Quarkus - Using Hibernate ORM and JPA

Hibernate ORM is the de facto standard JPA implementation and offers you the full breadth of an Object Relational Mapper. It works beautifully in Quarkus.

Setting up and configuring Hibernate ORM

When using Hibernate ORM in Quarkus, you don't need to have a `persistence.xml` resource to configure it.

Using such a classic configuration file is an option, but unnecessary unless you have specific advanced needs; so we'll see first how Hibernate ORM can be configured without a `persistence.xml` resource.

In Quarkus, you just need to:

- add your configuration settings in `application.properties`

- annotate your entities with `@Entity` and any other mapping annotation as usual

Other configuration needs have been automated: Quarkus will make some opinionated choices and educated guesses.

Add the following dependencies to your project:

- the Hibernate ORM extension: `io.quarkus:quarkus-hibernate-orm`

- your JDBC driver extension; the following options are available:

    ◦ `quarkus-jdbc-db2` for IBM DB2

    ◦ `quarkus-jdbc-derby` for Apache Derby

    ◦ `quarkus-jdbc-h2` for H2

    ◦ `quarkus-jdbc-mariadb` for MariaDB

    ◦ `quarkus-jdbc-mssql` for Microsoft SQL Server

    ◦ `quarkus-jdbc-mysql` for MySQL

    ◦ `quarkus-jdbc-postgresql` for PostgreSQL

Example dependencies using Maven

```xml
<dependencies>
    <!-- Hibernate ORM specific dependencies -->
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-hibernate-orm</artifactId>
    </dependency>

    <!-- JDBC driver dependencies -->
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-jdbc-postgresql</artifactId>
    </dependency>
</dependencies>
```

Annotate your persistent objects with `@Entity`, then add the relevant configuration properties in `application.properties`.

Example `application.properties`

```
# datasource configuration
quarkus.datasource.db-kind = postgresql
quarkus.datasource.username = hibernate
quarkus.datasource.password = hibernate
quarkus.datasource.jdbc.url =
jdbc:postgresql://localhost:5432/hibernate_db

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

Note that these configuration properties are not the same ones as in your typical Hibernate ORM configuration file: these drive Quarkus configuration properties, which often will map to Hibernate configuration properties but could have different names and don't necessarily map 1:1 to each other.

Also, Quarkus will set many Hibernate configuration settings automatically, and will often use more modern defaults.

Please see below section Hibernate ORM configuration properties for the list of properties you can set in `application.properties`.

An `EntityManagerFactory` will be created based on the Quarkus `datasource` configuration as long as the Hibernate ORM extension is listed among your project dependencies.

The dialect will be selected based on the JDBC driver - unless you set one explicitly.

You can then happily inject your `EntityManager`:

Example application bean using Hibernate

```java
@ApplicationScoped
public class SantaClausService {
    @Inject
    EntityManager em;  ①

    @Transactional  ②
    public void createGift(String giftDescription) {
        Gift gift = new Gift();
        gift.setName(giftDescription);
        em.persist(gift);
    }
}
```

① Inject your entity manager and have fun

② Mark your CDI bean method as `@Transactional` and the `EntityManager` will enlist and flush at commit.

Example Entity

```java
@Entity
public class Gift {
    private Long id;
    private String name;

