**1. Secure REST APIs**

**1.1 Create User, Role and User-Role Tables using SQL**

use mydb;

CREATE TABLE USER

(

ID INT NOT NULL AUTO\_INCREMENT,

FIRST\_NAME VARCHAR(20),

LAST\_NAME VARCHAR(20),

EMAIL VARCHAR(20),

PASSWORD VARCHAR(256),

PRIMARY KEY (ID),

UNIQUE KEY (EMAIL)

);

CREATE TABLE ROLE

(

ID INT NOT NULL AUTO\_INCREMENT,

NAME VARCHAR(20),

PRIMARY KEY (ID)

);

CREATE TABLE USER\_ROLE(

USER\_ID int,

ROLE\_ID int,

FOREIGN KEY (user\_id) REFERENCES user(id),

FOREIGN KEY (role\_id) REFERENCES role(id)

);

insert into user(first\_name,last\_name,email,password) values ('doug','bailey','doug@bailey.com','$2a$10$U2STWqktwFbvPPsfblVeIuy11vQ1S/0LYLeXQf1ZL0cMXc9HuTEA2');

insert into user(first\_name,last\_name,email,password) values ('john','ferguson','john@ferguson.com','$2a$10$YzcbPL.fnzbWndjEcRkDmO1E4vOvyVYP5kLsJvtZnR1f8nlXjvq/G');

insert into role values(1,'ROLE\_ADMIN');

insert into role values(2,'ROLE\_USER');

insert into user\_role values(1,1);

insert into user\_role values(2,2);

select \* from user;

select \* from role;

select \* from user\_role;

**1.2 Create User, Role and User-Role Entities using Spring boot**

**User Entity**

package com.bharath.springcloud.model;

import java.util.Set;

import javax.persistence.Entity;

import javax.persistence.FetchType;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.JoinColumn;

import javax.persistence.JoinTable;

import javax.persistence.ManyToMany;

@Entity

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String firstName;

private String lastName;

private String email;

private String password;

@ManyToMany(fetch = FetchType.EAGER)

@JoinTable(name = "user\_role", joinColumns = @JoinColumn(name = "user\_id"), inverseJoinColumns = @JoinColumn(name = "role\_id"))

private Set<Role> roles;

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

public String getFirstName() {

return firstName;

}

public void setFirstName(String firstName) {

this.firstName = firstName;

}

public String getLastName() {

return lastName;

}

public void setLastName(String lastName) {

this.lastName = lastName;

}

public String getEmail() {

return email;

}

public void setEmail(String email) {

this.email = email;

}

public String getPassword() {

return password;

}

public void setPassword(String password) {

this.password = password;

}

public Set<Role> getRoles() {

return roles;

}

public void setRoles(Set<Role> roles) {

this.roles = roles;

}

}

We should define the relationship between the user an role entity.

User entity has set of role because user will have multiple roles. We have to mark this with many to many annotation. In a many to many relationship, these roles will be loaded lazily so i want this to be fetched by using fetch type as eager by default.

@ManyToMany(fetch = FetchType.EAGER)

private Set<Role> roles;

We can define users in side role entity same as above and specify mappedBy as roles to join.

@ManyToMany(mappedBy = “roles”)

private Set<User> users;

By using @JoinTable annotation we can join both entities by creating new entity user\_role.

@JoinTable(name = "user\_role", joinColumns = @JoinColumn(name = "user\_id"), inverseJoinColumns = @JoinColumn(name = "role\_id"))

**Role Entity**

package com.bharath.springcloud.model;

import java.util.Set;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.ManyToMany;

import org.springframework.security.core.GrantedAuthority;

@Entity

public class Role implements GrantedAuthority {

private static final long serialVersionUID = 1L;

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

@ManyToMany(mappedBy = "roles")

private Set<User> users;

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

@Override

public String getAuthority() {

return name;

}

}

The Role entity should implement the granted authority interface from spring security.

The getAuthority( ) method returns the role name. Whenever the get authority name is invoked on a particular role instance, we should return back name of the role.

We can add serializable generated one if we want to get rid of that warning there had a default serialization id.

**1.3 Create user and role repos**

**UserRepo.java**

package com.bharath.springcloud.repos;

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

import com.bharath.springcloud.model.User;

public interface UserRepo extends JpaRepository<User, Long> {

User findByEmail(String email);

}

In user repo, we need to define a finder method that will retrieve the user details by email.

**RoleRepo.java**

package com.bharath.springcloud.repos;

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

import com.bharath.springcloud.model.Role;

public interface RoleRepo extends JpaRepository<Role, Long> {

}

**1.4 Custom user details service**

We need implement a custom user details service that will implement the user details interface.

Create new class UserDetailsServiceImpl and implements UserDetailsService from spring security.

