LAB 5.5: QUARKUS DEVELOP REVIEW

Autor: José Díaz

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

INSTRUCCIONES

1. Abrir el proyecto **04-develop-review-start**
2. Examina la clase **com.bcp.training.speaker.SpeakerResource**. Esta clase implementa dos endpoints:
    a. **GET /speakers**: retorna una lista de instancias Speaker guardadas en memoria.
    b. **POST /speakers**: recibe un request JSON, crea una instancia Speaker, y persiste la instancia en memoria. El endpoint retorna un mensaje vacío con HTTP code 201. Los headers del response incluyen los campos id y location de la instancia guardada.
3. Examina la clase **com.bcp.training.speaker.Speaker**. Esta clase implementa una representación basica de un speaker. Esto incluye un ID autogenerado, el name y la organización del speaker.
4. Agregar las dependencias requeridas para persistir la data de la aplicación en una base de datos PostgreSQL y exponer los endpoints del API, a través de, la especificación OpenAPI. Usa el JDBC driver PostgreSQL, el Hibernate ORM, y Panache para persistir la data.
5. Configurar la aplicación para conectarse a una base de datos PostgreSQL. Usa el profile dev para configurar una conexión a la base de datos que corre localmente.
    a. La base de datos esta corriendo en localhost y puerto 5432.
    b. El nombre de la base de datos es testing.
    c. La base de datos tiene una cuenta de usuario, developer, y password developer.
    d. Usa la estrategia de generación de base de datos drop-and-create en el profile dev.
6. Elimina la capa de persistencia in-memory y usa panache para persistir la data de la aplicación.
7. Implementa una nueva característica para administrar los talks de la conferencia asignado a los speakers. Crea un nuevo entity con los siguientes parámetros:
    a. Llama a la entidad Talk.
    b. Crea la entidad en el paquete com.bcp.training.speaker.
    c. La entidad debe tener los atributos String title e int duration.

    Cada speaker puede tener multiples talks en la conferencia asignados, y una relación one-to-manyu debe usar el ALL cascade type.
8. Agrega la documentación del API para los endpoints GET /speakers y POST /speaker.
    a. GET /speakers: documenta el summary de la operación y 200 como código de respuesta.
    b. POST /speakers: documenta el summary de la operación y 201 como código de respuesta. Documenta el name, description y schema para los response headersl id y location.
9. Ejecuta la aplicación en modo quarkus dev, y luego manualmente prueba la aplicación usando el Swagger UI. Tu puedes usar el siguiente JSON para crear un nuevo speaker.

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

}
```

10. Actualizar el endpoint REST para soportar la eliminación de instancias Speaker de la base de datos.
    a. El endpoint debe usar el path /speakers/{id}.
    b. Arroja una exception NotFoundException cuando la instancia Speaker no puede ser eliminada de la base de datos.
11. Actualiza el REST API para soportar ordenamiento y paginar los resultados retornados desde el endpoint GET /speakers.
    a. Agrega un query parameter llamado sortBy para ordenar speakers por id (default) o la columna name. Usa el método filterSortBy para filtrar valores invalidos recibidos en el parámetro.
    b. Agrega un query parameter llamado pageIndex que selecciona una página específica de un resultado paginado. Define un 0 como el valor por defecto.
    c. Agrega un query parameter llamado pageSize que limita el número de registros retornados de la base de datos. Define 25 como el valor por defecto.

12. Ejecuta los tests para validar los cambios de código. Opcionalmente tu puedes usar el Swagger UI para manualmente validar los cambios antes de ejecutar las pruebas.

SOLUCION

1. Abrir el proyecto **04-develop-review-start**
2. Examina la clase **com.bcp.training.speaker.SpeakerResource**. Esta clase implementa dos endpoints:
    a. **GET /speakers**: retorna una lista de instancias Speaker guardadas en memoria.
    b. **POST /speakers**: recibe un request JSON, crea una instancia Speaker, y persiste la instancia en memoria. El endpoint retorna un mensaje vacío con HTTP code 201. Los headers del response incluyen los campos id y location de la instancia guardada.
3. Examina la clase **com.bcp.training.speaker.Speaker**. Esta clase implementa una representación basica de un speaker. Esto incluye un ID autogenerado, el name y la organización del speaker.
4. Agregar las dependencias requeridas para persistir la data de la aplicación en una base de datos PostgreSQL y exponer los endpoints del API, a través de, la especificación OpenAPI. Usa el JDBC driver PostgreSQL, el Hibernate ORM, y Panache para persistir la data.

4.1 Retorna a la terminal de windows. Luego, usa el comando maven para instalar las extensiones: quarkus-jdbc-postgresql y quarkus-hibernate-orm-panache.

mvn quarkus:add-extension -Dextensions=”jdbc-postgresql, hibernate-orm-panache”

4.2 Usa el comando maven para instalar la extension quarkus-smallrye-openapi.

mvn quarkus:add-extension -Dextensions=smallrye-openapi

5. Configurar la aplicación para conectarse a la base de datos PostgreSQL. Usa el profile dev para configurar una conexión a la base de datos que corre localmente.
    a. La base de datos esta corriendo en localhost y puerto 5432
    b. El nombre de la base de datos es testing
    c. La base de datos tiene una cuenta developer y password developer.
    d. Usa la estrategia de generación de base de datos drop-and-create en el profile dev.

    5.1. Abre el src/main/resources/application.properties y establece el JDBC URL para el profile dev. Conecta la aplicación a la base de datos testing que esta corriendo en localhost y puerto 5432.

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

    5.2. Define el username y password de conexión a la base de datos en el profile dev.

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

    5.3. Establece la estrategia de generación de la base de datos en el profile dev a drop-and-create.

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

6. Eliminar la capa de persistencia en memoria y usar Panache para persistir la data de la aplicación.

    6.1. Transforma la clase Speaker a Panache Entity
    - Elimina el atributo id para usar el internal Panache ID.
    - Anota la clase con jakarta.persistence.Entity.
    - Hereda de la clase io.quarkus.hibernate.orm.panache.PanacheEntity.

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

6.2. Abre la clase SpeakerResource, luego elimina el atributo speakers.

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

    @GET
    public List<Speaker> getSpeakers(
```

