Day13: Spring Security: Authentication using Database, Authorization

Spring Security Project with SpringDataJPA

Step1- Create a Spring Boot project by adding the following Dependencies:

- **Spring Web** − It bundles all dependencies related to web development including Spring MVC(**DispatcherServlet**) , REST, and an **embedded Tomcat server**.
- **Spring Security** − For the implementation of security features provided by Spring Security.
- **Spring Data JPA** − In addition to using all features defined by JPA specification, Spring Data JPA adds its own features such as the no-code implementation of the repository pattern and the creation of database queries from the method name.
- **MySql Driver** − For the MySQL database driver.

Step2: specify the database configuration inside the application.properties file

```
#db specific properties
spring.datasource.url=jdbc:mysql://localhost:3306/masaidb
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.username=root
spring.datasource.password=root

#ORM s/w specific properties
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Step3: Create a Customer Entity class inside com.masai.model package:

Customer.java

```
package com.masai.model;

import com.fasterxml.jackson.annotation.JsonProperty;
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.UniqueConstraint;


@Entity
public class Customer {

  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Integer custId;
  private String name;
```

```
  @Column(unique = true)
  private String email;
  @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
  private String password;
  private String address;

  //getters and setters

}
```

In the above class **@JsonProperty(access = JsonProperty.Access.WRITE_ONLY)** is used so that password will be visible only while writing the Customer properties(while registering a new customer) but not be visible while loading or reading the customer details.

Step 4: Create an interface CustomerRepository against the above Customer class

CustomerRepository .java

```
package com.masai.repository;

import java.util.Optional;
import org.springframework.data.jpa.repository.JpaRepository;
import com.masai.model.Customer;

public interface CustomerRepository extends JpaRepository<Customer, Integer>{

  public Optional<Customer> findByEmail(String email);

}
```

Step 5: Create the implementation of UserDetailsService interface of the Spring Security framework:

Inside this class we will override the loadUserByUsername(-) method and will return the UserDetails object based on the registered Customer using the above CustomerRepository:

CustomerUserDetailsService.java

```
package com.masai.service;

import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.authentication.BadCredentialsException;
import org.springframework.security.core.GrantedAuthority;
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.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

import com.masai.model.Customer;
import com.masai.repository.CustomerRepository;

@Service
public class CustomerUserDetailsService implements UserDetailsService{

  @Autowired
  private CustomerRepository customerRepository;
```

```
  @Override
  public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {


    Optional<Customer> opt= customerRepository.findByEmail(username);

    if(opt.isPresent()) {

      Customer customer= opt.get();

      //Empty Authorities
      List<GrantedAuthority> authorities= new ArrayList<>();
      //authorities.add(new SimpleGrantedAuthority(customer.getRole()));


      return new User(customer.getEmail(), customer.getPassword(), authorities);

      //return new CustomerUserDetails(customer);


    }else
      throw new BadCredentialsException("User Details not found with this username: "+username);
  }

}
```

Note: In this Example we have used the default implementation of **UserDetails** interface i.e. **User** class. but if we want we can define our own implementation of the **UserDetails** interface having our own configuration and uncomment the last commented line from the above code:

Example:

CustomerUserDetails.java

```
package com.masai.service;

import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

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

import com.masai.model.Customer;

public class CustomerUserDetails implements UserDetails {


  private Customer customer;


  public CustomerUserDetails(Customer customer) {
    this.customer = customer;
  }

  @Override
  public Collection<? extends GrantedAuthority> getAuthorities() {

    List<GrantedAuthority> authorities = new ArrayList<>();

    //List<Authority> auths= customer.getAuthorities();

    //for(Authority auth:auths) {
      //SimpleGrantedAuthority simpleGrantedAuthority=new SimpleGrantedAuthority(auth.getName());
      //authorities.add(simpleGrantedAuthority);
    //}

    return authorities;
  }

  @Override
```

```
  public String getPassword() {
    // TODO Auto-generated method stub
    return customer.getPassword();
  }

  @Override
  public String getUsername() {
    // TODO Auto-generated method stub
    return customer.getEmail();
  }

  @Override
  public boolean isAccountNonExpired() {
    // TODO Auto-generated method stub
    return true;
  }

  @Override
  public boolean isAccountNonLocked() {
    // TODO Auto-generated method stub
    return true;
  }

  @Override
  public boolean isCredentialsNonExpired() {
    // TODO Auto-generated method stub
    return true;
  }

  @Override
  public boolean isEnabled() {
    // TODO Auto-generated method stub
    return true;
  }

