Quarkus - Extension for Spring Data API

While users are encouraged to use Hibernate ORM with Panache for Relational Database access, Quarkus provides a compatibility layer for Spring Data JPA repositories in the form of the `spring-data-jpa` extension.

> This technology is considered preview.
>
> In *preview*, backward compatibility and presence in the ecosystem is not guaranteed. Specific improvements might require to change configuration or APIs and plans to become *stable* are under way. Feedback is welcome on our mailing list or as issues in our GitHub issue tracker.
>
> For a full list of possible extension statuses, check our FAQ entry.

Prerequisites

To complete this guide, you need:

- less than 15 minutes
- an IDE
- JDK 1.8+ installed with `JAVA_HOME` configured appropriately
- Apache Maven 3.6.3

Solution

We recommend that you follow the instructions in the next sections and create the application step by step. However, you can go right to the completed example.

Clone the Git repository: `git clone https://github.com/quarkusio/quarkus-quickstarts.git`, or download an archive.

The solution is located in the `spring-data-jpa-quickstart` directory.

Creating the Maven project

First, we need a new project. Create a new project with the following command:

```
mvn io.quarkus:quarkus-maven-plugin:1.7.2.Final:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=spring-data-jpa-quickstart \
    -DclassName="org.acme.spring.data.jpa.FruitResource" \
    -Dpath="/greeting" \
    -Dextensions="spring-data-jpa,resteasy-jsonb,quarkus-jdbc
-postgresql"
cd spring-data-jpa-quickstart
```

This command generates a Maven project with a REST endpoint and imports the `spring-data-jpa` extension.

If you already have your Quarkus project configured, you can add the `spring-data-jpa` extension to your project by running the following command in your project base directory:

```
./mvnw quarkus:add-extension -Dextensions="spring-data-jpa"
```

This will add the following to your `pom.xml`:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-spring-data-jpa</artifactId>
</dependency>
```

Define the Entity

Throughout the course of this guide, the following JPA Entity will be used:

```
package org.acme.spring.data.jpa;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Fruit {

    @Id
    @GeneratedValue
    private Long id;

    private String name;

    private String color;
```

```java
    public Fruit() {
    }

    public Fruit(String name, String color) {
        this.name = name;
        this.color = color;
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }
}
```

Configure database access properties

Add the following properties to `application.properties` to configure access to a local PostgreSQL instance.

```
quarkus.datasource.db-kind=postgresql
quarkus.datasource.username=quarkus_test
quarkus.datasource.password=quarkus_test
quarkus.datasource.jdbc.url=jdbc:postgresql:quarkus_test
quarkus.datasource.jdbc.max-size=8
quarkus.datasource.jdbc.min-size=2
quarkus.hibernate-orm.database.generation=drop-and-create
```

This configuration assumes that PostgreSQL will be running locally.

A very easy way to accomplish that is by using the following Docker command:

```
docker run --ulimit memlock=-1:-1 -it --rm=true --memory
-swappiness=0 --name quarkus_test -e POSTGRES_USER=quarkus_test -e
POSTGRES_PASSWORD=quarkus_test -e POSTGRES_DB=quarkus_test -p
5432:5432 postgres:11.5
```

If you plan on using a different setup, please change your `application.properties` accordingly.

Prepare the data

To make it easier to showcase some capabilities of Spring Data JPA on Quarkus, some test data should be inserted into the database by adding the following content to a new file named `src/main/resources/import.sql`:

```
INSERT INTO fruit(id, name, color) VALUES (1, 'Cherry', 'Red');
INSERT INTO fruit(id, name, color) VALUES (2, 'Apple', 'Red');
INSERT INTO fruit(id, name, color) VALUES (3, 'Banana', 'Yellow');
INSERT INTO fruit(id, name, color) VALUES (4, 'Avocado', 'Green');
```

Hibernate ORM will execute these queries on application startup.

Define the repository

It is now time to define the repository that will be used to access `Fruit`. In a typical Spring Data fashion create a repository like so:

```
package org.acme.spring.data.jpa;

import org.springframework.data.repository.CrudRepository;

import java.util.List;

public interface FruitRepository extends CrudRepository<Fruit,
Long> {

    List<Fruit> findByColor(String color);
}
```

The `FruitRepository` above extends Spring Data's `org.springframework.data.repository.CrudRepository` which means that all of the latter's methods are available to `FruitRepository`. Additionally `findByColor` is defined whose purpose is to return all Fruit entities that match the specified color.

Update the JAX-RS resource

With the repository in place, the next order of business is to update the JAX-RS resource that will use the `FruitRepository`. Open `FruitResource` and change its contents to:

```java
package org.acme.spring.data.jpa;

import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

import org.jboss.resteasy.annotations.jaxrs.PathParam;

import java.util.List;
import java.util.Optional;

@Path("/fruits")
public class FruitResource {

    private final FruitRepository fruitRepository;

    public FruitResource(FruitRepository fruitRepository) {
        this.fruitRepository = fruitRepository;
    }

    @GET
    @Produces("application/json")
    public Iterable<Fruit> findAll() {
        return fruitRepository.findAll();
    }


    @DELETE
    @Path("{id}")
    public void delete(@PathParam long id) {
        fruitRepository.deleteById(id);
    }

    @POST
    @Path("/name/{name}/color/{color}")
    @Produces("application/json")
    public Fruit create(@PathParam String name, @PathParam String
color) {
        return fruitRepository.save(new Fruit(name, color));
    }
```

```
    @PUT
    @Path("/id/{id}/color/{color}")
    @Produces("application/json")
    public Fruit changeColor(@PathParam Long id, @PathParam String
color) {
        Optional<Fruit> optional = fruitRepository.findById(id);
        if (optional.isPresent()) {
            Fruit fruit = optional.get();
            fruit.setColor(color);
            return fruitRepository.save(fruit);
        }

        throw new IllegalArgumentException("No Fruit with id " + id
+ " exists");
    }

    @GET
    @Path("/color/{color}")
    @Produces("application/json")
    public List<Fruit> findByColor(@PathParam String color) {
        return fruitRepository.findByColor(color);
    }
}
```

`FruitResource` now provides a few REST endpoints that can be used to perform CRUD operation on `Fruit`.

Note on Spring Web

The JAX-RS resource can also be substituted with a Spring Web controller as Quarkus supports REST endpoint definition using Spring controllers. See the Spring Web guide for more details.

Update the test

To test the capabilities of `FruitRepository` proceed to update the content of `FruitResourceTest` to:

```
package org.acme.spring.data.jpa;

import io.quarkus.test.junit.QuarkusTest;
import org.junit.jupiter.api.Test;

import static io.restassured.RestAssured.given;
import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.CoreMatchers.is;
import static org.hamcrest.CoreMatchers.notNullValue;
import static org.hamcrest.core.IsNot.not;
```

```java
@QuarkusTest
class FruitResourceTest {

    @Test
    void testListAllFruits() {
        //List all, should have all 3 fruits the database has
initially:
        given()
                .when().get("/fruits")
                .then()
                .statusCode(200)
                .body(
                        containsString("Cherry"),
                        containsString("Apple"),
                        containsString("Banana")
                );

        //Delete the Cherry:
        given()
                .when().delete("/fruits/1")
                .then()
                .statusCode(204)
        ;

        //List all, cherry should be missing now:
        given()
                .when().get("/fruits")
                .then()
                .statusCode(200)
                .body(
                        not(containsString("Cherry")),
                        containsString("Apple"),
                        containsString("Banana")
                );

        //Create a new Fruit
        given()
                .when().post("/fruits/name/Orange/color/Orange")
                .then()
                .statusCode(200)
                .body(containsString("Orange"))
                .body("id", notNullValue())
                .extract().body().jsonPath().getString("id");

        //List all, Orange should be present now:
        given()
                .when().get("/fruits")
                .then()
```

```
                .statusCode(200)
                .body(
                        not(containsString("Cherry")),
                        containsString("Apple"),
                        containsString("Orange")
                );
    }

    @Test
    void testFindByColor() {
        //Find by color that no fruit has
        given()
                .when().get("/fruits/color/Black")
                .then()
                .statusCode(200)
                .body("size()", is(0));

        //Find by color that multiple fruits have
        given()
                .when().get("/fruits/color/Red")
                .then()
                .statusCode(200)
                .body(
                        containsString("Apple"),
                        containsString("Strawberry")
                );

        //Find by color that matches
        given()
                .when().get("/fruits/color/Green")
                .then()
                .statusCode(200)
                .body("size()", is(1))
                .body(containsString("Avocado"));