    @Id @GeneratedValue(strategy = GenerationType.SEQUENCE,
generator="giftSeq")
    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;
    }
}
```

To load some SQL statements when Hibernate ORM starts, add an `import.sql` in the root of your resources directory. Such a script can contain any SQL DML statements; make sure to terminate each statement with a semicolon. This is useful to have a data set ready for your tests or demos.

> ⚠️ Make sure to wrap methods modifying your database (e.g. `entity.persist()` within a transaction. Marking a CDI bean method `@Transactional` will do that for you and make that method a transaction boundary. We recommend doing so at your application entry point boundaries like your REST endpoint controllers.

Hibernate ORM configuration properties

There are various optional properties useful to refine your `EntityManagerFactory` or guide guesses of Quarkus.

There are no required properties, as long as a default datasource is configured.

When no property is set, Quarkus can typically infer everything it needs to setup Hibernate ORM and will have it use the default datasource.

The configuration properties listed here allow you to override such defaults, and customize and tune various aspects.

🔒 Configuration property fixed at build time - All other configuration properties are overridable at runtime

Configuration property	Type	Default
🔒 `quarkus.hibernate-orm.dialect` Class name of the Hibernate ORM dialect. The complete list of bundled dialects is available in the Hibernate ORM JavaDoc. > ℹ️ Not all the dialects are supported in GraalVM native executables: we currently provide driver extensions for PostgreSQL, MariaDB, Microsoft SQL Server and H2.	string	
🔒 `quarkus.hibernate-orm.dialect.storage-engine` The storage engine to use when the dialect supports multiple storage engines. E.g. `MyISAM` or `InnoDB` for MySQL.	string	

🔒 `quarkus.hibernate-orm.sql-load-script` Name of the file containing the SQL statements to execute when Hibernate ORM starts. Its default value differs depending on the Quarkus launch mode: • In dev and test modes, it defaults to `import.sql`. Simply add an `import.sql` file in the root of your resources directory and it will be picked up without having to set this property. Pass `no-file` to force Hibernate ORM to ignore the SQL import file. • In production mode, it defaults to `no-file`. It means Hibernate ORM won't try to execute any SQL import file by default. Pass an explicit value to force Hibernate ORM to execute the SQL import file. If you need different SQL statements between dev mode, test (`@QuarkusTest`) and in production, use Quarkus [configuration profiles facility](#). *application.properties* ```\n%dev.quarkus.hibernate-orm.sql-load-script = import-\ndev.sql\n%test.quarkus.hibernate-orm.sql-load-script =\nimport-test.sql\n%prod.quarkus.hibernate-orm.sql-load-script = no-\nfile\n``` ℹ️ Quarkus supports `.sql` file with SQL statements or comments spread over multiple lines. Each SQL statement must be terminated by a semicolon.	string	`import.sql in DEV, TEST ; no-file otherwise`
🔒 `quarkus.hibernate-orm.batch-fetch-size` The size of the batches used when loading entities and collections. `-1` means batch loading is disabled. This is the default.	int	`-1`
🔒 `quarkus.hibernate-orm.physical-naming-strategy` Pluggable strategy contract for applying physical naming rules for database object names. Class name of the Hibernate PhysicalNamingStrategy implementation	string	
🔒 `quarkus.hibernate-orm.implicit-naming-strategy` Pluggable strategy for applying implicit naming rules when an explicit name is not given. Class name of the Hibernate ImplicitNamingStrategy implementation	string	

	Type	Default
🔒 `quarkus.hibernate-orm.multitenant` Defines the method for multi-tenancy (DATABASE, NONE, SCHEMA). The complete list of allowed values is available in the Hibernate ORM JavaDoc. The type DISCRIMINATOR is currently not supported. The default value is NONE (no multi-tenancy).	string	