}
```

Step 6: Define the Services and the implementation which we want to expose as a Rest API.

CustomerService.java:

```
package com.masai.service;

import java.util.List;
import com.masai.exception.CustomerException;
import com.masai.model.Customer;

public interface CustomerService {

  public Customer registerCustomer(Customer customer);

  public Customer getCustomerDetailsByEmail(String email)throws CustomerException;

  public List<Customer> getAllCustomerDetails()throws CustomerException;

}
```

CustomerServiceImpl.java

```
package com.masai.service;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.masai.exception.CustomerException;
import com.masai.model.Customer;
import com.masai.repository.CustomerRepository;
```

```
@Service
public class CustomerServiceImpl implements CustomerService{

  @Autowired
  private CustomerRepository customerRepository;

  @Override
  public Customer registerCustomer(Customer customer) throws CustomerException {

    return customerRepository.save(customer);


  }

  @Override
  public Customer getCustomerDetailsByEmail(String email)throws CustomerException {

    return customerRepository.findByEmail(email).orElseThrow(() -> new CustomerException("Customer Not found with Email: "+email));
  }

  @Override
  public List<Customer> getAllCustomerDetails()throws CustomerException {

    List<Customer> customers= customerRepository.findAll();

    if(customers.isEmpty())
      throw new CustomerException("No Customer find");

    return customers;

  }

}
```

Step 7: Define the CustomerException handlers

CustomerException.java

```
package com.masai.exception;

public class CustomerException extends RuntimeException {

  public CustomerException() {
    // TODO Auto-generated constructor stub
  }

  public CustomerException(String message) {
    super(message);
  }
}
```

MyErrorDetails.java:

```
package com.masai.exception;

import java.time.LocalDateTime;

public class MyErrorDetails {

  private LocalDateTime timestamp;
  private String message;
  private String details;

  //getters and setters
}
```

GlobalExceptionHandler.java

```
package com.masai.exception;

import java.time.LocalDateTime;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.context.request.WebRequest;

@ControllerAdvice
public class GlobalExceptionHandler {

  @ExceptionHandler(CustomerException.class)
  public ResponseEntity<MyErrorDetails> customerExceptionHandler(CustomerException ce, WebRequest req){

    MyErrorDetails err= new MyErrorDetails();
      err.setTimestamp(LocalDateTime.now());
      err.setMessage(ce.getMessage());
      err.setDetails(req.getDescription(false));

    return new ResponseEntity<MyErrorDetails>(err, HttpStatus.BAD_REQUEST);
  }

  @ExceptionHandler(Exception.class)
  public ResponseEntity<MyErrorDetails> otherExceptionHandler(Exception se, WebRequest req){

    MyErrorDetails err= new MyErrorDetails();
      err.setTimestamp(LocalDateTime.now());
      err.setMessage(se.getMessage());
      err.setDetails(req.getDescription(false));

    return new ResponseEntity<MyErrorDetails>(err, HttpStatus.INTERNAL_SERVER_ERROR);
  }
}
```

Step 8: Expose some REST API to the user

```
GET  /hello : to test the api should be called only logged-in user (Protected)
POST /customers: to register any customer(any one should call this API) (non-protected)
GET /customers/email: to get any customer details based on their email (protected)
GET /customers: to get all the customer details (protected)
GET /signIn: to login a customer by passing their credentials and getting the authentication message.
```

CustomerController.java:

```
package com.masai.controller;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.security.core.Authentication;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

import com.masai.model.Customer;
import com.masai.service.CustomerService;

@RestController
```

```
public class CustomerController {

  @Autowired
  private CustomerService customerService;

  @Autowired
  private PasswordEncoder passwordEncoder;


  @GetMapping("/hello")
  public String testHandler() {
    return "Welcome to Spring Security";
  }

  @PostMapping("/customers")
  public ResponseEntity<Customer> saveCustomerHandler(@RequestBody Customer customer){

    customer.setPassword(passwordEncoder.encode(customer.getPassword()));

    Customer registeredCustomer= customerService.registerCustomer(customer);

    return new ResponseEntity<>(registeredCustomer,HttpStatus.ACCEPTED);

  }

  @GetMapping("/customers/{email}")
  public ResponseEntity<Customer> getCustomerByEmailHandler(@PathVariable("email") String email){


    Customer customer= customerService.getCustomerDetailsByEmail(email);

    return new ResponseEntity<>(customer,HttpStatus.ACCEPTED);

