Springboot for Quarkus users

https://youtu.be/9SGDpanrc8U

Errors :

Failed to determine a suitable driver class :

- Failed to connect to a database.
- Comment the « spring-boot-starter-data-jpa » if you have no database.

Execution :

- Tomcat webserver

Annotations :

@RestController :

Make the class able to serve REST endpoints

@GetMapping :

@GET in quarkus

@PostMapping :

@POST in quarkus

@DeleteMapping :

@DELETE in quarkus

@PutMapping :

@PUT in quarkus

@RequestMapping :

@Path in quarkus

@Autowired :

@Inject in quarkus

@Component :

Specifies that a class has to be instanciated. (Specifies that this class is a Bean).

@Service :

Is exactly the same as @Component but is more human readable, as we know exactly if a class is wether a service(@Service), a controller(@Controller) or a Repository(@Repository).

You can use @Component instead of any other annotation that inherits from it.

@Entity

Needed so hibernate can map our object into the database.
Often written just above @Table which specifies that our object is an entity of type table.

Using Quarkus we can tell our class to extend the PanacheEntity abstract class, which allows hibernate to do the class object to database object mapping for us.

@Table

Needed so hibernante knows the entity (Java class annotatoed with Entity) it's trying to map with an object of our database is a table.

@Id

Specifies that our object has an Id. (This annotation is always required, unless you pass an id when you insert a new object in your table).

Same in Quarkus.

@SequenceGenerator

Sequences are typically used with postgreSQL databases.
Sequence is the schema object that generates a sequence of numbers in ascending or descending order.

Same in Quarkus.

```
@SequenceGenerator(
    name = "student_sequence",
    sequenceName = "student_sequence",
    allocationSize = 1
)
```

@GeneratedValue

Specifies a sequence to use.

Same in Quarkus.

```
@GeneratedValue(
    strategy = GenerationType.SEQUENCE,
    generator = "student_sequence"
)
```

@Repository

@RegisterRestClient in Quarkus.

@Configuration

Indicates that a class declares one or more @Bean methods.

@Bean

Applied on a method to specify that it returns a bean to be managed by Spring context.

Spring Bean annotation is usually declared in Configuration classes methods.

Bean methods may reference other @Bean methods in the same class by calling them directly.

@Transient

A transient variable specifies that we don't want that variable to be mapped as a column in our database table.

@RequestBody

Maps the posted values into our class type.

```
@PostMapping
public void registerNewStudent(@RequestBody Student student) {
    studentService.addNewStudent(student);
}
```

@Query

Allows to pass a query using JPQL (Java Persistence Query Language).

@Transactional

Means that you don't have to implement any JPQL query.
It uses the status from your entity to check whether you can or cannot update.
You can then use the status to update your entity in your database.
This puts the entity into a manage state ().

Same in quarkus.

@RequestParam

Used to bind a web request parameter to a method parameter.

Specifications :

JSON returns :

List<Object/Type> returns JSON by default.

```
@GetMapping
    public List<String> hello() {
        return List.of("Hello", "World");
    }
```

Controllers :

Controller classes are called Resource classes in quarkus.
Ex : (Quarkus)GreetingResource.java = (Stringboot)GreetingController.java

Services :

Services are the same as in Quarkus.
Services are used to consume controllers.

Services injection are specified by the @Autowired annotation.

Configurations :

Configuration classes, are classes which are often annotated with @Repository.

They are used to perform action in between the Data access layer and the Service layer.
Example : Add new students, Delete new students, . . .

Data Layer Access :

Interface that is responsible for data access.

Using spring boot, we use the @Repository annotation instead of @RegisterRestClient in Quarkus.
These classes are called **repositories** (ex: StudentRepository) instead of **services in Quarkus**.

Our interface has to extends the JpaRepository interface.
It attends the type of object which this repository is going to work with (ex: Student)

and also the id which is the type of the id of our students class.

```
@Repository
public interface StudentRepository extends JpaRepository<Student, Long> {

    public interface StudentRepository extends JpaRepository<Student, Long> {

    }
}
```

To use a repository, we declare it exactly as we would declare a service (don't forget to use the @Autowired annotation).

JPA :

Jakarta Persistence.

Java Persistence API is a collection of classes and methods to persistently store the vast amounts of data into a database.

Usage :

Services Usage :

Annotate your services with the @Service annotation.

Student controller has a final StudentService variable which is going to inherit our StudentService object instanciation :

```
private final StudentService studentService;

@Autowired
    public StudentController(StudentService studentService) {
        this.studentService = studentService;
    }
```

Map objects of a class to your database table :

```
@Entity
@Table
public class Student extends {

    @Id
    @SequenceGenerator(
            name = "student_sequence",
            sequenceName = "student_sequence",
            allocationSize = 1
    )
    @GeneratedValue(
            strategy = GenerationType.SEQUENCE,
            generator = "student_sequence"
    )

    private Long id;
    private String name;
    private String email;
    private LocalDate dob;
    private Integer age;

    public Student() {
    }

    public Student(Long id, String name, ...) {
        this.id = id;
        this.name = name;
        this.email = email;
        this.dob = dob;
        this.age = age;
    }

}
```

Application.properties file

Database basic properties :

```
spring.datasource.url=jdbc:mysql://localhost:3306/student
spring.datasource.username=root
springdatabase.datasource.password=1234
spring.jpa.hibernate.ddl-auto=update/none/create/validate/create-drop/update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect
spring.jpa.properties.hibernate.format_sql=true
```

Data Layer Access

In the data layer we use repositories to persist actions on the database (using the @Repository annotation).

```
@Repository
public interface StudentRepository extends JpaRepository<Student, Long> {

}
```