🔒 `quarkus.hibernate-orm.multitenant-schema-datasource` Defines the name of the data source to use in case of SCHEMA approach. The default data source will be used if not set.	string	
🔒 `quarkus.hibernate-orm.statistics` Whether statistics collection is enabled. If 'metrics.enabled' is true, then the default here is considered true, otherwise the default is false.	boolean	
🔒 `quarkus.hibernate-orm.metrics.enabled` Whether or not metrics are published if a metrics extension is enabled.	boolean	`false`
🔒 `quarkus.hibernate-orm.second-level-caching-enabled` The default in Quarkus is for 2nd level caching to be enabled, and a good implementation is already integrated for you. Just cherry-pick which entities should be using the cache. Set this to false to disable all 2nd level caches.	boolean	`true`
Query related configuration	**Type**	**Default**
🔒 `quarkus.hibernate-orm.query.query-plan-cache-max-size` The maximum size of the query plan cache.	string	
🔒 `quarkus.hibernate-orm.query.default-null-ordering` Default precedence of null values in `ORDER BY` clauses. Valid values are: `none`, `first`, `last`.	string	
Database related configuration	**Type**	**Default**
🔒 `quarkus.hibernate-orm.database.generation` Select whether the database schema is generated or not. `drop-and-create` is awesome in development mode. Accepted values: `none`, `create`, `drop-and-create`, `drop`, `update`.	string	`none`

	Type	Default
🔒 `quarkus.hibernate-orm.database.generation.halt-on-error` Whether we should stop on the first error when applying the schema.	boolean	`false`
🔒 `quarkus.hibernate-orm.database.default-catalog` The default catalog to use for the database objects.	string	
🔒 `quarkus.hibernate-orm.database.default-schema` The default schema to use for the database objects.	string	
🔒 `quarkus.hibernate-orm.database.charset` The charset of the database. Used for DDL generation and also for the SQL import scripts.	Charset	`UTF-8`
🔒 `quarkus.hibernate-orm.database.globally-quoted-identifiers` Whether Hibernate should quote all identifiers.	boolean	`false`
JDBC related configuration	**Type**	**Default**
🔒 `quarkus.hibernate-orm.jdbc.timezone` The time zone pushed to the JDBC driver.	string	
🔒 `quarkus.hibernate-orm.jdbc.statement-fetch-size` How many rows are fetched at a time by the JDBC driver.	int	
🔒 `quarkus.hibernate-orm.jdbc.statement-batch-size` The number of updates (inserts, updates and deletes) that are sent by the JDBC driver at one time for execution.	int	
Logging configuration	**Type**	**Default**
🔒 `quarkus.hibernate-orm.log.sql` Show SQL logs and format them nicely. Setting it to true is obviously not recommended in production.	boolean	`false`
🔒 `quarkus.hibernate-orm.log.bind-param` Logs SQL bind parameter. Setting it to true is obviously not recommended in production.	boolean	`false`

	Type	Default
🔒 `quarkus.hibernate-orm.log.jdbc-warnings` Whether JDBC warnings should be collected and logged.	boolean	depends on dialect
Caching configuration	**Type**	**Default**
🔒 `quarkus.hibernate-orm.cache."cache".expiration.max-idle` The maximum time before an object of the cache is considered expired.	Duration ❓	
🔒 `quarkus.hibernate-orm.cache."cache".memory.object-count` The maximum number of objects kept in memory in the cache.	long	

ℹ️ *About the Duration format*

The format for durations uses the standard `java.time.Duration` format. You can learn more about it in the Duration#parse() javadoc.

You can also provide duration values starting with a number. In this case, if the value consists only of a number, the converter treats the value as seconds. Otherwise, PT is implicitly prepended to the value to obtain a standard `java.time.Duration` format.

ℹ️ Do not mix `persistence.xml` and `quarkus.hibernate-orm.*` properties in `application.properties`. Quarkus will raise an exception. Make up your mind on which approach you want to use.

💡 Want to start a PostgreSQL server on the side with Docker?