  }

  @GetMapping("/customers")
  public ResponseEntity<List<Customer>> getAllCustomerHandler(){


    List<Customer> customers= customerService.getAllCustomerDetails();

    return new ResponseEntity<>(customers,HttpStatus.ACCEPTED);

  }

  @GetMapping("/signIn")
  public ResponseEntity<String> getLoggedInCustomerDetailsHandler(Authentication auth){

    System.out.println(auth); // this Authentication object having Principle object details

     Customer customer= customerService.getCustomerDetailsByEmail(auth.getName());

     return new ResponseEntity<>(customer.getName()+"Logged In Successfully", HttpStatus.ACCEPTED);
  }
}
```

Step 9: Define the Security configuration class based on the above requirement:

AppConfig.java

```
package com.masai.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpMethod;
import org.springframework.security.config.Customizer;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
```

```
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
public class AppConfig {

  @Bean
  public SecurityFilterChain springSecurityConfiguration(HttpSecurity http) throws Exception {


    http.authorizeHttpRequests(auth ->{

      auth.requestMatchers(HttpMethod.POST, "/customers").permitAll()
      .anyRequest().authenticated();
    })
    .csrf(csrf -> csrf.disable())
    .formLogin(Customizer.withDefaults())
    .httpBasic(Customizer.withDefaults());


    return http.build();

  }

  @Bean
  public PasswordEncoder passwordEncoder() {

    return new BCryptPasswordEncoder();

  }

}
```

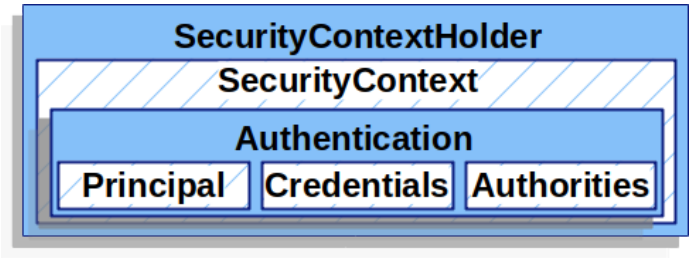
In case of Swagger we need to whitelist the Swagger-UI related end points:

```
.requestMatchers("/swagger-ui*/**","/v3/api-docs/**").permitAll()
```

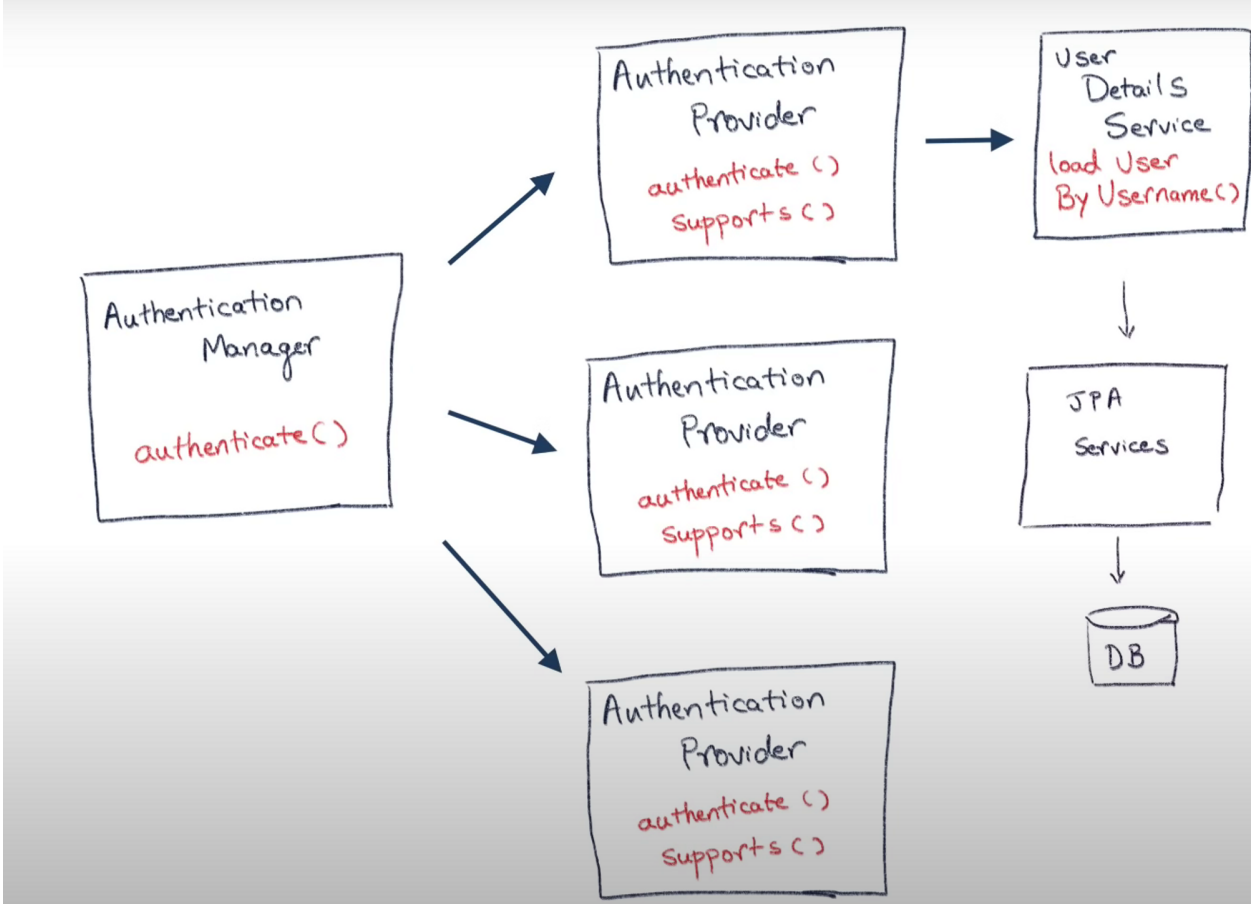
SecurityContextHolder

The most fundamental object is SecurityContextHolder. This is where we store details of the present security context of the application, which includes details of the principal, currently using the application.

The `SecurityContextHolder` is where Spring Security stores the details of who is authenticated. The `SecurityContext` is obtained from the SecurityContextHolder. The `SecurityContext` contains an Authentication object.



Internal Flow



DaoAuthenticationProvider is an AuthenticationProvider implementation that uses a UserDetailsService and PasswordEncoder to authenticate a username and password.

The authentication Filter from the Reading the Username & Password section passes a UsernamePasswordAuthenticationToken to the AuthenticationManager, which is implemented by ProviderManager.

The ProviderManager is configured to use an AuthenticationProvider of type DaoAuthenticationProvider. DaoAuthenticationProvider looks up the UserDetails from the UserDetailsService.

DaoAuthenticationProvider uses the PasswordEncoder-to validate the password on the UserDetails returned in the previous step.

When authentication is successful, the Authentication that is returned is of type UsernamePasswordAuthenticationToken and has a principal that is the UserDetails returned by the configured UserDetailsService. Ultimately, the returned UsernamePasswordAuthenticationToken is set on the SecurityContextHolder by the authentication Filter.

Implementing the AuthenticationProvider:

- Till now we are using default AuthenticationProvider given by the Spring Security, which is DAOAuthenticationProvider.
- Whenever we are using the default AuthhenticationProvider, as a developer our responsibility is to only load the UserDetails from the storage system like database and configure the Password encoders. and we forward these changes to the DAOAuthhenticationProvider, then the DAOAuthhenticationProvider will take care of Authenticating the user by comparing the pass, and along with the pass comparison it will also check other parameters like whether is account is locked or disabled or expired, whether the password is expired.
- So this DAOAuthhenticationProvider provides most of the scenarios that we want to consider during the authentication of a end-user. this default logics inside the DAOAuthhenticationProvider is more than enough for most of the project.
- But in realworld we may have some complex requirement whenever we want to authenticate an enduser. in those kind of scenarios, apart from the loading the userdetails from the storeage system, we also want to execute our own authentication logic,
- May be our client have some unique requirement like only allow the users into the system whose age is more than 18 years. or only allows the user to login from the list of allowed countries. etc.
- So whenever we want to perform some of our own custom authentication logic then we can write our own AuthhenticationProvider.
- There can be multiple AP inside our application.
  scenarios to have the multiple APs:
  - client may allow the users to get authentication by various approaches.
    like using:
    - Username and password.
      Outh2 authentication (it is an advance authentication and authorization f/w)
      OTP authentication
- It is the responsibility of the PoviderManager which is the implementation of AuthenticationManager, to check with all the implementations of AuthhenticationProviders and try to authenticate the user.