        //Update color of Avocado
        given()
                .when().put("/fruits/id/4/color/Black")
                .then()
                .statusCode(200)
                .body(containsString("Black"));

        //Find by color that Avocado now has
        given()
                .when().get("/fruits/color/Black")
                .then()
                .statusCode(200)
                .body("size()", is(1))
                .body(
```

```
                           containsString("Black"),
                           containsString("Avocado")
                );
    }

}
```

The test can be easily run by issuing: `./mvnw test`

Package and run the application

Quarkus dev mode works with the defined repositories just like with any other Quarkus extension, greatly enhancing your productivity during the dev cycle. The application can be started in dev mode as usual using:

```
./mvnw compile quarkus:dev
```

Run the application as a native binary

You can of course create a native executable following the instructions of the this guide.

Supported Spring Data JPA functionalities

Quarkus currently supports a subset of Spring Data JPA's features, namely the most useful and most commonly used features.

An important part of this support is that all repository generation is done at build time thus ensuring that all supported features work correctly in native mode. Moreover, developers know at build time whether or not their repository method names can be converted to proper JPQL queries. This also means that if a method name indicates that a field should be used that is not part of the Entity, developers will get the relevant error at build time.

What is supported

The following sections described the most important supported features of Spring Data JPA.

Automatic repository implementation generation

Interfaces that extend any of the following Spring Data repositories are automatically implemented:

- `org.springframework.data.repository.Repository`

- `org.springframework.data.repository.CrudRepository`

- `org.springframework.data.repository.PagingAndSortingRepository`

- `org.springframework.data.jpa.repository.JpaRepository`

The generated repositories are also registered as beans so they can be injected into any other bean. Furthermore the methods that update the database are automatically annotated with `@Transactional`.

Fine tuning of repository definition

This allows user defined repository interfaces to cherry-pick methods from any of the supported Spring Data repository interfaces without having to extend those interfaces. This is particularly useful when for example a repository needs to use some methods from `CrudRepository` but it's undesirable to expose the full list of methods of said interface.

Assume for example that a `PersonRepository` that shouldn't extend `CrudRepository` but would like to use `save` and `findById` methods which are defined in said interface. In such a case, `PersonRepository` would look like so:

```
package org.acme.spring.data.jpa;

import org.springframework.data.repository.Repository;

public interface PersonRepository extends Repository<Person, Long>
{

    Person save(Person entity);

    Optional<Person> findById(Person entity);
}
```

Customizing individual repositories using repository fragments

Repositories can be enriched with additional functionality or override the default implementation of methods of the supported Spring Data repositories. This is best shown with an example.

A repository fragment is defined as so:

```
public interface PersonFragment {

    // custom findAll
    List<Person> findAll();

    void makeNameUpperCase(Person person);
}
```

The implementation of that fragment looks like this:

```java
import java.util.List;

import io.quarkus.hibernate.orm.panache.runtime.JpaOperations;

public class PersonFragmentImpl implements PersonFragment {

    @Override
    public List<Person> findAll() {
        // do something here
        return (List<Person>)
JpaOperations.findAll(Person.class).list();
    }

    @Override
    public void makeNameUpperCase(Person person) {
        person.setName(person.getName().toUpperCase());
    }
}
```

Then the actual `PersonRepository` interface to be used would look like:

```java
public interface PersonRepository extends JpaRepository<Person,
Long>, PersonFragment {

}
```

Derived query methods

Methods of repository interfaces that follow the Spring Data conventions can be automatically implemented (unless they fall into one of the unsupported cases listed later on). This means that methods like the following will all work:

```java
public interface PersonRepository extends CrudRepository<Person,
Long> {

    List<Person> findByName(String name);

    Person findByNameBySsn(String ssn);

    Optional<Person> findByNameBySsnIgnoreCase(String ssn);

    boolean existsBookByYearOfBirthBetween(Integer start, Integer
end);

    List<Person> findByName(String name, Sort sort);

    Page<Person> findByNameOrderByJoined(String name, Pageable
pageable);

    List<Person> findByNameOrderByAge(String name);

    List<Person> findByNameOrderByAgeDesc(String name, Pageable
pageable);

    List<Person> findByAgeBetweenAndNameIsNotNull(int
lowerAgeBound, int upperAgeBound);