Performing queries

We can uncomment the @Query annotation to specify what query we want to pass.
The s is an alias.
The Student corresponds to the Student class.

```
@Repository
public interface StudentRepository extends JpaRepository<Student, Long> {
 // @Query("SELECT s FROM Student s WHERE s.email = ?1")
    Optional<Student> findStudentByEmail(String email);
}
```

Display exception messages :

```
spring.error.include-message=always
```

Delete queries :

```
@DeleteMapping(path = "{studentId}")
public void deleteStudent(@PathVariable("studentId") Long studentId) {
    studentService.deleteStudent(studentId);
}
```

```
public void deleteStudent(Long studentId) {
    boolean exists = studentRepository.existsById(studentId);
    if (!exists) {
        throw new IllegalStateException(
               "student with id " + studentId + " does not exists");
    }
    studentRepository.deleteById(studentId);
}
```

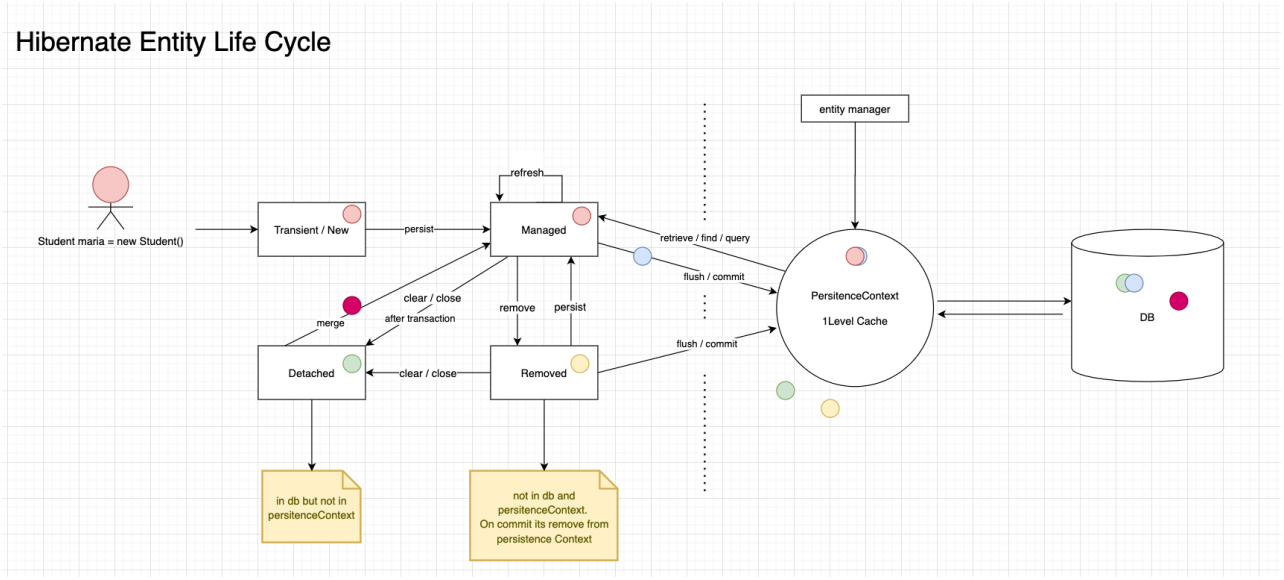
Put queries :

When using put queries it's recommended to use the @PutMapping and @Transactional annotations. Here we also use the @RequestParam annotation.

```
@PutMapping(path = "{studentId}")
public void updateStudent(
    @PathVariable("studentId") Long studentId,
    @RequestParam(required = false) String name,
    @RequestParam(required = false) String email) {
    studentService.updateStudent(studentId, name, email);
}
```

```
@Transactional
public void updateStudent(Long studentId, String name, String email) {
    Student student = studentRepository.findById(studentId)
        .orElseThrow(() -> new IllegalStateException(
            "student with id " + studentId + " does not exists"));
    if (name != null && name.length() > 0 &&
!Objects.equals(student.getName(), name)) {
        student.setName(name);
    }
    if (email != null && email.length() > 0 &&
!Objects.equals(student.getEmail(), email)) {
        Optional<Student> studentOptional =
studentRepository.findStudentByEmail(email);
        if (studentOptional.isPresent()) {
            throw new IllegalStateException("email taken");
        }
        student.setEmail(email);
    }
}
```

Hibernate Entity Lifecycle Model



Source: https://www.filepicker.io/api/file/UfnqU7j2T4OUDRMcRKgT

EntityManager :

The EntityManager API is **used to access a database in a particular unit of work**. It is used to create and remove persistent entity instances, to find entities by their primary key identity, and to query over all entities. This interface is similar to the Session in Hibernate. Persistence context.

Persistence context :

The persistence context is **the first-level cache where all the entities are fetched from the database or saved to the database**. It sits between our application and persistent storage. Persistence context keeps track of any changes made into a managed entity.

Flush :

Flushing the session **forces Hibernate to synchronize the in-memory state of the Session with the database** (i.e. to write changes to the database).

States :

- Transient
- Managed
- Removed
- Detached

Transient state :

Lifecycle of a newly instantiated entity object is called transient.

Managed state :

Lifecycle of an entity changes to Managed when you provide that entity to the EntityManager persist method.

The entity also gets attached to the current persistence context.

Detached state :

An entity that was previously managed but is no longer attached to the current persistence context is in the lifecycle state detached.

For example:

> When we persist an entity and we want our EntityManager to return us an entity, that returned entity will be in the lifecycle state detached.

Removed state :

When calling the remove method on the EntityManager, the entity doesn't gets removed until the next flush of the persistence context.
But the lifecycle of the entity is directly in to removed state.

Made by :

Matthias Brat

Socials:

github.com/matthiasbrat

stackoverflow.com/users/17921879/matthias-b

matthiasbrat.dev