CST8277 (21S) Assignment 4: REST BloodBank

Java EE Group Project

Please read this document carefully, all sections (perhaps multiple times to make sure you find all the places that say you '***must***'). If your submission does not meet the requirements as stated here, you may lose marks even though your program runs. Additionally, this assignment is also a teaching opportunity – **material presented here will be on the Final Exam!**

Model Entities – some familiar, some new

For Assignment 4, you need to re-use the BloodBank entities from Assignment 3. Additionally, a new entity SecurityUser will be mapped to the **SECURITY_USER** table and assigned one of two JEE Security Roles: ***USER_ROLE*** or ***ADMIN_ROLE***. The security will be backed by additional database tables: **SECURITY_ROLE** table and a **USER_HAS_ROLE** join table. This means an additional entity SecurityRole needs to be mapped as well.

Theme for the Group Project

The theme for the Group Project is to bring together everything you have learned this term:

1. JPA – for model objects
2. Session beans – for business logic
3. REST – representation of back-end resources
4. JEE Security Roles – controls who can invoke which operation
5. Testing using JUnit – a series of test cases that demonstrate the operation of the system

Submission

Assignment 4's submission is to be uploaded to Brightspace/Activities/Assignments. Optionally, if you finish early you can demo as well.

The submission ***must*** include:

- Zip your project and submit it.
- Do not reduce anything but feel free to add.
- **Style**: Every class file has a multiline comment block at the top giving the name of the file, your names (authors), and creation date.
    - ***Important*** - The names of all group members ***must*** appear at the top of each and every source code file submitted; otherwise, you will lose marks (up to a score of 0) for the coding portion of the rubric.
- JUnit Test Suite: Test cases that demonstrate all operations.

Task One – Finish Custom Authentication Mechanism

In the starter code, you will find code similar to the JEE Security demo 'rest-demo-security':

```
@ApplicationScoped
public class CustomAuthenticationMechanism implements
HttpAuthenticationMechanism {

    @Inject
    protected IdentityStore identityStore;
```

```
@ApplicationScoped
@Default
public class CustomIdentityStore implements IdentityStore {

    @Inject
    protected CustomIdentityStoreJPAHelper jpaHelper;
```

```
@Singleton
public class CustomIdentityStoreJPAHelper {

     private static final Logger LOG = LogManager.getLogger();

     @PersistenceContext(name = PU_NAME)
     protected EntityManager em;

     public SecurityUser findUserByName(String username) {
          LOG.debug("find a User By the Name={}", username);
          SecurityUser user = null;
          //TODO:  ...
```

The **TODO** here is to make the custom authentication mechanism actually use the database must be done in the **CustomIdentityStoreJPAHelper** class.

Task Two – Relationship Between SecurityUser and Person

One of the tasks to be done is to map a 1:1 relationship between a SecurityUser and a Person. Please see the TODO inside SecurityUser class. This is done so that when the custom authentication mechanism successfully resolves the Principal (SecurityUser implements the Principal interface), it can be inject'd into your code – then the developer can un-wrap the object and find the SecurityUser inside and from there access the related Person as was done in getPersonById() in PersonResource:

```
@Inject
protected SecurityContext sc;

@GET
@RolesAllowed({ADMIN_ROLE, USER_ROLE})
@Path(RESOURCE_PATH_ID_PATH)
public Response getPersonById(@PathParam(RESOURCE_PATH_ID_ELEMENT) int
id) {
	LOG.debug("try to retrieve specific person " + id);
	Response response = null;
	Person person = null;

	if (sc.isCallerInRole(ADMIN_ROLE)) {
		person = service.getPersonId(id);
		response = Response.status(person == null ? Status.NOT_FOUND
: Status.OK).entity(person).build();
	} else if (sc.isCallerInRole(USER_ROLE)) {
		WrappingCallerPrincipal wCallerPrincipal =
(WrappingCallerPrincipal) sc.getCallerPrincipal();
		SecurityUser sUser = (SecurityUser)
wCallerPrincipal.getWrapped();
		person = sUser.getPerson();
		if (person != null && person.getId() == id) {
			response =
Response.status(Status.OK).entity(person).build();
		} else {
			throw new ForbiddenException("User trying to access
resource it does not own (wrong userid)");
		}
	} else {
		response = Response.status(Status.BAD_REQUEST).build();
	}
	return response;
}
```

There is no requirement to create an (administrative) API for creating SecurityUser's and SecurityRole's – you may populate the SECURITY_USER, SECURITY_ROLE and USER_HAS_ROLE tables using 'raw' SQL (or use MySQL Workbench or DBeaver).

Task Three and Lesson 1 – Building a REST API

We need to build JAX-RS REST'ful resources for our model objects/entities (remember the security requirements from Task Two):

```
@Path(PERSON_RESOURCE_NAME)
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class PersonResource {

	private static final Logger LOG = LogManager.getLogger();

	@EJB
	protected BloodBankService service;

	...

	@GET
	@RolesAllowed({ADMIN_ROLE, USER_ROLE})
	@Path(RESOURCE_PATH_ID_PATH)
	public Response getPersonById(@PathParam(
RESOURCE_PATH_ID_ELEMENT) int id) {
		...
	}

	...
}
```

The main focus is on C-R-U-D:

- Q1: What REST message creates a Person?
  - What endpoint API should we send the above message to?
- Q2: What REST method relates an Address to a Person?
  - What endpoint API should we send the above message to?
- .. you get the idea (!)

Swagger and Postman

Before making your JUnit, you can use Swagger or Postman to test your REST APIs.