```
docker run --ulimit memlock=-1:-1 -it --rm=true
--memory-swappiness=0 \
        --name postgres-quarkus-hibernate -e
POSTGRES_USER=hibernate \
        -e POSTGRES_PASSWORD=hibernate -e
POSTGRES_DB=hibernate_db \
        -p 5432:5432 postgres:10.5
```

This will start a non-durable empty database: ideal for a quick experiment!

Setting up and configuring Hibernate ORM with a `persistence.xml`

Alternatively, you can use a `META-INF/persistence.xml` to setup Hibernate ORM. This is useful for:

- migrating existing code

- when you have relatively complex settings requiring the full flexibility of the configuration

- or if you like it the good old way

> If you have a `persistence.xml`, then you cannot use the `quarkus.hibernate-orm.*` properties and only persistence units defined in `persistence.xml` will be taken into account.

Your `pom.xml` dependencies as well as your Java code would be identical to the precedent example. The only difference is that you would specify your Hibernate ORM configuration in `META-INF/persistence.xml`:

Example persistence.xml resource

```xml
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence

http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
             version="2.1">

    <persistence-unit name="CustomerPU" transaction-type="JTA">

        <description>My customer entities</description>

        <properties>
            <!-- Connection specific -->
            <property name="hibernate.dialect"
value="org.hibernate.dialect.PostgreSQL95Dialect"/>

            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>

            <!--
                Optimistically create the tables;
                will cause background errors being logged if they
already exist,
                but is practical to retain existing data across
runs (or create as needed) -->
            <property name="javax.persistence.schema-
generation.database.action" value="drop-and-create"/>

            <property name="javax.persistence.validation.mode"
value="NONE"/>
        </properties>

    </persistence-unit>
</persistence>
```

When using the `persistence.xml` configuration you are configuring Hibernate ORM directly, so in this case the appropriate reference is the documentation on hibernate.org.

Please remember these are not the same property names as the ones used in the Quarkus `application.properties`, nor will the same defaults be applied.

Defining entities in external projects or jars

Hibernate ORM in Quarkus relies on compile-time bytecode enhancements to your entities. If you define your entities in the same project where you build your Quarkus application, everything will work

fine.

If the entities come from external projects or jars, you can make sure that your jar is treated like a Quarkus application library by adding an empty `META-INF/beans.xml` file.

This will allow Quarkus to index and enhance your entities as if they were inside the current project.

Hibernate ORM in development mode

Quarkus development mode is really useful for applications that mix front end or services and database access.

There are a few common approaches to make the best of it.

The first choice is to use `quarkus.hibernate-orm.database.generation=drop-and-create` in conjunction with `import.sql`.

That way for every change to your app and in particular to your entities, the database schema will be properly recreated and your data fixture (stored in `import.sql`) will be used to repopulate it from scratch. This is best to perfectly control your environment and works magic with Quarkus live reload mode: your entity changes or any change to your `import.sql` is immediately picked up and the schema updated without restarting the application!

> 💡 By default in `dev` and `test` modes, Hibernate ORM, upon boot, will read and execute the SQL statements in the `/import.sql` file (if present). You can change the file name by changing the property `quarkus.hibernate-orm.sql-load-script` in `application.properties`.

The second approach is to use `quarkus.hibernate-orm.database.generation=update`. This approach is best when you do many entity changes but still need to work on a copy of the production data or if you want to reproduce a bug that is based on specific database entries. `update` is a best effort from Hibernate ORM and will fail in specific situations including altering your database structure which could lead to data loss. For example if you change structures which violate a foreign key constraint, Hibernate ORM might have to bail out. But for development, these limitations are acceptable.

The third approach is to use `quarkus.hibernate-orm.database.generation=none`. This approach is best when you are working on a copy of the production data but want to fully control the schema evolution. Or if you use a database schema migration tool like Flyway.

With this approach when making changes to an entity, make sure to adapt the database schema accordingly; you could also use `validate` to have Hibernate verify the schema matches its expectations.

> ⚠️ Do not use `quarkus.hibernate-orm.database.generation drop-and-create` and `update` in your production environment.

These approaches become really powerful when combined with Quarkus configuration profiles. You can define different configuration profiles to select different behaviors depending on your environment. This is great because you can define different combinations of Hibernate ORM

properties matching the development style you currently need.

application.properties

```
%dev.quarkus.hibernate-orm.database.generation = drop-and-create
%dev.quarkus.hibernate-orm.sql-load-script = import-dev.sql

%dev-with-data.quarkus.hibernate-orm.database.generation = update
%dev-with-data.quarkus.hibernate-orm.sql-load-script = no-file

%prod.quarkus.hibernate-orm.database.generation = none
%prod.quarkus.hibernate-orm.sql-load-script = no-file
```

Start "dev mode" using a custom profile via Maven

