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 testcases that demonstrate the operation of the system

Submission

Assignment 4's submission is to be uploaded to BS/Activities/Assignments. Optionaly 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), 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: testcases that demonstrates 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
    private IdentityStore identityStore;
```

...

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

    @Inject
    protected CustomIdentityStoreJPAHelper jpaHelper;
```

...

```
@Singleton
public class CustomIdentityStoreJPAHelper {

    @PersistenceContext(name=PU_NAME)
    protected EntityManager em;

    @Inject
    protected Pbkdf2PasswordHash pbAndjPasswordHash;

    public SecurityUser findUserByName(String username) {
        SecurityUser user = null;
        //TODO
```

The work to make the custom Authentication mechanism actually use the database (`TODO`) 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 an `Person`. 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:

```
@Inject
protected SecurityContext sc;

@RolesAllowed({USER_ROLE})
@GET
@Path("{userId}")
public Response getPersonById(@PathParam("id")
    int id) {
    Response response = null;
    WrappingCallerPrincipal wCallerPrincipal =
        (WrappingCallerPrincipal)sc.getCallerPrincipal();
    SecurityUser sUser =
        (SecurityUser)wCallerPrincipal.getWrapped();
    Person c = sUser.getPerson();
    if (c!=null && c.getId() != id) {
        throw new ForbiddenException(
         "User trying to access resource it does not own" +
          "(wrong userid)");
    }
    // ...
    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 `SECURITY_ROLE_SECURITY_USER` tables using 'raw' SQL (or use DBeaver).

Task Three and Lesson 1 – Building a REST API

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

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

    @EJB
    protected BloodBankService service;

    @GET
    @Path("{id}")
    public Response getPersonById(@PathParam("id") int id) {
        return Response.ok(
           customerServiceBean.getPersonById(id)).build();
    }
}
```

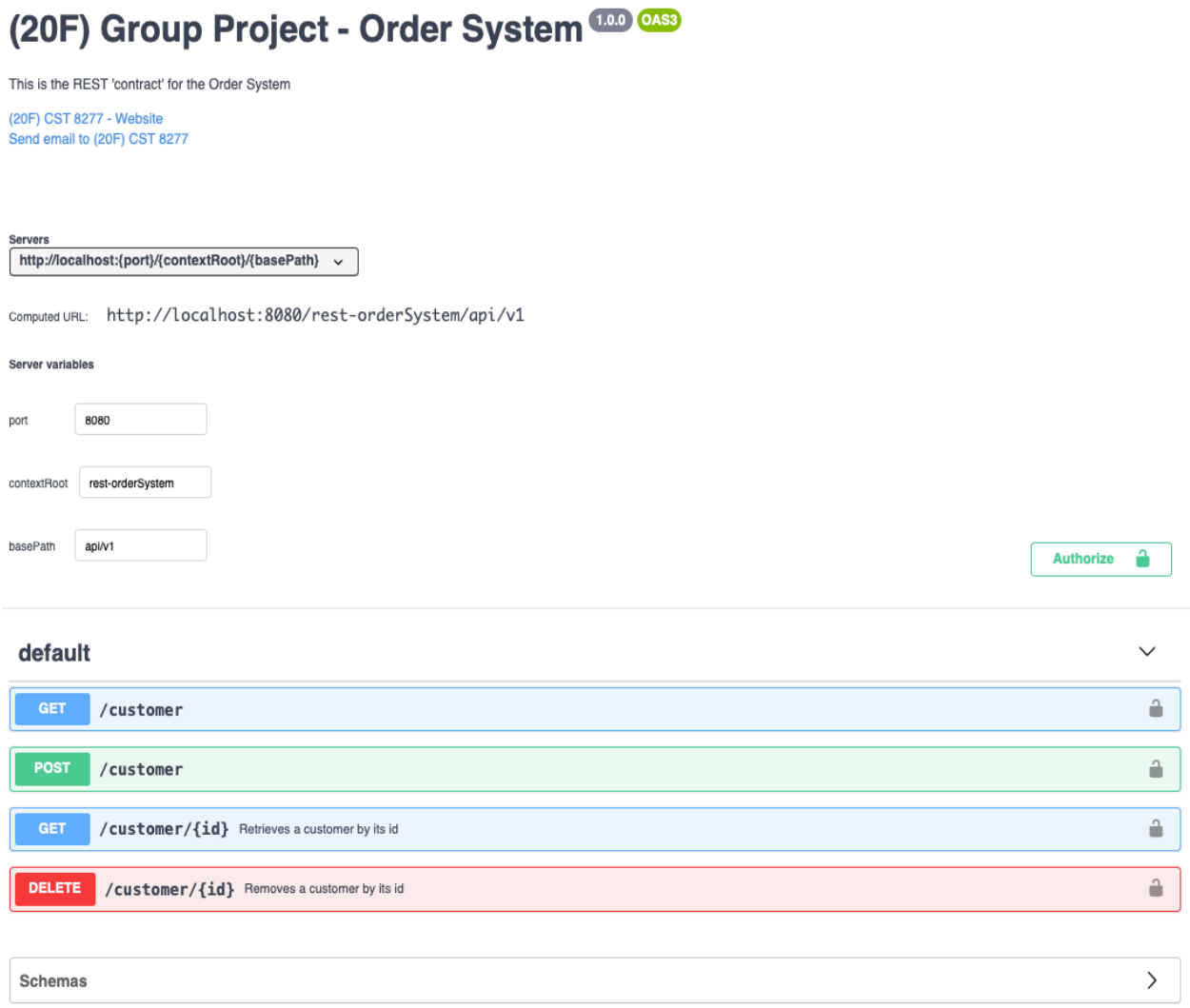
The main focus is C-R-U-D:

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

Testing your REST API (optional)

Before making your junit you can use swager or Postman.

On Brightspace in the 'Assignment 4' module is a document called `OpenAPI3_OrderSystem.yaml`. When it is imported into the online OpenAPI editor at https://editor.swagger.io, it describes a (proposed) REST API for retrieving Customer (from the above `CustomerResource`.



*just an example image.

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 3 – Securing REST Endpoints

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

- Only a user with the `SecurityRole` '`ADMIN_ROLE`' can get the list of all Customers.
- A user with either the role '`ADMIN_ROLE`' or '`USER_ROLE`' can get a specific Customer. 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`.

- Any user can retrieve the list of BloodDonations and BloodBanks.
- Only an 'ADMIN_ROLE' user can apply CRUD to one or all BloodRecords.
- Only a 'USER_ROLE' user can read their own BloodRecord.
- 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 allow to add new BloodDonation? New BloodBanks?

Task Five and Lesson 4 - Building JUnit tests

For Java JAX-RS resources (e.g. `CustomerResource`), there is a Client API to remotely invoke behaviour on the REST'ful resources (https://javaee.github.io/tutorial/jaxrs-client.html)
[Note: in `OrderSystemTestSuite`, the first test-case **`test01_all_customers_with_adminrole`** is implemented just a bit differently than below: uses JUnit 5's `@BeforeAll` and `@BeforeEach` annotations to make things more neat-&-tidy]

```
    @Test
    public void test_hello_get() {
        WebTarget webTarget;
        Client client = ClientBuilder.newClient();
        UriBuilder uriBuilder = UriBuilder.fromUri("")
           .scheme("http")
           .host("localhost")
           .port(8080)
           .path("rest-demo/api/v1");
        webTarget = client.target(uriBuilder);
        Response response = webTarget
            .path("hello")
            .request(APPLICATION_JSON)
            .get();
        assertThat(response.getStatus(), is(200));
        MessageHolder msgH = response.readEntity(MessageHolder.class);
        assertEquals("hello from the other side!", msgH.getMsg());
    }
}
```

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

```
    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.

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 your project 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 eit is already imported. You can do these 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.
- Manulay delete the target folder in your project.

Requirment Summery

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. Example 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 serilaxation of your entity. Fro example, when creating json for blood bank we just need to see the count.
    d. If you need access to your lazy fetch 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 upto you. Display enough meanigfull info in json.
3. Create junit for your REST API.
    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.