LAB 5.5: QUARKUS DEVELOP REVIEW

Autor: José Díaz

Github Repo: https://github.com/joedayz/quarkus-bcp-2025.git

Instructions

1. Examine the application in the `~/DO378/develop-review` directory with an editor, such as VSCodium or vim.

   1.1. Navigate to the `~/DO378/develop-review` directory.

```
[student@workstation ~]$ cd ~/DO378/develop-review
```

   1.2. Open the project with an editor, such as VSCodium or vim.

   1.3. Examine the `com.redhat.training.speaker.SpeakerResource` class. This class implements two endpoints:

   - GET `/speakers`: returns the list of `Speaker` instances persisted in memory.
   - POST `/speakers`: receives a JSON request, creates a `Speaker` instance, and persists the instance in memory. The endpoint returns an empty message with HTTP code `201`. The response headers include the `id` and `location` fields of the persisted instance.

   1.4. Examine the `com.redhat.training.speaker.Speaker` class. This class implements a basic representation of a speaker. It includes an autogenerated ID, the name, and the speaker organization.

2. Add the required dependencies to persist the application data in a PostgreSQL database, and to expose the API endpoints through the OpenAPI specification. Use the JDBC PostgreSQL driver, the Hibernate ORM, and Panache to persist the data.
3. Configure the application to connect to a PostgreSQL database. Use the `dev` profile to configure a connection to the database that runs locally.
   - The database is running at `localhost` on port `5432`.
   - The database name is `testing`.
   - The database has a standard user account, `developer`, which has the password `developer`.
   - Use the `drop-and-create` database generation strategy on the `dev` profile.
4. Remove the in-memory persistence layer and use Panache to persist the application data.
5. Implement a new feature to manage the conference talks assigned to speakers.

   Create a new entity with the following parameters:
   - Call the entity `Talk`.
   - Create the entity in the `com.redhat.training.speaker` package.
   - The entity must have the `String title` and `int duration` attributes.

   Each speaker can have multiple conference talks assigned, and the one-to-many relationship must use the `ALL` cascade type.
6. Improve the API documentation for the GET `/speakers`, and POST `/speaker` endpoints.
   - GET `/speakers`: document the operation summary, and `200` as the response code.
   - POST `/speakers`: document the operation summary, and `201` as the response code. Document the name, description and schema of the `id` and `location` response headers.
7. Execute the Quarkus application in dev mode, and then manually test the application by using the Swagger UI. You can use the following JSON to create a new speaker.

```
{
  "name": "Pablo",
  "organization": "Red Hat",
  "talks": [
    {
      "title": "Quarkus is fun",
      "duration": 15
    }
  ]
}
```

8. Update the REST API to support the removal of `Speaker` instances from the database.
   - The endpoint must use the `/speakers/{id}` path.
   - Throw a `NotFoundException` exception when the `Speaker` instance cannot be deleted from the database.

9. Update the REST API to support sorting and paginating the results returned from the GET `/speakers` endpoint.

   - Add a string query parameter called `sortBy` to sort speakers by `id` (default) or `name` columns. Use the `filterSortBy` method to filter invalid values received in the parameter.
   - Add a query parameter called `pageIndex` that selects a specific page from a paginated result. Define `0` as the default value.
   - Add a query parameter called `pageSize` that limits the number of records returned from the database. Define `25` as the default value.

10. Execute the tests to validate the code changes. Optionally, you can use the Swagger UI to manually validate the changes before executing the tests.

Evaluation

As the `student` user on the `workstation` machine, use the `lab` command to grade your work. Correct any reported failures and rerun the command until successful.

```
[student@workstation develop-review]$ lab grade develop-review
```

Finish

Run the `lab finish` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish develop-review
```

This concludes the section.

SOLUCION

Instructions

1. Examine the application in the `~/DO378/develop-review` directory with an editor, such as VSCodium or vim.

   1.1. Navigate to the `~/DO378/develop-review` directory.