```
public interface AuthentcationProvider{

Authentication authenticate(Authentication authentication) throws AuthenticationException;

boolean support(Class<?> authentication);

}

-
```

We need to override both methods to implement our own custom AuthenticationProvider.

MyAuthenticationProvider.java

```
@Component
public class MyAuthenticationProvider implements AuthenticationProvider{

  @Autowired
  private CustomerRepository customerRepository;

  @Autowired
  private PasswordEncoder pEncoder;



  @Override
  public Authentication authenticate(Authentication authentication) throws AuthenticationException {
```

```
    System.out.println("Out Authentication Provider is used...");

    String username = authentication.getName();
    String pwd = authentication.getCredentials().toString();

    Optional<Customer> opt = customerRepository.findByEmail(username);

    if (opt.isEmpty())
      throw new BadCredentialsException("No User registerd with this details");
    else {

        Customer customer= opt.get();

      if (pEncoder.matches(pwd, customer.getPassword())) {

    List<GrantedAuthority> authorities = new ArrayList<>();
    //authorities.add(new SimpleGrantedAuthority(customer.getRole()));

    return new UsernamePasswordAuthenticationToken(username, pwd, authorities);

      } else
        throw new BadCredentialsException("Invalid Password");


    }


  }

  @Override
  public boolean supports(Class<?> authentication) {
    return UsernamePasswordAuthenticationToken.class.isAssignableFrom(authentication);



  }
}
```

Note: If we compare the logic with MyAuthenticationProvider .java class and CustomerUserDetails.java class, inside the CustomerUserDetails.java class, we just loaded the UserDetails from the database and converted to the UserDetails object. by the using the constructors of User class.

This UserDetails object will be used by the DAOAuthenticationProvider, and inside our DAOAuthenticationProvider, it will compare the password and if everything is successful it is going to create an object of Authentication, similar kind of things we have done inside our MyAuthenticationProvider.java class.

To test this application we can delete the CustomerUserDetails.java class, because we are not using the DAOAuthenticationProvider.

Authorization:

Authentication vs Authorization:

Authentication	Authorization
here identity of the users are checked for providing the access to the system.	here person's or user's authorities are checked for accessing the resources.
AuthN is done before AuthZ.	AuthZ always happens after AuthN
It needs usually user's login details.	It needs user's privilege or roles.
If fails we get 401 error response	If fails we get 403 error response
Who are you ?	What you want ?
If a Bank Customer/Employee in order to perform actions in application, we need to prove identity.	Once logged in the application based on our roles, authorities will decide what kind of action i can do.

How Authorities stored inside the Spring Security ?

Authorities/Roles information in Spring Security is stored inside the **GrantedAuthority**. It is an interface and there is only one method inside the GrantedAuthority interface which returns the name of the authority or role.

```
public interface GrantedAuthority{
  public String getAuthority();
}
```

SimpleGrantedAuthority class is the default implementation of the GrantedAuthority interface.

There is a difference between Authorities and roles, but both is stored inside the Spring Security framework is same.

This Authorities information is stored inside the object of **UserDetails** and the **Authentication** interface which plays a vital role during authentication of the user.

Example:

```
Authentication(I)
          |
UsernamePasswordAuthenticationToken
          |
```

```
public Collection<GrantedAuthority> getAuthorities();


and

   UserDetails(I)
         |
        User
         |
public Collection<GrantedAuthority> getAuthorities();
```

In order to fetch the authorities for a perticular user, that we have loaded from the database, under the **User** class there is a method call **getAuthorities()**, which will be invoked by our Spring Security framework.

In case of our custom AuthenticationProvider implementation, when we create object of the UsernamePasswordAuthenticationToken, here also Spring Security framework will invoke the **getAuthorities()** method.

Authority vs Roles:

Authority:

If the user is trying to do something in the application then application will
allow or authorized them to do it only if the user has been granted authority to do so.

Authorities are the fine grain permission of what the user can do.

It is like an individual privileges or an action.

Example: VIEWCUSTOMER, UPDATECUSTOMER, etc.

Role:(Group of Authority)

Assigning same set of authorities to the multiple clerks again and again will become tedious.

Roles are very much like a group of authority that are usually assign together.

Roles are more Coarse-grained permission unlike fine-grain permission authority.

Example: ROLE_ADMIN, ROLE_USER, etc.

Note: Roles are also represented using the same contract GrantedAutority in Spring Security.
when defining a role, its name should starts with **ROLE**_ prefix .it is a standards follows inside the Spring Security, this prefix specifies the difference between the role and an authority.

So instead of creating so many authorities inside our app, we can assume our **USER** can perform some actions and our **ADMIN** can perform some activities. and accordingly we can create certain roles inside our application.

Example of using Authority:

Spring Security given us the flexibilities to define any number of authorities or any number of roles to a single user/customer.

For that we can create a new table called **authorities** and define the set of authorities that any user can have and from this table we can have a foreign key link with the Customer table.

So when we try to load the customer details from the database we will also get the List of all the authorities for that customer.

One to Many relation with Customer and authorities.

Customer.java:

```
package com.masai.model;

import java.util.ArrayList;
```

```
import java.util.List;

import com.fasterxml.jackson.annotation.JsonProperty;

import jakarta.persistence.CascadeType;
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.OneToMany;

@Entity
public class Customer {

  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Integer custId;
  private String name;

  @Column(unique = true)
  private String email;
  @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
  private String password;
  private String address;


  @OneToMany(cascade = CascadeType.ALL, mappedBy = "customer",fetch=FetchType.EAGER)
  private List<Authority> authorities = new ArrayList<>();

  //getters and setters

}
```

Authority.java:

```
package com.masai.model;

import com.fasterxml.jackson.annotation.JsonIgnore;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.ManyToOne;

@Entity
public class Authority {

  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Integer authId;
  private String name;

  @JsonIgnore
  @ManyToOne
  private Customer customer;

  //getters and setters
}
```

CustomerRepository.java:

```
package com.masai.repository;

import java.util.Optional;

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

```
import com.masai.model.Customer;

public interface CustomerRepository extends JpaRepository<Customer, Integer>{


  public Optional<Customer> findByEmail(String email);
}
```

CustomerService.java:

```
package com.masai.service;
import java.util.List;

import com.masai.exception.CustomerException;
import com.masai.model.Customer;

public interface CustomerService {

  public Customer registerCustomer(Customer customer);

  public Customer getCustomerDetailsByEmail(String email)throws CustomerException;

  public List<Customer> getAllCustomerDetails()throws CustomerException;

}
```

CustomerServiceImpl.java:

```
package com.masai.service;

import java.util.List;

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

import com.masai.exception.CustomerException;
import com.masai.model.Authority;
import com.masai.model.Customer;
import com.masai.repository.CustomerRepository;

@Service
public class CustomerServiceImpl implements CustomerService{

  @Autowired
  private CustomerRepository customerRepository;

  @Override
  public Customer registerCustomer(Customer customer) throws CustomerException {

    List<Authority> authorities= customer.getAuthorities();

    //associating each authority with customer
    for(Authority authority:authorities) {
      authority.setCustomer(customer);
    }

    return customerRepository.save(customer);


  }

  @Override
  public Customer getCustomerDetailsByEmail(String email)throws CustomerException {
```

```
    return customerRepository.findByEmail(email).orElseThrow(() -> new CustomerException("Customer Not found with Email: "+email));
  }

  @Override
  public List<Customer> getAllCustomerDetails()throws CustomerException {

    List<Customer> customers= customerRepository.findAll();

    if(customers.isEmpty())
      throw new CustomerException("No Customer find");

    return customers;

  }

}
```

CustomerUserDetailsService.java:

```
package com.masai.service;

import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.authentication.BadCredentialsException;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
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.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

import com.masai.model.Authority;
import com.masai.model.Customer;
import com.masai.repository.CustomerRepository;

@Service
public class CustomerUserDetailsService implements UserDetailsService {

  @Autowired
  private CustomerRepository customerRepository;

  @Override
  public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

    Optional<Customer> opt = customerRepository.findByEmail(username);

    if (opt.isPresent()) {

      Customer customer = opt.get();

      List<GrantedAuthority> authorities = new ArrayList<>();

      List<Authority> auths = customer.getAuthorities();

      for (Authority auth : auths) {
        SimpleGrantedAuthority sga = new SimpleGrantedAuthority(auth.getName());
        authorities.add(sga);
      }

      System.out.println("granted authorities " + authorities);

      return new User(customer.getEmail(), customer.getPassword(), authorities);

//return new User(customer.getEmail(), customer.getPassword(), getGrantedAuthorities(customer.getAuthorities()));

    } else
      throw new BadCredentialsException("User Details not found with this username: " + username);
```

```
  }

//   private List<GrantedAuthority> getGrantedAuthorities(List<Authority> authorities) {
//          List<GrantedAuthority> grantedAuthorities = new ArrayList<>();
//          for (Authority authority : authorities) {
//              grantedAuthorities.add(new SimpleGrantedAuthority(authority.getName()));
//          }
//          return grantedAuthorities;
//      }

}
```

Configuring Authorities inside the application:

hasAuthority() method:

- It accepts a single authority for which the endpoint will be configured and user will be validated against the single authorities mentioned. only users having the same authority configured can invoke the endpoint.

hasAnyAuthority() method:

- It accepts multiple authorities for which the endpoint will be configured and user will be validated against the authorities mentioned. Only users having any of the authority configured can invoke the endpoint.