Mark this class as service by using @Service.

There is only one method inside of this interface called LoadUserByUserName, and we need to provide username.

We receive a user name, whatever the client sends in the request for authentication, that user name will be handled over to this load user by this user name and we need to return user details object from spring security that user details will be used by the authentication provider within spring security. It will do the authentication for us.

So to fetch the user details based on the user name, we have to use the user repo so mark it as auto wired annotation.

We returns instance of inbuilt spring security user along with our custom user details.

**return new org.springframework.security.core.userdetails.User(user.getEmail(), user.getPassword(), user.getRoles());**

**Example:**

package com.bharath.springcloud.security;

import org.springframework.beans.factory.annotation.Autowired;

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

import org.springframework.security.core.userdetails.UserDetailsService;

import org.springframework.security.core.userdetails.UsernameNotFoundException;

import org.springframework.stereotype.Service;

import com.bharath.springcloud.model.User;

import com.bharath.springcloud.repos.UserRepo;

**@Service**

public class UserDetailsServiceImpl implements UserDetailsService {

**@Autowired**

UserRepo userRepo;

**@Override**

public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

User user = userRepo.findByEmail(username);

if (user == null) {

throw new UsernameNotFoundException("User not found for email" + username);

}

return new org.springframework.security.core.userdetails.User(user.getEmail(), user.getPassword(),

user.getRoles());

}

}

**1.5 Create the Web security config class**

Create web security config class and extends Web security configurer adapter from spring security.

It has 2 configure methods out of 3.

The first one is takes authentication manager builder and configures the authentication.

Here we will pass our own user details service. Autowire that customized user details service impl.

Now spring security knows that it should be using this user details we have provided.

Next configure the password encoder by defining a bean.

The second configure method use the HTTP security.

Here we secure the URLs and methods.

We use HTTP basic authentication for now.

For authorization we use authorizedRequests( ) methods. Here we can use different types of matchers for authorization.

Mvc matchers is preferable. Take Http & Url string pattern parameterized method.

For first parameter of mvc matcher use HTTP method enum from spring boot and use GET method.

For second parameter use end ponts as we want then check access for these end points.

Use has role method provide authorized role to those end points i.e any user role person can access.

If we need one more mvc mathers we can write after this.

We will write here for post HTTP method. We provide access to only admin role as we want.

For GET endpoints access we can use another method hasAnyRole( ) method. Here I given access to both user and admin by using this method.

We need to disable the CSRF(Cross Site Request Forgery) check by default. If it is enabled, will not be able to perform a post operation from postman tool.

We use and( ) method for multiple conditions.

If we use anyRequest( ).denyAll( ) after role authorization then it will deny access to remaining end points. So we can access what end points we mentioned in matchers only.

We can use patterns like \*\* or any customized patterns to specify the url.

We can use ant matchers but these are not intelligent enough to figure out URL patterns and also not support extensions as well.

**Example:**

package com.bharath.springcloud.security.config;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;

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.annotation.authentication.builders.AuthenticationManagerBuilder;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

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

import org.springframework.web.cors.CorsConfiguration;

import org.springframework.web.cors.CorsConfigurationSource;

import com.bharath.springcloud.security.UserDetailsServiceImpl;

//@Configuration

public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

**@Autowired**

UserDetailsServiceImpl userDetailsService;

**@Override**

protected void configure(AuthenticationManagerBuilder auth) throws Exception {

auth.userDetailsService(userDetailsService);

}

**@Override**

protected void configure(HttpSecurity http) throws Exception {

http.httpBasic( );

http.authorizeRequests()

.mvcMatchers(HttpMethod.GET, "/couponapi/coupons/{code:^[A-Z]\*$}", "/index", "/showGetCoupon",

"/getCoupon", "/couponDetails")

.hasAnyRole("USER", "ADMIN")

.mvcMatchers(HttpMethod.GET, "/showCreateCoupon", "/createCoupon", "/createResponse").hasRole("ADMIN")

.mvcMatchers(HttpMethod.POST, "/getCoupon").hasAnyRole("USER", "ADMIN")

.mvcMatchers(HttpMethod.POST, "/couponapi/coupons", "/saveCoupon", "/getCoupon").hasRole("ADMIN")

.mvcMatchers("/", "/login", "/logout", "/showReg", "/registerUser").permitAll().anyRequest().denyAll()

.and().logout().logoutSuccessUrl("/");

}

**@Bean**

public BCryptPasswordEncoder bCryptPasswordEncoder() {

return new BCryptPasswordEncoder();

}

**@Override**

**@Bean**

public AuthenticationManager authenticationManagerBean() throws Exception {

return super.authenticationManagerBean();

}

}

**2. Secure a Web Application**

* 1. **Custome Login UI page**

Open the POM.xml add dependency thyme leaf for UI pages implementations.