    List<Person> findByAgeGreaterThanEqualOrderByAgeAsc(int age);

    List<Person> queryByJoinedIsAfter(Date date);

    Collection<Person> readByActiveTrueOrderByAgeDesc();

    Long countByActiveNot(boolean active);

    List<Person> findTop3ByActive(boolean active, Sort sort);

    Stream<Person> findPersonByNameAndSurnameAllIgnoreCase(String
name, String surname);
}
```

User defined queries

User supplied queries contained in the @Query annotation. For example things like the following all work:

```java
public interface MovieRepository extends CrudRepository<Movie,
Long> {

    Movie findFirstByOrderByDurationDesc();

    @Query("select m from Movie m where m.rating = ?1")
    Iterator<Movie> findByRating(String rating);

    @Query("from Movie where title = ?1")
    Movie findByTitle(String title);

    @Query("select m from Movie m where m.duration > :duration and
m.rating = :rating")
    List<Movie> withRatingAndDurationLargerThan(@Param("duration")
int duration, @Param("rating") String rating);

    @Query("from Movie where title like concat('%', ?1, '%')")
    List<Object[]> someFieldsWithTitleLike(String title, Sort
sort);

    @Modifying
    @Query("delete from Movie where rating = :rating")
    void deleteByRating(@Param("rating") String rating);

    @Modifying
    @Query("delete from Movie where title like concat('%', ?1,
'%')")
    Long deleteByTitleLike(String title);

    @Modifying
    @Query("update Movie m set m.rating = :newName where m.rating =
:oldName")
    int changeRatingToNewName(@Param("newName") String newName,
@Param("oldName") String oldName);

    @Modifying
    @Query("update Movie set rating = null where title =?1")
    void setRatingToNullForTitle(String title);

    @Query("from Movie order by length(title)")
    Slice<Movie> orderByTitleLength(Pageable pageable);
}
```

All methods that are annotated with @Modifying will automatically be annotated with @Transactional.

Naming Strategies

Hibernate ORM maps property names using a physical naming strategy and an implicit naming strategy. If you wish to use Spring Boot's default naming strategies, the following properties need to be set:

```
quarkus.hibernate-orm.physical-naming-
strategy=org.springframework.boot.orm.jpa.hibernate.SpringPhysicalN
amingStrategy
quarkus.hibernate-orm.implicit-naming-
strategy=org.springframework.boot.orm.jpa.hibernate.SpringImplicitN
amingStrategy
```

More examples

An extensive list of examples can be seen in the integration tests directory which is located inside the Quarkus source code.

What is currently unsupported

- Methods of the `org.springframework.data.repository.query.QueryByExampleExecutor` interface - if any of these are invoked, a Runtime exception will be thrown.

- QueryDSL support. No attempt will be made to generate implementations of any of the QueryDSL related repositories.

- Customizing the base repository for all repository interfaces in the code base.

  - In Spring Data JPA this is done by registering a class that extends `org.springframework.data.jpa.repository.support.SimpleJpaRepository` however in Quarkus this class is not used at all (since all the necessary plumbing is done at build time). Similar support might be added to Quarkus in the future.

- Using `java.util.concurrent.Future` and classes that extend it as return types of repository methods.

- Native and named queries when using `@Query`

- Entity State-detection Strategies via `EntityInformation`.

  - As of Quarkus 1.6.0, only "Version-Property and Id-Property inspection" is implemented (which should cover most cases).

  - As of Quarkus 1.7.0, `org.springframework.data.domain.Persistable` is also implemented.

The Quarkus team is exploring various alternatives to bridging the gap between the JPA and Reactive worlds.

Important Technical Note

Please note that the Spring support in Quarkus does not start a Spring Application Context nor are any Spring infrastructure classes run. Spring classes and annotations are only used for reading metadata and / or are used as user code method return types or parameter types.

More Spring guides

Quarkus has more Spring compatibility features. See the following guides for more details:

* Quarkus - Extension for Spring DI

* Quarkus - Extension for Spring Web

* Quarkus - Extension for Spring Security

* Quarkus - Reading properties from Spring Cloud Config Server

* Quarkus - Extension for Spring Boot properties

* Quarkus - Extension for Spring Cache

* Quarkus - Extension for Spring Scheduled