```
[student@workstation ~]$ cd ~/DO378/develop-review
```

   1.2. Open the project with an editor, such as VSCodium or vim.

   1.3. Examine the `com.redhat.training.speaker.SpeakerResource` class. This class implements two endpoints:

   - GET `/speakers`: returns the list of `Speaker` instances persisted in memory.
   - POST `/speakers`: receives a JSON request, creates a `Speaker` instance, and persists the instance in memory. The endpoint returns an empty message with HTTP code `201`. The response headers include the `id` and `location` fields of the persisted instance.

   1.4. Examine the `com.redhat.training.speaker.Speaker` class. This class implements a basic representation of a speaker. It includes an autogenerated ID, the name, and the speaker organization.

2. Add the required dependencies to persist the application data in a PostgreSQL database, and to expose the API endpoints through the OpenAPI specification. Use the JDBC PostgreSQL driver, the Hibernate ORM, and Panache to persist the data.

   2.1. Return to the terminal window. Then, use the Maven command to install the `quarkus-jdbc-postgresql` and `quarkus-hibernate-orm-panache` extensions.

```
[student@workstation develop-review]$ mvn quarkus:add-extension \
-Dextensions="jdbc-postgresql,hibernate-orm-panache"
...output omitted...
[INFO] [SUCCESS] ... Extension io.quarkus:quarkus-jdbc-postgresql has been
 installed
[INFO] [SUCCESS] ... Extension io.quarkus:quarkus-hibernate-orm-panache has been
 installed
...output omitted...
```

   2.2. Use the Maven command to install the `quarkus-smallrye-openapi` extension.

```
[student@workstation develop-review]$ mvn quarkus:add-extension \
-Dextension=smallrye-openapi
...output omitted...
[INFO] [SUCCESS] ...  Extension io.quarkus:quarkus-smallrye-openapi has been
 installed
...output omitted...
```

3. Configure the application to connect to a PostgreSQL database. Use the `dev` profile to configure a connection to the database that runs locally.

   - The database is running at `localhost` on port `5432`.
   - The database name is `testing`.
   - The database has a standard user account, `developer`, which has the password `developer`.
   - Use the `drop-and-create` database generation strategy on the `dev` profile.

   3.1. Open the `src/main/resources/application.properties` file, and set the JDBC URL for the `dev` profile. Connect the application to the `testing` database that is running at `localhost` on port `5432`.

```
%dev.quarkus.datasource.jdbc.url = jdbc:postgresql://localhost:5432/testing
```

   3.2. Define the connection username and password for the `dev` profile.

```
%dev.quarkus.datasource.jdbc.url = jdbc:postgresql://localhost:5432/testing
%dev.quarkus.datasource.username = developer
%dev.quarkus.datasource.password = developer
```

   3.3. Set the database generation strategy as `drop-and-create` for the `dev` profile.

```
%dev.quarkus.datasource.jdbc.url = jdbc:postgresql://localhost:5432/testing
%dev.quarkus.datasource.username = developer
%dev.quarkus.datasource.password = developer
%dev.quarkus.hibernate-orm.database.generation = drop-and-create
```

4. Remove the in-memory persistence layer and use Panache to persist the application data.

   4.1. Transform the `Speaker` class into a Panache entity:

   - Remove the `id` attribute to use the internal Panache ID.
   - Annotate the class with the `javax.persistence.Entity` annotation.
   - Extend the class from the `io.quarkus.hibernate.orm.panache.PanacheEntity` class.

```
package com.redhat.training.speaker;

...code omitted...

@Entity
public class Speaker extends PanacheEntity {
    public String name;
    public String organization;
}
```

   4.2. Open the `SpeakerResource` class, then remove the `speakers` attribute.

```
...code omitted...

@Path("/speakers")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class SpeakerResource {

    @GET
    public Set<Speaker> getSpeakers() {
...code omitted...
```

   4.3. Update the `getSpeakers()` method. Use the `findAll()` method to retrieve a list of speakers from the database.