Example:

AppConfig.java:

```
package com.masai.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpMethod;
import org.springframework.security.config.Customizer;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
public class AppConfig {


  @Bean
  public SecurityFilterChain springSecurityConfiguration(HttpSecurity http) throws Exception {

    http.authorizeHttpRequests(auth ->{

      auth
      .requestMatchers(HttpMethod.POST,"/customers").permitAll()
      .requestMatchers(HttpMethod.GET,"/customers","/hello").hasAnyAuthority("VIEWALLCUSTOMER")
      .requestMatchers(HttpMethod.GET,"/customers/**").hasAnyAuthority("VIEWALLCUSTOMER","VIEWCUSTOMER")
      .anyRequest().authenticated();

    })
    .csrf(csrf -> csrf.disable())
    .formLogin(Customizer.withDefaults())
    .httpBasic(Customizer.withDefaults());


    return http.build();

  }

  @Bean
  public PasswordEncoder passwordEncoder() {

    return new BCryptPasswordEncoder();

  }
```

```
}
```

CustomerController.java:

```
package com.masai.controller;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.security.core.Authentication;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

import com.masai.model.Customer;
import com.masai.service.CustomerService;

@RestController
public class CustomerController {

  @Autowired
  private CustomerService customerService;

  @Autowired
  private PasswordEncoder passwordEncoder;


  @GetMapping("/hello")
  public String testHandler() {
    return "Welcome to Spring Security";
  }

  /*

    {
        "name": "ram",
        "email": "ram@gmail.com",
        "password": "1234",
        "address": "delhi",
        "authorities":[
            {
                "name": "VIEWALLCUSTOMER"
            },
            {
                "name": "VIEWCUSTOMER"
            }
        ]
    }




   */

  // add another Customer with only one authority "VIEWCUSTOMER"


  @PostMapping("/customers")
  public ResponseEntity<Customer> saveCustomerHandler(@RequestBody Customer customer){

    customer.setPassword(passwordEncoder.encode(customer.getPassword()));

    Customer registeredCustomer= customerService.registerCustomer(customer);

    return new ResponseEntity<>(registeredCustomer,HttpStatus.ACCEPTED);
```

```
  }

  @GetMapping("/customers/{email}")
  public ResponseEntity<Customer> getCustomerByEmailHandler(@PathVariable("email") String email){


    Customer customer= customerService.getCustomerDetailsByEmail(email);

    return new ResponseEntity<>(customer,HttpStatus.ACCEPTED);

  }

  @GetMapping("/customers")
  public ResponseEntity<List<Customer>> getAllCustomerHandler(){


    List<Customer> customers= customerService.getAllCustomerDetails();

    return new ResponseEntity<>(customers,HttpStatus.ACCEPTED);

  }

  @GetMapping("/signIn")
  public ResponseEntity<String> getLoggedInCustomerDetailsHandler(Authentication auth){

    System.out.println(auth); // this Authentication object having Principle object details

     Customer customer= customerService.getCustomerDetailsByEmail(auth.getName());

     return new ResponseEntity<>(customer.getName()+"Logged In Successfully", HttpStatus.ACCEPTED);
  }
}
```

Exception handler related classes as same as previous application

Example: ROLE based authorization

In Spring Security the ROLES requirements can be configured using the following ways:

- hasRole()
- hasAnyRole()

Note: ROLE_ prefix only to be used while adding the role inside the database, i.e. while registering the customer with the Role, we need to prefix those roles with the prefix, so that it will stored inside the database with the prefix ROLE_. But when we configure the roles inside our Spring Security application, we do it only by its name.

Customer.java:

```
package com.masai.model;

import com.fasterxml.jackson.annotation.JsonProperty;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

@Entity
public class Customer {

  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Integer custId;
  private String name;
```

```
  @Column(unique = true)
  private String email;
  @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
  private String password;
  private String address;

  private String role;

  //getters and setters

}
```

CustomerService.java, CustomerRepository.java and Exception handler classes are from previous example.

