Quarkus - Generating JAX-RS resources with Panache

A lot of web applications are monotonous CRUD applications with REST APIs that are tedious to write. To streamline this task, REST Data with Panache extension can generate the basic CRUD endpoints for your entities and repositories.

While this extension is still experimental and provides a limited feature set, we hope to get an early feedback for it. Currently this extension supports Hibernate ORM with Panache and can generate CRUD resources that work with `application/json` and `application/hal+json` content.

> This technology is considered experimental.
>
> In *experimental* mode, early feedback is requested to mature the idea. There is no guarantee of stability nor long term presence in the platform until the solution matures. 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.

Setting up REST Data with Panache

To begin with:

- Add the required dependencies to your `pom.xml`
  - Hibernate ORM REST Data with Panache extension (`quarkus-hibernate-orm-rest-data-panache`)
  - A JDBC driver extension (`quarkus-jdbc-postgresql`, `quarkus-jdbc-h2`, `quarkus-jdbc-mariadb`, …)
  - One of the RESTEasy JSON serialization extensions (`quarkus-resteasy-jackson` or `quarkus-resteasy-jsonb`)

```
<dependencies>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-hibernate-orm-rest-data-
panache</artifactId>
    </dependency>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-jdbc-postgresql</artifactId>
    </dependency>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-resteasy-jackson</artifactId>
    </dependency>
</dependencies>
```

- Implement the Panache entities and/or repositories as explained in the Hibernate ORM with Panache guide.

- Define the interfaces for generation as explained in the sections below.

Generating resources

REST Data with Panache generates JAX-RS resources based on the interfaces available in your application. For each entity and repository that you want to generate, provide an interface that extends either `PanacheEntityResource` or `PanacheRepositoryResource` interface. *Do not implement these interfaces and don't provide custom methods because they will be ignored.* You can, however, override the methods from the extended interface in order to customize them (see the section at the end).

PanacheEntityResource

If your application has an entity (e.g. `Person`) that extends either `PanacheEntity` or `PanacheEntityBase` class, you could instruct REST Data with Panache to generate its JAX-RS resource with the following interface:

```
public interface PeopleResource extends
PanacheEntityResource<Person, Long> {
}
```

PanacheRepositoryResource

If your application has a simple entity (e.g. `Person`) and a repository (e.g. `PersonRepository`) that implements either `PanacheRepository` or `PanacheRepositoryBase` interface, you could instruct REST Data with Panache to generate its JAX-RS resource with the following interface:

```
public interface PeopleResource extends
PanacheRepositoryResource<PersonRepository, Person, Long> {
}
```

The generated resource

The generated resources will be functionally equivalent for both entities and repositories. The only difference being the particular data access pattern in use.

If you have defined one of the `PeopleResource` interfaces mentioned above, REST Data with Panache will generate a JAX-RS resource similar to this:

```
@Path("/people") // Default path is a hyphenated lowercase resource
name without a suffix of `resource` or `controller`.
public class PeopleResourceImpl implements PeopleResource { // The
actual class name is going to be unique
    @GET
    @Produces("application/json")
    @Path("{id}")
    public Person get(@PathParam("id") Long id){
        // ...
    }

    @GET
    @Produces("application/json")
    public Response list(){
        // ...
    }

    @Transactional
    @POST
    @Consumes("application/json")
    @Produces("application/json")
    public Response add(Person entity) {
        // ..
    }

    @Transactional
    @PUT
    @Consumes("application/json")
    @Produces("application/json")
    @Path("{id}")
    public Response update(@PathParam("id") Long id, Person person)
{
        // ..
    }

    @Transactional
    @DELETE
    @Path("{id}")
    public void delete(@PathParam("id") Long id) {
        // ..
    }
}
```

Resource customisation

REST Data with Panache provides a `@ResourceProperties` and `@MethodProperties` annotations that can be used to customize certain features of the resource. It can be used in your

resource interface:

```java
@ResourceProperties(hal = true, path = "my-people")
public interface PeopleResource extends
PanacheEntityResource<Person, Long> {
    @MethodProperties(path = "all")
    Response list();

    @MethodProperties(exposed = false)
    void delete(Long id);
}
```

Available options

@ResourceProperties

- hal - in addition to the standard application/json responses, generates additional methods that can return application/hal+json responses if requested via an Accept header. Default is false.

- path - resource base path. Default path is a hyphenated lowercase resource name without a suffix of resource or controller.

- paged - whether collection responses should be paged or not. First, last, previous and next page URIs are included in the response headers if they exist. Request page index and size are taken from the page and size query parameters that default to 0 and 20 respectively. Default is true.

@MethodProperties

- exposed - does not expose a particular HTTP verb when set to false. Default is true.

- path - operation path (this is appended to the resource base path). Default is an empty string.

Response body examples

As mentioned above REST Data with Panache supports the application/json and application/hal+json response content types. Here are a couple of examples of how a response body would look like for the get and list operations assuming there are five Person records in a database.

GET /people/1

Accept: application/json

```
{
  "id": 1,
  "name": "John Johnson",
  "birth": "1988-01-10"
}
```

Accept: application/hal+json

```
{
  "id": 1,
  "name": "John Johnson",
  "birth": "1988-01-10",
  "_links": {
    "self": "http://example.com/people/1",
    "remove": "http://example.com/people/1",
    "update": "http://example.com/people/1",
    "add": "http://example.com/people",
    "list": "http://example.com/people"
  }
}
```

GET /people?page=0&size=2

Accept: application/json

```
[
  {
    "id": 1,
    "name": "John Johnson",
    "birth": "1988-01-10"
  },
  {
    "id": 2,
    "name": "Peter Peterson",
    "birth": "1986-11-20"
  }
]
```

Accept: application/hal+json

```json
{
  "_embedded": [
    {
      "id": 1,
      "name": "John Johnson",
      "birth": "1988-01-10",
      "_links": {
        "self": "http://example.com/people/1",
        "remove": "http://example.com/people/1",
        "update": "http://example.com/people/1",
        "add": "http://example.com/people",
        "list": "http://example.com/people"
      }
    },
    {
      "id": 2,
      "name": "Peter Peterson",
      "birth": "1986-11-20",
      "_links": {
        "self": "http://example.com/people/2",
        "remove": "http://example.com/people/2",
        "update": "http://example.com/people/2",
        "add": "http://example.com/people",
        "list": "http://example.com/people"
      }
    }
  ],
  "_links": {
    "add": "http://example.com/people",
    "list": "http://example.com/people",
    "first": "http://example.com/people?page=0&size=2",
    "last": "http://example.com/people?page=2&size=2",
    "next": "http://example.com/people?page=1&size=2"
  }
}
```

Both responses would also contain these headers:

- Link: < http://example.com/people?page=0&size=2 >; rel="first"

- Link: < http://example.com/people?page=2&size=2 >; rel="last"

- Link: < http://example.com/people?page=1&size=2 >; rel="next"

A `previous` link header (and hal link) would not be included, because the previous page does not exist.