**Programmatic Approach:**

**Use case: Banking Application**

Roles

|-Members (Banking users)

|-Employees (Staff)

|-Moderators (Monitor who will monitors consolidated reports)

|-Admin

|-Manager

Money transfer from one account to another account, then we need to authenticate but here we need to check this user is this user to transfer by granting the permissions.

For example if we wanted to transfer the money from one account to another then 1st our account should be activated under the specified role that means when our account is activated then only money can transferred otherwise cannot. If we look into the SBI account if we want to transfer money 1st we need to login as User role by using Config approach but later once we login if we wanted transfer money we need to login with profile password under the specified role then only we will allowed to transfer the money.

But we login successfully but money transfer option is not enabled so we need to check this user is account is activated for transfer money or not based on the authentication and Authorization role that means at the runtime programmatically we need to apply security even though we applied config based security, so in an application both "Configuration based Security" and "Programmatic Security" are needed.

That means for conditional based check security at run time like Access Controlling Security we need to programmatic security.

Configuration based Security to provide the application access like gate way of the application, and Programmatic security for runtime purpose like to check whether user is authorized for money transfer or not for this we need programmatic Security.

🡪Do you applied programmatic security in your application?

Yes, we have both config and Programmatic based security both and it is must for bigger applications.

🡪How many ways we can implement security in an application?

Ans: 2-Ways using Config and Programatic (Either by using API classes or Annotaions)

🡪When do we need to go for Programatic Approach?

For conditional based check security at run time like Access Controlling Security we need to programmatic security.

The security features that we discussed till are related declarative security metadata, or metadata that is statically defined before an application even runs. JAX-RS also has a small programmatic API for gathering security information about a secured request. Specifically, the javax.ws.rs.core.SecurityContext interface has a method for determining the identity of the user making the secured HTTP invocation. It also has a method that allows us to check whether or not the current user belongs to a certain role:

public interface SecurityContext {

public Principal getUserPrincipal();

public boolean isUserInRole(String role);

public boolean isSecure();

public String getAuthenticationScheme();

}

The getUserPrincipal() method returns a standard Java Standard Edition (SE) javax.security.Principal security interface. A Principal object represents the individual user who is currently invoking the HTTP request.

The isUserInRole() method allows us to determine whether the current calling user belongs to a certain role.

The isSecure() method returns true if the current request is a secure connection.

The getAuthenticationScheme() tells us which authentication mechanism was used to secure the request.

BASIC, DIGEST, CLIENT\_CERT, and FORM are typical values returned by this method. We get access to a SecurityContext instance by injecting it into a field, setter method, or resource method parameter using the @Context annotation.

Let’s examine this security interface with an example. Let’s say we want to have a security log of all access to a customer database by users who are not administrators. Here is how it might look:

@Path("/customers")

public class CustomerService {

@GET

@Produces("application/xml")

public Customer[] getCustomers(@Context SecurityContext sec) {

if (sec.isSecure() && !sec.isUserInRole("ADMIN")) {

logger.log(sec.getUserPrincipal() + " accessed customer database.");

}

...

}

}

In this example, we inject the SecurityContext as a parameter to our getCustomer() JAX-RS resource method. We use the method SecurityContext.isSecure() to determine whether or not this is an authenticated request. We then use the method SecurityContext.isUserInRole() to find out if the caller is an ADMIN or not. Finally, we print out to our audit log.

With the introduction of the filter API in JAX-RS 2.0, we can implement the SecurityContext interface and override the current request’s SecurityContext via the ContainerRequestContext.setSecurityContext () method. What’s interesting about this is that we can implement your own custom security protocols.

Here’s an example:

@PreMatching

class CustomAuth implements ContainerRequestFilter {

protected MyCustomerProtocolHandler customProtocol = ...;

public void filter(ContainerRequestContext requestContext) {

String authHeader = request.getHeaderString(HttpHeaders.AUTHORIZATION);

SecurityContext newSecurityContext = customProtocol.validate(authHeader);

requestContext.setSecurityContext(authHeader);

}

}

This filter leaves out a ton of detail, but hopefully we get the idea. It extracts the Authorization header from the request and passes it to the customProtocol service that we have written. This returns an implementation of SecurityContext. We can override the default SecurityContext with this variable.

Example:

Access the application:

Case: 1 Failure case:

http://localhost:8082/2.2ProgramaticSecurityRESTEasy/rest/trading/buy

Select POST

Content-Type: application/xml

Req Body:

<buyOrder>

<customerID>2</customerID>

<stockName>cypla</stockName>

<quatity>24</quatity>

<exchange>bse</exchange>

</buyOrder>

Login as

Username= robin

password= welcome

Response:

Status Code: 401: UnAutherized

Case: 2 Success case:

http://localhost:8082/2.2ProgramaticSecurityRESTEasy/rest/trading/buy

Select POST

Content-Type: application/xml

Req Body:

<buyOrder>

<customerID>2</customerID>

<stockName>cypla</stockName>

<quatity>24</quatity>

<exchange>bse</exchange>

</buyOrder>

Login as

Username= john

password= welcome

Response:

Status Code: 200 (OK)

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<invoice><invoiceNo>0398b752-beb6-4297-b1fc-47fbdccae109</invoiceNo>

<status>partial</status>

</invoice>