Simplified Hibernate ORM with Panache

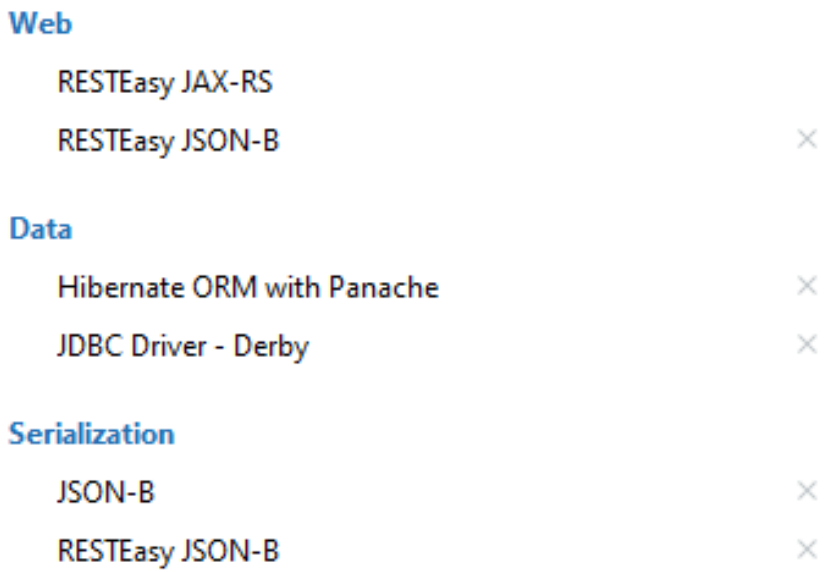
Was ist Panache?

- Hibernate ORM ist die am meisten verbreitetste JPA Implementierung
- Erlaubt detailliert konfigurierbares und komplexes OR-Mapping
- Allerdings nicht wirklich einfach und trivial bei Standardfällen

→ ***Panache*** hilft dabei, Entities und Zugriffe einfach und möglichst ohne Boilerplate-Code zu erstellen

Konfiguration

- Projekt-Konfiguration:



- Konfiguration der Data-Source (application.properties)

```
# configure your datasource
quarkus.datasource.db-kind = derby
quarkus.datasource.username = app
quarkus.datasource.password = app
quarkus.datasource.jdbc.url = jdbc:derby://localhost:1527/myDB;create=true

# drop and create the database at startup (use `update` to only update the schema)
quarkus.hibernate-orm.database.generation = drop-and-create
```

Zugriff mittels klassischem JPA

- Entity-Klasse erstellen

```
@Entity
public class Person {
    @Id
    private Long svnr;

    private String firstname;
    private String lastname;

    public Person() {}

    public Person(Long svnr, String firstname, String lastname) {
        this.svnr = svnr;
        this.firstname = firstname;
        this.lastname = lastname;
    }

    // Getter und Setter
    …
}
```

Zugriff mittels klassischem JPA

- Zugriff über EntityManager

```
@Path("/person")
public class ExampleResource {

    @Inject
    EntityManager em;

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Person getPerson() {
        return em.find(Person.class, 1L);
    }


    @GET
    @Path("init")
    @Transactional
    public String init() {
        Person p = new Person(1L, "Max", "Muster");
        em.persist(p);

        return "init ok";
    }
}
```

Zugriff mittels klassischem JPA

Nachteile / Boilerplate-Code:

- Felder *private*
- generierte *Getter/Setter* blasen den Code unnötig auf
- Methoden für Standard-Zugriffe werden immer wieder neu implementiert (zB Count, …)

→ Panache versucht das zu vereinfachen!

2 Patterns:

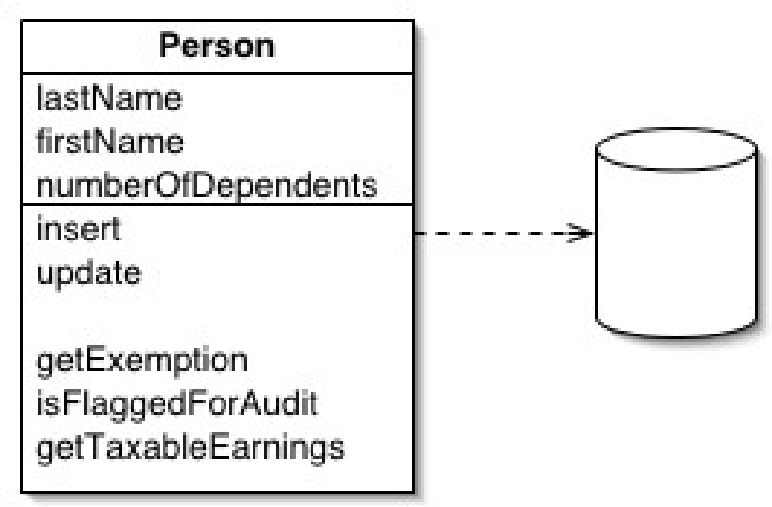
- Active Record-Pattern
- Repository-Pattern

Active Record - Pattern

Beschreibung Active Record-Pattern

„An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data.“

(siehe Martin Fowler, Patterns of Enterprise Application Architecture)



Die Objekte enthalten also Daten und Verhalten!