```
@GET
public List<Speaker> getSpeakers() {
    return Speaker
            .findAll()
            .list();
}
```

   4.4. Update the `createSpeaker()` method. Annotate the method with the `@Transactional` annotation. Then, use the `persist()` method of the Panache entity to store a new `Speaker` instance in the database.

```
@POST
@Transactional
public Response createSpeaker(Speaker newSpeaker, @Context UriInfo uriInfo) {
    newSpeaker.persist();

    return Response.created(generateUriForSpeaker(newSpeaker, uriInfo))
        .header("id", newSpeaker.id)
        .build();
}
```

5. Implement a new feature to manage the conference talks assigned to speakers.

   Create a new entity with the following parameters:

   - Call the entity `Talk`.
   - Create the entity in the `com.redhat.training.speaker` package.
   - The entity must have the `String title` and `int duration` attributes.

   Each speaker can have multiple conference talks assigned, and the one-to-many relationship must use the `ALL` cascade type.

   5.1. Create a Panache entity that represents each talk:

   - Call the entity `Talk`.
   - Create the entity in the `com.redhat.training.speaker` package.
   - The entity must have a `String title` and `int duration` attributes.

```
package com.redhat.training.speaker;

import io.quarkus.hibernate.orm.panache.PanacheEntity;

import javax.persistence.Entity;

@Entity
public class Talk extends PanacheEntity {
    public String title;
    public int duration;
}
```

   5.2. Update the `Speaker` class to define a one-to-many relationship with the `Talk` class. Set the cascade type to `ALL`.

```
...code omitted...

@Entity
public class Speaker extends PanacheEntity {
    public String name;
    public String organization;
```

```
    @OneToMany(cascade = CascadeType.ALL)
    public List<Talk> talks;
}
```

6. Improve the API documentation for the GET `/speakers`, and POST `/speaker` endpoints.

   - GET `/speakers`: document the operation summary, and `200` as the response code.
   - POST `/speakers`: document the operation summary, and `201` as the response code. Document the name, description and schema of the `id` and `location` response headers.

   6.1. Open the `SpeakerResource` class and define the OpenAPI documentation for the GET `/speakers` endpoint.

```
...code omitted...

@GET
@Operation(summary = "Retrieves the list of speakers")
@APIResponse(responseCode = "200")
public List<Speaker> getSpeakers() {
    return Speaker
            .findAll()
            .list();
}

...code omitted...
```

   6.2. Define the OpenAPI documentation for the POST `/speakers` endpoint.

```
...code omitted...

@POST
@Transactional
@Operation(summary = "Adds a speaker")
@APIResponse(
        responseCode = "201",
        headers = {
                @Header(
                        name = "id",
                        description = "ID of the created entity",
                        schema = @Schema(implementation = Integer.class)
                ),
                @Header(
                        name = "location",
                        description = "URI of the created entity",
                        schema = @Schema(implementation = String.class)
                ),
        },
        description = "Entity successfully created"
)
public Response createSpeaker(Speaker newSpeaker, @Context UriInfo uriInfo) {
...code omitted...
```

7. Execute the Quarkus application in dev mode, and then manually test the application by using the Swagger UI. You can use the following JSON to create a new speaker.

```
{
  "name": "Pablo",
  "organization": "Red Hat",
  "talks": [
    {
      "title": "Quarkus is fun",
      "duration": 15
    }
  ]
}
```

7.1. Return to the terminal window, and then use the Maven command to start the Quarkus application in dev mode.

```
[student@workstation develop-review]$ mvn quarkus:dev
...output omitted...
... INFO [io.quarkus] ... Listening on: http://localhost:8080
...output omitted...
```

7.2. Open the web browser and navigate to http://localhost:8080/q/swagger-ui.

7.3. Click **Try it out** in the GET `/speakers` endpoint, and then click **Execute**. The Swagger UI displays an empty response body:

```
[]
```

7.4. Click **Try it out** in the POST `/speakers` endpoint. Set the following content as the request body, and then click **Execute**.

