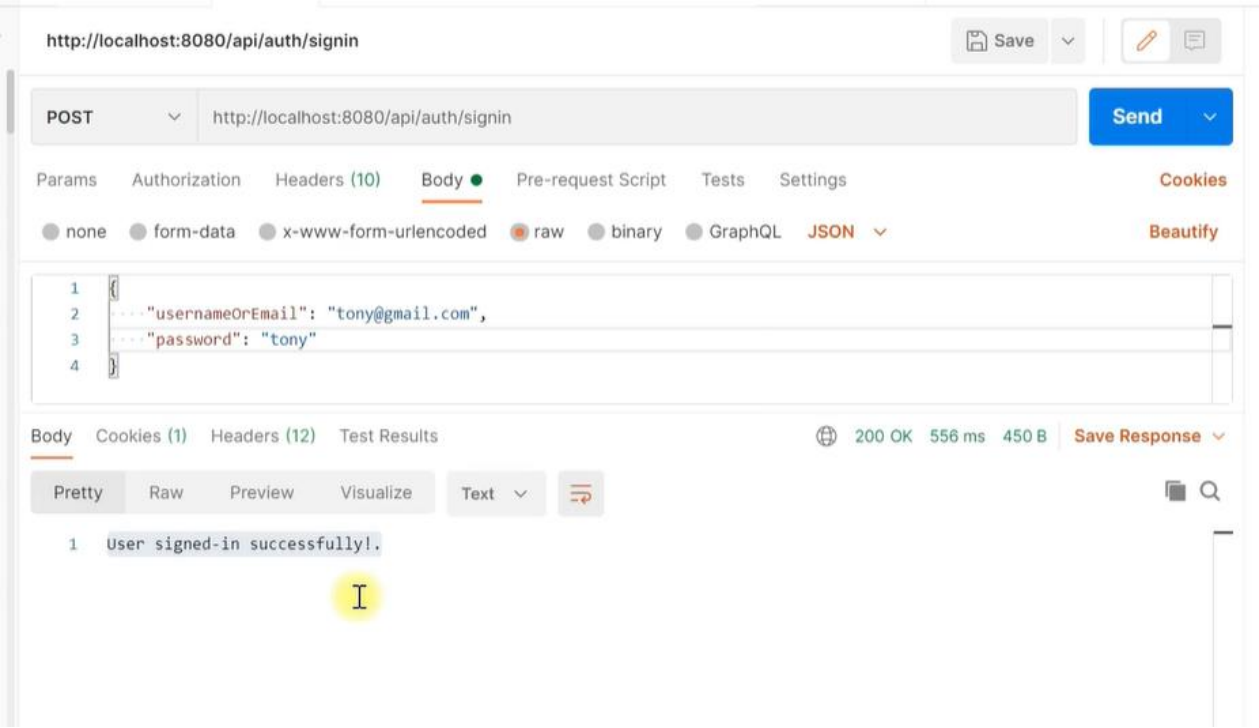
Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

id	email	name	password	username
2	tony@gmail.com	tony	$2a$10$L6540HQ.ijPlc95JF4ewT.HaKNtvvy3Ka/qowhb9fzqGgVNXRM0kK	tony
3	tech.sagir@gmail.com	sagir	$2a$10$3baECj11.HkebRZb1e.etuNLg.Rlxel7gO2IFjkpPxkRUVANTRFXK	password
NULL	NULL	NULL	NULL	NULL

11. Test using Postman

Refer to the below screenshots to test Login and Registration REST API using Postman:

SignIn/Login REST API:



GitHub Repository

https://github.com/RameshMF/springboot-blog-rest-api