Umsetzung

Entity-Klasse erstellen:

```
@Entity
public class Person extends PanacheEntity {
    public String name;
    public String firstname;
    public String lastname;

    public String getLastname() {
        return this.lastname.toUpperCase();
    }
}
```

PanacheEntity vergibt einen **Surrogatschlüssel** (id)! Ansonsten PanacheEntityBase verwenden...

- Felder sind *public*, keine *Getter/Setter* notwendig
- Getter/Setter können aber optional erstellt werden (für Validierungen, Umwandlungen) → werden auch **bei Feldzugriff automatisch genutzt**
- @Transient benutzen, um Felder nicht zu persistieren

Umsetzung

Einfacher Zugriff über vordefinierte Methoden, kein expliziter EntityManager notwendig:

```
Person p = new Person();
p.firstname = "Max";
p.lastname = "Muster";
p.persist();

Person p2 = Person.findById(1L);

List<Person> personList = Person.listAll();

personList = Person.list("lastname", ln);
long personCount = Person.count("lastname", ln));

Person.update("firstname='Susanne' where firstname='Susi'");
```

Custom-Queries werden dem Pattern entsprechend als *static*-Methoden in der Entity-Klasse implementiert.

Umsetzung

Für die List-Methoden können auch Streams verwendet werden:

```
@GET
@Path("listWithoutLastname/{lastname}")
@Produces(MediaType.APPLICATION_JSON)
@Transactional
public List<String> listWithoutLastname(
                                @PathParam("lastname") String lastname) {
    try (Stream<Person> persons = Person.streamAll(
                                      Sort.ascending("lastname", "firstname"))) {
        return
           persons
              .filter(person -> !person.lastname.equalsIgnoreCase(lastname))
              .map(person -> person.firstname + " " + person.lastname)
              .collect(Collectors.toList());
    }
}
```

Stream-Methoden benötigen Transaktionen!
Da sie I/O´s durchführen, sollten sie geschlossen werden um das darunterliegende ResultSet zu schliesen (try-with-resource)!

Repository - Pattern

Beschreibung Repository-Pattern

„Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects.“

(siehe Martin Fowler, Patterns of Enterprise Application Architecture)

Umsetzung

Entity-Klasse erstellen, je nach Bedarf:

- Klassische JPA-Entities oder
- Extend PanacheEntityBase (keine Getter/Setter) od.
- Extend PanacheEntity (default ID)

```
@Entity
public class Student {
    @Id
    @GeneratedValue
    private Long id;

    private String name;
    private LocalDate birth;

    public Student() { }

    public Student(String name, LocalDate birth) {
        this.name = name;    this.birth = birth;
    }

    //Getter and Setter
}
```

Umsetzung

Repository-Klasse anlegen und PanacheRepository implementieren (od. PanacheRepositoryBase für eigene IDs):

```
@ApplicationScoped
public class StudentRepository implements PanacheRepository<Student> {

    public Student findByName(String name) {
        return find("name", name).firstResult();
    }

}
```

Alle in PanacheEntityBase implementierten Methoden stehen sofort zur Verfügung (vgl. Active Record-Pattern)! Auch Streams können wiederum verwendet werden!

Eigene Methoden werden als Instanzmethoden im Repository implementiert.

Umsetzung

@Inject der erstellten Repository-Klasse für den Zugriff:

```
@Path("/student")
public class StudentResource {

    @Inject
    StudentRepository studentRepo;

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Student getStudent() {
        return studentRepo.findById(1L);
    }

    @GET
    @Path("list")
    @Produces(MediaType.APPLICATION_JSON)
    public List<Student> allPerson() {
        return studentRepo.listAll();
    }

    …
}
```

Advanced Query

Paging

```
// create a query for all living persons
PanacheQuery<Person> livingPersons = Person.find("status", Status.Alive);

// make it use pages of 25 entries at a time
livingPersons.page(Page.ofSize(25));

// get the first page
List<Person> firstPage = livingPersons.list();
// get the second page
List<Person> secondPage = livingPersons.nextPage().list();
// get page 7
List<Person> page7 = livingPersons.page(Page.of(7, 25)).list();

// get the number of pages
int numberOfPages = livingPersons.pageCount();

// get the total number of entities returned by this query without paging
long count = livingPersons.count();

// and you can chain methods of course

return Person.find("status", Status.Alive)
        .page(Page.ofSize(25))
        .nextPage()
        .stream()   ….
```

Ranges anstatt Paging

```
// create a query for all living persons
PanacheQuery<Person> livingPersons = Person.find("status", Status.Alive);

// make it use a range: start at index 0 until index 24 (inclusive).
livingPersons.range(0, 24);

// get the range
List<Person> firstRange = livingPersons.list();

// to get the next range, you need to call range again
List<Person> secondRange = livingPersons.range(25, 49).list();
```

Sortierung, Named Queries

Sortierung:

```
List<Person> persons = Person.list("order by name,birth");
List<Person> students = Person.listAll(Sort.by("name").and("birth"));
```

NamedQueries: Zugriff über #<*Queryname*>

```
@Entity
@NamedQuery(name = "Person.getByName", query = "from Person where lastname = ?1")
public class Person extends PanacheEntity { … }

return Person.list("#Person.getByName", ln);
```

Weitere Möglichkeiten siehe:
https://quarkus.io/guides/hibernate-orm-panache