CST8277 (24S) Assignment 4: REST ACME College

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 – **materials presented here will be on the Final Exam!**

Model Entities – Some Familiar, Some New

For Assignment 4, you need to re-use the ACME College entities from Assignment 3 (**NOTE: Solution annotations for entities from Assignment 3 will be provided after the due date for Assignment 3 has passed. However, your group should still be able to start working on this group project while waiting for the solution annotations by reusing your own JPA annotations 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 semester:

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 and correctness of the system

Submission

Assignment 4's submission is to be uploaded to Brightspace/Activities/Assignments. **NOTE: Your group needs to demo the completed project to your lab professor as well.**

The submission ***must*** include:

- Zip of your project folder (**Note:** Do not reduce anything from the skeleton project but feel free to add and modify the existing annotations and/or code.)
- Updated Postman collection **REST-ACMECollege-Sample.postman_collection.json** for your project. (**Note:** Feel free to add and modify the REST requests/messages. When submitting your completed work, please include **only** those REST requests that are working properly for your REST API system and remove those REST requests that are not working or not supported by your REST API system. Your work will be graded based on the REST requests included in your Postman collection.)
- **Style**: Every class file has a multi-line comment block at the top giving the name of the file, your names (authors), and date modified.
    - ***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 Zero – Finish the JPA Annotations for Entities from Assignment 3

NOTE: Solution annotations for entities from Assignment 3 will be provided after the due date for Assignment 3 has passed. However, your group should still be able to start working on this group project while waiting for the solution annotations by reusing your own JPA annotations from Assignment 3.

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 and must be done in the **CustomIdentityStoreJPAHelper** class.

Task Two – Relationship Between SecurityUser and Student

One of the tasks to be done is to map a 1:1 relationship between a SecurityUser and a Student. 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 Student as was done in getStudentById() in StudentResource:

```
@Inject
protected SecurityContext sc;

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

	if (sc.isCallerInRole(ADMIN_ROLE)) {
		student = service.getStudentById(id);
		response = Response.status(student == null ?
Status.NOT_FOUND : Status.OK).entity(student).build();
	} else if (sc.isCallerInRole(USER_ROLE)) {
		WrappingCallerPrincipal wCallerPrincipal =
(WrappingCallerPrincipal) sc.getCallerPrincipal();
		SecurityUser sUser = (SecurityUser)
wCallerPrincipal.getWrapped();
		student = sUser.getStudent();
		if (student != null && student.getId() == id) {
			response =
Response.status(Status.OK).entity(student).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).

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(STUDENT_RESOURCE_NAME)
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class StudentResource {

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

	@EJB
	protected ACMECollegeService service;

	...

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

	...
}
```

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

- Q1: What REST request/message creates a Student?
  - What endpoint API should we send the above message to?
- Q2: What REST request/message relates a Peer Tutor to a Student?
  - What endpoint API should we send the above message to?
- .. you get the idea (!)

NOTE: There are also 2 **TODO**s inside **ACMECollegeService.java** which your group need to complete.

Swagger and Postman

Before writing 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 students.
- A user with either the role 'ADMIN_ROLE' or 'USER_ROLE' can get a specific student. However, there is logic inside the **getStudentById** method that disallows a 'USER_ROLE' user from getting a student that is not linked to the SecurityUser.
- Only a user with the SecurityRole 'ADMIN_ROLE' can add a new student.
- Any user can retrieve the list of ClubMembership and StudentClub.
- Only an 'ADMIN_ROLE' user can apply CRUD to one or all MembershipCard.
- Only a 'USER_ROLE' user can read their own MembershipCard.
- Only an 'ADMIN_ROLE' user can associate a Peer Tutor and/or Course to a Student.
- Only an 'ADMIN_ROLE' user can delete any entities.
- **Q3:** Based on these rules, what role should be allowed to add new ClubMembership? New StudentClub? etc.

