SEW IV - ToDoList

Things To Learn

- Implementing *CRUD* functionality as a *REST* service.
- Using *JPA* and *JPQL*.

Submission Guidelines

- Your implemented solution as ***zipped*** *IntelliJ*-project.

Task

Create a *REST* service for managing your daily tasks.

Quarkus

Create a new *Quarkus* project using *Maven* as *Build System*. Add the following extensions:

- *REST*
- *REST Jackson*
- *Hibernate ORM*
- *JDBC Driver - H2*

Make sure you can run the application and perform some calls to the *REST Service* before continuing with the next steps.

Model

Create a *Model*-class `Todo` as *POJO* - i.e. a *Plain Old Java Object* containing properties accessible by *getters* and *setters* and a *default constructor*. Your class should provide a numerical *ID*, a description, a deadline (as `LocalDate`) and a numerical priority. The latter one needs to be between 1 and 3 - throw an `IllegalArgumentException` if it is not. You can derive the exception message from the provided *unit tests*.

Your class should be *persistable* using *JPA*, so make sure to add the appropriate *annotations*. Use `@GeneratedValue` for your *ID* to outsource the task of generating keys to the persistence provider (i.e. the database).

Repository

Create a *repository*-class for managing the *Todos* and providing methods for *CRUD*-operations. Consider the following implementation details:

- *Inject* an `EntityManager` to store the entities in the database.
- All methods looking for specific entities (e.g. when deleting or updating) should throw a `NotFoundException` when no object with the specific key has been found.
- Make sure to mark all *relevant* methods (i.e. the ones performing changes) as `Transactional`.

- The method for *listing* all entities should sort the `Todo`s by their deadlines. Provide an additional method for filtering the `Todo`s by their priority.
    - **Both methods should use *JPQL* for sorting/filtering.**
    - Bonus: Use `NamedQueries` for well-organized code.

Boundary

Create a *REST endpoint* for accessing the methods in your *repository* from the last step. **Make sure to use the appropriate *HTTP* methods** so the following example calls are possible:

- `POST`-ing a *JSON* object to `http://localhost:8080/api/todos` should create a new *Todo*.
    - A successful `POST` should return the *URL* for *getting* the newly created item. See the *Hints* section below.
- `GET`-ting from `http://localhost:8080/api/todos/13` should return the *Todo* with *ID* 13.
    - If no item with the *ID* provided could be found, a 404 *status code* should be returned. This can be easily be achieved by throwing a `NotFoundException` as described above.
- `GET`-ting from `http://localhost:8080/api/todos/list` should return a list of all *Todos* appropriately sorted.
- `GET`-ting from `http://localhost:8080/api/todos/list/2` should return all *Todos* with a priority of 2.
- `PUT`-ing a *JSON* object to `http://localhost:8080/api/todos/7` should overwrite the *Todo* with *ID* 7 and return it.
- `DELETE`-ing at `http://localhost:8080/api/todos/5` should delete the *Todo* with *ID* 5.

N.B.: All methods should return `OK` if the operation was performed successfully.

Hints

Returning the created *URL*

The following example code returns the *URL* of a newly created `Haxi`:

```java
@Context
UriInfo uriInfo;

@POST
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.MEDIA_TYPE_WILDCARD)
public Response addHaxi(Haxi myHaxi) {
    Haxi haxiCreated = this.haxiRepository.addHaxi(myHaxi);
    UriBuilder uriBuilder = this.uriInfo.getAbsolutePathBuilder();
    uriBuilder.path(Long.toString(haxiCreated.getId()));
    return Response.created(uriBuilder.build()).build();
}
```