CustomerServiceImpl.java

```
package com.masai.service;

import java.util.List;

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

import com.masai.exception.CustomerException;
import com.masai.model.Customer;
import com.masai.repository.CustomerRepository;

@Service
public class CustomerServiceImpl implements CustomerService{

  @Autowired
  private CustomerRepository customerRepository;

  @Override
  public Customer registerCustomer(Customer customer) throws CustomerException {

    return customerRepository.save(customer);


  }

  @Override
  public Customer getCustomerDetailsByEmail(String email)throws CustomerException {

    return customerRepository.findByEmail(email).orElseThrow(() -> new CustomerException("Customer Not found with Email: "+email));
  }

  @Override
  public List<Customer> getAllCustomerDetails()throws CustomerException {

    List<Customer> customers= customerRepository.findAll();

    if(customers.isEmpty())
      throw new CustomerException("No Customer find");

    return customers;

  }

}
```

CustomerUserDetailsService.java

```
package com.masai.service;

import java.util.ArrayList;
```

```
import java.util.List;
import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.authentication.BadCredentialsException;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
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.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

import com.masai.model.Customer;
import com.masai.repository.CustomerRepository;

@Service
public class CustomerUserDetailsService implements UserDetailsService{

  @Autowired
  private CustomerRepository customerRepository;


  @Override
  public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {


    Optional<Customer> opt= customerRepository.findByEmail(username);

    if(opt.isPresent()) {

      Customer customer= opt.get();

      List<GrantedAuthority> authorities= new ArrayList<>();

      SimpleGrantedAuthority sga= new SimpleGrantedAuthority(customer.getRole());
      authorities.add(sga);

      return new User(customer.getEmail(), customer.getPassword(), authorities);

    }else
      throw new BadCredentialsException("User Details not found with this username: "+username);
  }

}
```

AppConfig.java

```
package com.masai.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpMethod;
import org.springframework.security.config.Customizer;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
public class AppConfig {


  @Bean
  public SecurityFilterChain springSecurityConfiguration(HttpSecurity http) throws Exception {

    http.authorizeHttpRequests(auth ->{

      auth
      .requestMatchers(HttpMethod.POST,"/customers").permitAll()
    .requestMatchers(HttpMethod.GET, "/customers","/hello").hasRole("ADMIN")
```

```
    .requestMatchers(HttpMethod.GET, "/customers/**").hasAnyRole("ADMIN","USER")
      .anyRequest().authenticated();

    })
    .csrf(csrf -> csrf.disable())
    .formLogin(Customizer.withDefaults())
    .httpBasic(Customizer.withDefaults());


    return http.build();

  }

  @Bean
  public PasswordEncoder passwordEncoder() {

    return new BCryptPasswordEncoder();

  }


}
```

CustomerController.java:

```
package com.masai.controller;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.security.core.Authentication;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

import com.masai.model.Customer;
import com.masai.service.CustomerService;

@RestController
public class CustomerController {

  @Autowired
  private CustomerService customerService;

  @Autowired
  private PasswordEncoder passwordEncoder;


  @GetMapping("/hello")
  public String testHandler() {
    return "Welcome to Spring Security";
  }

  /*

    {
        "name": "ram",
        "email": "ram@gmail.com",
        "password": "1234",
        "address": "delhi",
        "role": "ROLE_ADMIN"
    }
```

```
   */

  // add another Customer with only the role "ROLE_USER"


  @PostMapping("/customers")
  public ResponseEntity<Customer> saveCustomerHandler(@RequestBody Customer customer){

    customer.setPassword(passwordEncoder.encode(customer.getPassword()));

    Customer registeredCustomer= customerService.registerCustomer(customer);

    return new ResponseEntity<>(registeredCustomer,HttpStatus.ACCEPTED);

  }

  @GetMapping("/customers/{email}")
  public ResponseEntity<Customer> getCustomerByEmailHandler(@PathVariable("email") String email){


    Customer customer= customerService.getCustomerDetailsByEmail(email);

    return new ResponseEntity<>(customer,HttpStatus.ACCEPTED);

  }

  @GetMapping("/customers")
  public ResponseEntity<List<Customer>> getAllCustomerHandler(){


    List<Customer> customers= customerService.getAllCustomerDetails();

    return new ResponseEntity<>(customers,HttpStatus.ACCEPTED);

  }

  @GetMapping("/signIn")
  public ResponseEntity<String> getLoggedInCustomerDetailsHandler(Authentication auth){

    System.out.println(auth); // this Authentication object having Principle object details

     Customer customer= customerService.getCustomerDetailsByEmail(auth.getName());

     return new ResponseEntity<>(customer.getName()+"Logged In Successfully", HttpStatus.ACCEPTED);
  }
}
```