All the thyme leaf HTML pages are the templates will live under the source/main/resources.

Right click on the project and go to the properties ,source then click resources and click add folder then choose templates. So it will come out from the resources and display.

To create HTML page right click on templates choose other in new and search HTML file.

When you make changes to the templates if you don’t want to stop and start, you can add a property in the application.properties as follows.

Spring.thymeleaf.cache = false

Create new interface security service that will host the login method that returns Boolean.

It takes user name and password .

Create new class security service impl which implements above security service. Mark it as @Service.

To fetch the user inject the user details service.

By using loadUserByUsername method we can load user by user name.

We need to create a username password authentication token so that we can pass it to the authentication manager.

By using UsernamePasswordAuthenticationToken method we created a token according to given username, password and authorization.

We need authentication manager to authenticate the token.

Once the authentication finishes, the authentication manager will update the token status. Inside a token there is a flag called authenticated.

So we need to check Boolean result of token by using method isAuthenticated( ).

If it is true then authentication was successful else fail.

If result is true, then security context holder set security token into the security context.

Now we have to implement the Log-in view and controller.

Create a class User Controller, it will handle the login for us. Inject security service. Use post mapping for login action.

Go to web security config class, here we don’t need farm log in because we have our own form.

Give permit all for login and root end points in mvc matchers.

**Example:**

**Security Service interface.java**

package com.bharath.springcloud.security;

public interface SecurityService {

boolean login(String userName,String password);

}

**SecurityServiceImpl.java**

package com.bharath.springcloud.security;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.security.authentication.AuthenticationManager;

import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;

import org.springframework.security.core.context.SecurityContextHolder;

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

import org.springframework.security.core.userdetails.UserDetailsService;

import org.springframework.stereotype.Service;

**@Service**

public class SecurityServiceImpl implements SecurityService {

**@Autowired**

UserDetailsService userDetailsService;

**@Autowired**

AuthenticationManager authenticationManger;

**@Override**

public boolean login(String userName, String password) {

UserDetails userDetails = userDetailsService.loadUserByUsername(userName);

UsernamePasswordAuthenticationToken token = new UsernamePasswordAuthenticationToken(userDetails, password,

userDetails.getAuthorities());

authenticationManger.authenticate(token);

boolean result = token.isAuthenticated();

if (result) {

SecurityContextHolder.getContext().setAuthentication(token);

}

return result;

}

}

**UserController .java**

package com.bharath.springcloud.controllers;

import org.springframework.beans.factory.annotation.Autowired;

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

import org.springframework.stereotype.Controller;

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

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

import com.bharath.springcloud.model.User;

import com.bharath.springcloud.repos.UserRepo;

import com.bharath.springcloud.security.SecurityService;

**@Controller**

public class UserController {

**@Autowired**

private SecurityService securityService;

**@GetMapping("/")**

public String showLoginPage() {

return "login";

}

**@PostMapping("/login")**

public String login(String email, String password) {

boolean loginResponse = securityService.login(email, password);

if (loginResponse) {

return "index";

}

return "login";

}

}

**2.2 Log out**

Go to web security config and at the end of configuration add logout( ) then it is simple log out of our application.

There are various things we can pass it logout successful URL. We can also define our own success handlers .

If we set any cookis in the process of browsing application we can delete those cookies by delete cookies method.

When a user logs in spring will internally maintain a session using the J session id. If we don’t want that session then pass false in invalidateHttpSession method.

We don’t have to write any controller methods. By default we have /logout .

**Sample Code:**

.mvcMatchers("/", "/login", "/logout", "/showReg", "/registerUser").permitAll().anyRequest().denyAll().and().logout().logoutSuccessUrl("/");

**2.3 User Registration Process**

While saving user details we should encode the password by using PasswordEncoder.

**Example:**

package com.bharath.springcloud.controllers;

import org.springframework.beans.factory.annotation.Autowired;

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

import org.springframework.stereotype.Controller;

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

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

import com.bharath.springcloud.model.User;

import com.bharath.springcloud.repos.UserRepo;

import com.bharath.springcloud.security.SecurityService;

@Controller

public class UserController {

@Autowired

private SecurityService securityService;

@Autowired

private UserRepo userRepo;

@Autowired

private PasswordEncoder encoder;

@GetMapping("/showReg")

public String showRegistrationPage() {

return "registerUser";

}

@PostMapping("/registerUser")

public String register(User user) {

user.setPassword(encoder.encode(user.getPassword()));

userRepo.save(user);

return "login";

}

@GetMapping("/")

public String showLoginPage() {

return "login";

}

@PostMapping("/login")

public String login(String email, String password) {

boolean loginResponse = securityService.login(email, password);

if (loginResponse) {

return "index";

}

return "login";

}

}