You should do the tutorial at https://app.swaggerhub.com/help/tutorials/openapi-3-tutorial to become familiar with how to use the editor and add to the .yaml document.

Or you can use https://www.postman.com/downloads/.

Task Four and Lesson 2 – Securing REST Endpoints

You need to put JEE security annotations on your REST'ful resources to enforce the following rules:

- Only a user with the SecurityRole 'ADMIN_ROLE' can get the list of all persons.
- A user with either the role 'ADMIN_ROLE' or 'USER_ROLE' can get a specific person. However, there is logic inside the **getPersonById** method that disallows a 'USER_ROLE' user from getting a person that is not linked to the SecurityUser.
- Only a user with the SecurityRole 'ADMIN_ROLE' can add a new person.
- Any user can retrieve the list of BloodDonations and BloodBanks.
- Only an 'ADMIN_ROLE' user can apply CRUD to one or all DonationRecord.
- Only a 'USER_ROLE' user can read their own DonationRecord.
- Only an 'ADMIN_ROLE' user can associate an Address and/or Phone to a Person.
- Only an 'ADMIN_ROLE' user can delete any entities.
- **Q3:** Based on these rules, what role should be allowed to add new BloodDonation? New BloodBanks?

Task Five and Lesson 3 - Building JUnit Tests

For Java JAX-RS resources (e.g. PersonResource), there is a Client API to remotely invoke behavior on the REST'ful resources (https://javaee.github.io/tutorial/jaxrs-client.html).
[Note: In TestBloodBankSystem.java, the first test-case **test01_all_persons_with_adminrole** is implemented below. It uses JUnit 5's @BeforeAll and @BeforeEach annotations to make things more neat-&-tidy.

```
@Test
public void test01_all_persons_with_adminrole() throws JsonMappingException,
JsonProcessingException {
    Response response = webTarget
        //.register(userAuth)
        .register(adminAuth)
        .path(PERSON_RESOURCE_NAME)
        .request()
        .get();
    assertThat(response.getStatus(), is(200));
    List<Person> persons = response.readEntity(new
GenericType<List<Person>>(){});
    assertThat(persons, is(not(empty())));
    assertThat(persons, hasSize(1));
}
```

Remember, negative testing is also useful, i.e. for example:

```
assertThat(response.getMediaType(), is(not(MediaType.APPLICATION_XML));
```

You ***must*** build a collection with 30 (minimum) tests to various REST'ful URI endpoints of your BloodBank app, testing the full C-R-U-D lifecycle of the entities, building associations between entities … all using REST messages.

Fetch Strategy

Fetch should always be lazy. Because of this, you will sometimes get **LazyInitializationException**. The way to solve this is to use "fetch" in your named queries as explained below.

Normally, you will have basic named queries like these:

@NamedQuery(name = BloodBank.ALL_BLOODBANKS_QUERY_NAME, query = "SELECT distinct b FROM BloodBank b")
@NamedQuery(name = BloodBank.SPECIFIC_BLOODBANKS_QUERY_NAME, query = "SELECT distinct b FROM BloodBank b")

With join fetch, you will grab the entity from the DB and grab the dependency as well. You can have multiple join fetches.

@NamedQuery(name = BloodBank.ALL_BLOODBANKS_QUERY_NAME, query = "SELECT distinct b FROM BloodBank b **left JOIN FETCH b.donations**")
@NamedQuery(name = BloodBank.SPECIFIC_BLOODBANKS_QUERY_NAME, query = "SELECT distinct b FROM BloodBank b **left JOIN FETCH b.donations where b.id=:param1**")

Security Users

Role	User	Pass
Admin	admin	admin
User	cst8288	8288

Running the Skeleton

Unzip the skeleton project provided and then open it with your Eclipse. **Do not put your code in any shared drive like OneDrive.** This project will make many files in the background and having it synched will be major problem.

When running your code, you might be getting some errors that make no sense. Like Eclipse saying you have to import while it is already imported. You can do the following to help the situation. You might have to do one or all of these steps many times during your project:

- Go to Project → Clean.
- Update your Maven project. Right-click Project/Maven/Update.
- Clean and build your Maven project.
- Remove your project from Payara and run it again.
- Restart your Payara server.
- Manually delete the target folder in your project.

Requirement Summary

1. Make a resource for all tables.
    a. Person is given as an example.
    b. Each resource needs to support CRUD.
        i. **Update is Optional.**
    c. Resource is **not** needed for contact, security_user, security_role, and user_has_role.
2. Update your entities with appropriate Jackson annotations.
    a. Examples of all you need is in the code already.
    b. Use @JsonIgnore if you need to remove a field from being processed by Jackson. For example, blood bank does not to display all the blood donations.
    c. Use @JsonSerialize if you like to create a custom serialization of your entity. For example, when creating JSON for blood bank we just need to see the count.
    d. If you need access to your lazy fetched objects, you need to create a namedQuery with join. For example, we need all banks with their donation counts. "SELECT distinct b FROM BloodBank b **left JOIN FETCH b.donations**".
    e. What exactly needs to be displayed is up to you. Display enough meaningful information in your JSON.
3. Create JUnit tests for your REST APIs.
    a. Minimum of 30 tests.
    b. Use the Client API to test your code.
    c. Remember to run your server first as the REST API needs to be running first.
    d. Tests the roles and CRUD.

Submit your project zip file to Brightspace. You can create a **readme.txt** inside the project to put the names of all members of the group. You can also put more information inside the **readme.txt** file for me to read if necessary.

– End –