```
{
  "name": "Pablo",
  "organization": "Red Hat",
  "talks": [
    {
      "title": "Quarkus is fun",
      "duration": 15
    }
  ]
}
```

The Swagger UI displays a response with `201` code and the following response headers:

```
content-length: 0
id: 1
location: http://localhost:8080/speakers/1
```

7.5. Click **Execute** in the GET `/speakers` endpoint. The Swagger UI displays a list of speakers as the response body:

```
[
  {
    "id": 1,
    "name": "Pablo",
    "organization": "Red Hat",
    "talks": [
      {
        "id": 2,
        "duration": 15,
        "title": "Quarkus is fun"
      }
    ]
  }
]
```

8. Update the REST API to support the removal of `Speaker` instances from the database.

   - The endpoint must use the `/speakers/{id}` path.
   - Throw a `NotFoundException` exception when the `Speaker` instance cannot be deleted from the database.

   8.1. Open the `SpeakerResource` class, and then create a DELETE `/{id}` endpoint.

```
...code omitted...

@DELETE
@Path("/{id}")
@Transactional
public void deleteSpeaker(@PathParam("id") Long id) {
    if (!Speaker.deleteById(id)) {
        throw new NotFoundException();
    }
}

private URI generateUriForSpeaker(Speaker speaker, UriInfo uriInfo) {
...code omitted...
```

9. Update the REST API to support sorting and paginating the results returned from the GET `/speakers` endpoint.

   - Add a string query parameter called `sortBy` to sort speakers by `id` (default) or `name` columns. Use the `filterSortBy` method to filter invalid values received in the parameter.
   - Add a query parameter called `pageIndex` that selects a specific page from a paginated result. Define `0` as the default value.
   - Add a query parameter called `pageSize` that limits the number of records returned from the database. Define `25` as the default value.

   9.1. Update the `getSpeakers` method. Add a query parameter called `sortBy` with `id` as the default value. Then, update the `findAll` Panache method to use a `Sort` instance with the parameter value.

```
@GET
@Operation(summary = "Retrieves the list of speakers")
@APIResponse(responseCode = "200")
public List<Speaker> getSpeakers(
        @DefaultValue("id") @QueryParam("sortBy") String sortBy
) {
    return Speaker
            .findAll(
                Sort.by(filterSortBy(sortBy))
            )
            .list();
}
```

9.2. Update the `getSpeakers` method. Add a query parameter called `pageIndex` with `0` as the default value. Add a query parameter called `pageSize` with `25` as the default value. Finally, use the `page` Panache method to limit the collection limits.

```
@GET
@Operation(summary = "Retrieves the list of speakers")
@APIResponse(responseCode = "200")
public List<Speaker> getSpeakers(
        @DefaultValue("id") @QueryParam("sortBy") String sortBy ,
        @DefaultValue("0") @QueryParam("pageIndex") int pageIndex,
        @DefaultValue("25") @QueryParam("pageSize") int pageSize
) {
    return Speaker
            .findAll(
                Sort.by(filterSortBy(sortBy))
            )
            .page(pageIndex, pageSize)
            .list();
}
```

10. Execute the tests to validate the code changes. Optionally, you can use the Swagger UI to manually validate the changes before executing the tests.

10.1. Return to the terminal window. Then press q to stop the running application, and finally use the Maven command to execute the tests.

```
[student@workstation develop-review]$ mvn test
...output omitted...
[INFO]
[INFO] Tests run: 10, Failures: 0, Errors: 0, Skipped: 0
[INFO]
...output omitted...
```

Evaluation

As the `student` user on the `workstation` machine, use the `lab` command to grade your work. Correct any reported failures and rerun the command until successful.

```
[student@workstation develop-review]$ lab grade develop-review
```

Finish

Run the `lab finish` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish develop-review
```

This concludes the section.

enjoy!

Jose