```
./mvnw compile quarkus:dev -Dquarkus.profile=dev-with-data
```

Hibernate ORM in production mode

Quarkus comes with default profiles (dev, test and prod). And you can add your own custom profiles to describe various environments (staging, prod-us, etc).

The Hibernate ORM Quarkus extension sets some default configurations differently in dev and test modes than in other environments.

- quarkus.hibernate-orm.sql-load-script is set to no-file for all profiles except the dev and test ones.

You can override it in your application.properties explicitly (e.g. %prod.quarkus.hibernate-orm.sql-load-script = import.sql) but we wanted you to avoid overriding your database by accident in prod :)

Speaking of, make sure to not drop your database schema in production! Add the following in your properties file.

application.properties

```
%prod.quarkus.hibernate-orm.database.generation = none
%prod.quarkus.hibernate-orm.sql-load-script = no-file
```

Caching

Applications that frequently read the same entities can see their performance improved when the Hibernate ORM second-level cache is enabled.

Caching of entities

To enable second-level cache, mark the entities that you want cached with
@javax.persistence.Cacheable:

```
@Entity
@Cacheable
public class Country {
    int dialInCode;
    // ...
}
```

When an entity is annotated with @Cacheable, all its field values are cached except for collections and relations to other entities.

This means the entity can be loaded without querying the database, but be careful as it implies the loaded entity might not reflect recent changes in the database.

Caching of collections and relations

Collections and relations need to be individually annotated to be cached; in this case the Hibernate specific @org.hibernate.annotations.Cache should be used, which requires also to specify the CacheConcurrencyStrategy:

```
package org.acme;

@Entity
@Cacheable
public class Country {
    // ...

    @OneToMany
    @Cache(usage = CacheConcurrencyStrategy.READ_ONLY)
    List<City> cities;

    // ...
}
```

Caching of queries

Queries can also benefit from second-level caching. Cached query results can be returned immediately to the caller, avoiding to run the query on the database.

Be careful as this implies the results might not reflect recent changes.

To cache a query, mark it as cacheable on the Query instance:

```
Query query = ...
query.setHint("org.hibernate.cacheable", Boolean.TRUE);
```

If you have a NamedQuery then you can enable caching directly on its definition, which will usually be on an entity:

```
@Entity
@NamedQuery(name = "Fruits.findAll",
        query = "SELECT f FROM Fruit f ORDER BY f.name",
        hints = @QueryHint(name = "org.hibernate.cacheable", value =
"true") )
public class Fruit {
    ...
```

That's all! Caching technology is already integrated and enabled by default in Quarkus, so it's enough to set which ones are safe to be cached.

Tuning of Cache Regions

Caches store the data in separate regions to isolate different portions of data; such regions are assigned a name, which is useful for configuring each region independently, or to monitor their statistics.

By default entities are cached in regions named after their fully qualified name, e.g. org.acme.Country.

Collections are cached in regions named after the fully qualified name of their owner entity and collection field name, separated by # character, e.g. org.acme.Country#cities.

All cached queries are by default kept in a single region dedicated to them called default-query-results-region.

All regions are bounded by size and time by default. The defaults are 10000 max entries, and 100 seconds as maximum idle time.

The size of each region can be customized via the quarkus.hibernate-orm.cache."<region_name>".memory.object-count property (Replace *region_name* with the actual region name).

To set the maximum idle time, provide the duration (see note on duration's format below) via the quarkus.hibernate-orm.cache."<region_name>".expiration.max-idle property (Replace *region_name* with the actual region name).

The double quotes are mandatory if your region name contains a dot. For instance:

```
quarkus.hibernate-
orm.cache."org.acme.MyEntity".memory.object-count=1000
```

The format for durations uses the standard `java.time.Duration` format. You can learn more about it in the Duration#parse() javadoc.

You can also provide duration values starting with a number. In this case, if the value consists only of a number, the converter treats the value as seconds. Otherwise, PT is implicitly prepended to the value to obtain a standard `java.time.Duration` format.

Limitations of Caching

The caching technology provided within Quarkus is currently quite rudimentary and limited.

The team thought it was better to have *some* caching capability to start with, than having nothing; you can expect better caching solution to be integrated in future releases, and any help and feedback in this area is very welcome.

These caches are kept locally, so they are not invalidated or updated when changes are made to the persistent store by other applications.

Also, when running multiple copies of the same application (in a cluster, for example on Kubernetes/OpenShift), caches in separate copies of the application aren't synchronized.

For these reasons, enabling caching is only suitable when certain assumptions can be made: we strongly recommend that only entities, collections and queries which never change are cached. Or at most, that when indeed such an entity is mutated and allowed to be read out of date (stale) this has no impact on the expectations of the application.

Following this advice guarantees applications get the best performance out of the second-level cache and yet avoid unexpected behaviour.

On top of immutable data, in certain contexts it might be acceptable to enable caching also on mutable data; this could be a necessary tradeoff on selected entities which are read frequently and for which some degree of staleness is acceptable; this " acceptable degree of staleness" can be tuned by setting eviction properties. This is however not recommended and should be done with extreme care, as it might produce unexpected and unforeseen effects on the data.

Rather than enabling caching on mutable data, ideally a better solution would be to use a clustered cache; however at this time Quarkus doesn't provide any such implementation: feel free to get in touch and let this need known so that the team can take this into account.

Finally, the second-level cache can be disabled globally by setting `hibernate.cache.use_second_level_cache` to `false`; this is a setting that needs to be specified in the `persistence.xml` configuration file.

When second-level cache is disabled, all cache annotations are ignored and all queries are run ignoring caches; this is generally useful only to diagnose issues.

Hibernate Envers

The Envers extension to Hibernate ORM aims to provide an easy auditing / versioning solution for entity classes.

In Quarkus, Envers has a dedicated Quarkus Extension `io.quarkus:quarkus-hibernate-envers`; you just need to add this to your project to start using it.

Additional dependency to enable Hibernate Envers