Task Five and Lesson 3 - Building JUnit Tests

For Java JAX-RS resources (e.g. StudentResource), there is a Client API to remotely invoke behavior on the REST'ful resources (https://javaee.github.io/tutorial/jaxrs-client.html).
[Note: In TestACMECollegeSystem.java, the first test-case **test01_all_students_with_adminrole** is implemented below. It uses JUnit 5's @BeforeAll and @BeforeEach annotations to make things more neat and tidy.

```
@Test
public void test01_all_students_with_adminrole() throws JsonMappingException,
JsonProcessingException {
    Response response = webTarget
        //.register(userAuth)
        .register(adminAuth)
        .path(STUDENT_RESOURCE_NAME)
        .request()
        .get();
    assertThat(response.getStatus(), is(200));
    List<Student> students = response.readEntity(new
GenericType<List<Student>>(){});
    assertThat(students, is(not(empty())));
    assertThat(students, 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 40 (minimum) tests to various REST'ful URI endpoints of your ACME College app, testing the full C-R-U-D lifecycle of the entities, building associations between entities … all using REST requests/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 = StudentClub.ALL_STUDENT_CLUBS_QUERY_NAME, query = "SELECT distinct sc FROM StudentClub sc")
@NamedQuery(name = StudentClub.SPECIFIC_STUDENT_CLUB_QUERY_NAME, query = "SELECT distinct sc FROM StudentClub sc WHERE sc.id = :param1")

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

@NamedQuery(name = StudentClub.ALL_STUDENT_CLUBS_QUERY_NAME, query = "SELECT distinct sc FROM StudentClub sc **LEFT JOIN FETCH sc.clubMemberships**")
@NamedQuery(name = StudentClub.SPECIFIC_STUDENT_CLUB_QUERY_NAME, query = "SELECT distinct sc FROM StudentClub sc **LEFT JOIN FETCH sc.clubMemberships** WHERE sc.id=:param1")

Security Users

Role	User	Pass
Admin	admin	admin
User	cst8277	8277

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. Student is given as an example.
    b. Each resource needs to support CRUD operations.
        i. **Update is optional.**
    c. Resource is **not** needed for security_user, security_role, and user_has_role.
    d. Resource is needed for the following classes/entities:
        i. ClubMembership
        ii. Course
        iii. PeerTutorRegistration
        iv. MembershipCard
        v. PeerTutor
        vi. Student
        vii. StudentClub
    e. Import and use the provided Postman collection **REST-ACMECollege-Sample.postman_collection.json** to test your REST endpoints. (**Note:** Feel free to add and modify the REST requests/messages. Please include **only** those REST requests that are working properly for your REST API system and remove those REST requests that are not working or not supported by your REST API system. When submitting your completed work, please include **only** those REST requests that are working properly for your REST API system and remove those REST requests that are not working or not supported by your REST API system. Your work will be graded based on the REST requests included in your Postman collection.)
2. Update your entities with appropriate Jackson annotations similar to Lab 4.
    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, student club does not to display all the club memberships.
    c. Use @JsonSerialize if you like to create a custom serialization of your entity. For example, when creating JSON for student club 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 student clubs with their club membership counts. "SELECT distinct sc FROM StudentClub sc **left JOIN FETCH sc.clubMemberships**".
    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 40 JUnit 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 operations.
    e. **Optionally**, you may want to generate a **Maven surefire report** that summarizes your test results (see Assignment 3 on how to generate the Maven surefire report).

IMPORTANT: Please create a **ReadMe.doc** inside of your project and put the names of all members of the group. Please also specify how the work was divided among the members of the group (that is, who did what). Finally, the entire group should grade each individual member of the group based on the member's contribution to the group work. For example, the below table shows the contribution of each member and how the entire group has graded each member of the group (please see next page):

Member	Contributions	Average Peer Grade (Grade Provided by Group #1)
John Smith	Wrote 5 JUnit tests, absent from a lot of group meetings, …	60%
Mary Lee	Completed Security User & Roles annotations + wrote 24 JUnit tests, acts as group leader, …	100%
Martha Snow	Wrote Person, Phone, Blood Bank REST resources	90%
Emmanuel Gray	Wrote Donation Record, Blood Donation resources	80%

Submit a zip file containing your entire project folder to Brightspace. You should name your zip file as: <Lastname1>-<Lastname2>-<Lastname3>-<Lastname4>-Assignment4.zip. For example: **Gray-Lee-Smith-Snow-Assignment4.zip**.

– end –