6.3. Actualiza el método getSpeakers(). Usa el método findAll() para recuperar la lista de speakers de la base de datos.

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

6.4. Actualiza el método createSpeaker(). Anota el método con @Transactional. Luego usa el método persist() de la entidad Panache para guardar una instancia de Speaker en la base de datos.

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

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

7. Implementa una nueva característica para administrar las talks de la conferencias asignadas a los speakers. Crea una nueva entidad con los siguientes parámetros:
    a. Llama a la entidad Talk
    b. Crea la entidad en el paquete com.bcp.training.speaker.
    c. La entidad debe tener los atributos String title e int duration.

    Cada speaker puede tener múltiples talks asignadas, y la relación one-to-many debe usar el ALL cascade type.

    7.1. Crea una entidad Panache que representa cada talk:
    - Llama a la entidad Talk
    - Crea la entidad en el paquete com.bcp.training.speaker.
    - La entidad debe tener atributos String title e int duration.

```
package com.bcp.training.speaker;

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

import jakarta.persistence.Entity;

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

    7.2. Actualiza la clase Speaker para definir una relación one-to-many con la clase Talk. Establece el tipo de cascade a ALL.

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

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

```
}
```

8. Mejorar la documentación de la API para los endpoints GET /speakers y POST /speakers.
    - GET /speakers: documenta el operation summary y 200 como response code.
    - POST /speakers: documenta el operation summary y 201 como response code. Documenta el name, description, y schema de los response headers id y location.

    8.1. Abre la clase SpeakerResource y define la documentación OpenAPI para el endpoint GET /speakers.

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

    8.2. Define la documentación OpenAPI para el endpoint POST /speakers.

```
    @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) {…}
```

9. Ejecuta la aplicación Quarkus en modo dev y luego manualmente probar la aplicación usando el Swagger UI. Tu puedes usar el siguiente JSON para crear un nuevo speaker.

```
{
```

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

9.1. Retornar a la terminal de windows, y luego usar el comando maven para iniciar quarkus en modo dev.

mvn quarkus:dev

9.2. Abre el browser y navega a http://localhost:8080/q/swagger-ui
9.3. Clic en Try it out in el GET /speakers y luego clic en Execute. El swagger UI muestra una respuesta vacía: []

9.4. Clic en Try it out en el POST /speakers. Establece el siguiente contenido como el request body y luego en clic Execute.

```
{

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

El swagger UI muestra una respuesta con código 201 y los siguientes responses headers:

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

9.5. Clic en Execute en el GET /speakers. El Swagger UI muestra una lista de speakers en el response body:

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


10. Actualiza el REST API para soportar la eliminación de las instancias Speaker de la base de datos.
    - El endpoint debe usar el path /speakers/{id}.
    - Throw una exception NotFoundException cuando la instancia Speaker no puede ser eliminada de la base de datos.

    10.1 Abre el SpeakerResource y crea un DELETE /{id}.

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

11. Actualiza el REST API para soportar sorting y pagination en el GET /speakers:
    a. Agrega un query parameter llamado sortBy para ordenar speakers por id (default) o la columna name. Usa el método filterSortBy para filtrar valores invalidos recibidos en el parámetro.
    b. Agrega un query parameter llamado pageIndex que selecciona una página específica de un resultado paginado. Define un 0 como el valor por defecto.
    c. Agrega un query parameter llamado pageSize que limita el número de registros retornados de la base de datos. Define 25 como el valor por defecto.

```java
@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();
    }
```

12. Ejecuta los tests para validar los cambios de código. Opcionalmente tu puedes usar el Swagger UI para manualmente validar los cambios antes de ejecutar las pruebas.

    mvn test

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

    Enjoy!

    José