```xml
<!-- Add the Hibernate Envers extension -->
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-hibernate-envers</artifactId>
</dependency>
```

At this point the extension does not expose additional configuration properties.

For more information about Hibernate Envers, see hibernate.org/orm/envers/.

Metrics

The SmallRye Metrics extension is capable of exposing metrics that Hibernate ORM collects at runtime. To enable exposure of Hibernate metrics on the `/metrics` endpoint, make sure your project depends on the `quarkus-smallrye-metrics` artifact and set the configuration property `quarkus.hibernate-orm.metrics.enabled` to `true`. Metrics will then be available under the `vendor` scope.

Limitations and other things you should know

Quarkus does not modify the libraries it uses; this rule applies to Hibernate ORM as well: when using this extension you will mostly have the same experience as using the original library.

But while they share the same code, Quarkus does configure some components automatically and injects custom implementations for some extension points; this should be transparent and useful but if you're an expert of Hibernate you might want to know what is being done.

Automatic build time enhancement

Hibernate ORM can use build time enhanced entities; normally this is not mandatory but it's useful and will have your applications perform better.

Typically you would need to adapt your build scripts to include the Hibernate Enhancement plugins; in Quarkus this is not necessary as the enhancement step is integrated in the build and analysis of the Quarkus application.

Automatic integration

Transaction Manager integration

You don't need to set this up, Quarkus automatically injects the reference to the Narayana Transaction Manager. The dependency is included automatically as a transitive dependency of the Hibernate ORM extension. All configuration is optional; for more details see Using Transactions in Quarkus.

Connection pool

Don't need to choose one either. Quarkus automatically includes the Agroal connection pool; just configure your datasource as in the above examples and it will setup Hibernate ORM to use Agroal. More details about this connection pool can be found in Quarkus - Datasources.

Second Level Cache

as explained above in section Caching, you don't need to pick an implementation. A suitable implementation based on technologies from Infinispan and Caffeine is included as a transitive dependency of the Hibernate ORM extension, and automatically integrated during the build.

Limitations

XML mapping

Hibernate ORM allows to map entities using XML files; this capability isn't enabled in Quarkus: use annotations instead as Quarkus can handle them very efficiently. This limitation could be lifted in the future, if there's a compelling need for it and if someone contributes it.

JMX

Management beans are not working in GraalVM native images; therefore Hibernate's capability to register statistics and management operations with the JMX bean is disabled when compiling into a native image. This limitation is likely permanent, as it's not a goal for native images to implement support for JMX. All such metrics can be accessed in other ways.

JACC Integration

Hibernate ORM's capability to integrate with JACC is disabled when building GraalVM native images, as JACC is not available - nor useful - in native mode.

Binding the Session to ThreadLocal context

Essentially using the `ThreadLocalSessionContext` helper of Hibernate ORM is not implemented. The team believes this isn't a big deal as it's trivial to inject the Session via CDI instead, or handling the binding into a ThreadLocal yourself, making this a legacy feature. This limitation might be resolved in the future, if someone opens a ticket for it and provides a reasonable use case to justify the need.

JPA Callbacks

Annotations allowing for application callbacks on entity lifecycle events defined by JPA such as `@javax.persistence.PostUpdate`, `@javax.persistence.PostLoad`, `@javax.persistence.PostPersist`, etc... are currently not processed. This limitation could be resolved in a future version, depending on user demand.

Single instance

It is currently not possible to configure more than one instance of Hibernate ORM. This is a temporary limitation, the team is working on it - please be patient!

Other notable differences

Format of `import.sql`

> When importing a `import.sql` to setup your database, keep in mind that Quarkus reconfigures Hibernate ORM so to require a semicolon (';') to terminate each statement. The default in Hibernate is to have a statement per line, without requiring a terminator other than newline: remember to convert your scripts to use the ';' terminator character if you're reusing existing scripts. This is useful so to allow multi-line statements and human friendly formatting.

Simplifying Hibernate ORM with Panache

The Hibernate ORM with Panache extension facilitates the usage of Hibernate ORM by providing active record style entities (and repositories) and focuses on making your entities trivial and fun to write in Quarkus.

Configure your datasource

Datasource configuration is extremely simple, but is covered in a different guide as technically it's implemented by the Agroal connection pool extension for Quarkus.

Jump over to Quarkus - Datasources for all details.

Multitenancy

"The term multitenancy, in general, is applied to software development to indicate an architecture in which a single running instance of an application simultaneously serves multiple clients (tenants). This is highly common in SaaS solutions. Isolating information (data, customizations, etc.) pertaining to the various tenants is a particular challenge in these systems. This includes the data owned by each tenant stored in the database" (Hibernate User Guide).

Quarkus currently supports the separate database and the separate schema approach.

Writing the application

Let's start by implementing the `/{tenant}` endpoint. As you can see from the source code below it is just a regular JAX-RS resource:

```java
import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;
import javax.persistence.EntityManager;
import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

@ApplicationScoped
@Produces("application/json")
@Consumes("application/json")
@Path("/{tenant}")
public class FruitResource {

    @Inject
    EntityManager entityManager;

    @GET
    @Path("fruits")
    public Fruit[] getFruits() {
        return entityManager.createNamedQuery("Fruits.findAll",
Fruit.class)
                .getResultList().toArray(new Fruit[0]);
    }

}
```

In order to resolve the tenant from incoming requests and map it to a specific tenant configuration, you need to create an implementation for the `io.quarkus.hibernate.orm.runtime.tenant.TenantResolver` interface.

```java
import javax.enterprise.context.ApplicationScoped;

import io.quarkus.arc.Arc;
import io.quarkus.arc.Unremovable;
import io.quarkus.hibernate.orm.runtime.tenant.TenantResolver;
import io.vertx.ext.web.RoutingContext;

@RequestScoped
@Unremovable
public class CustomTenantResolver implements TenantResolver {

    @Inject
    RoutingContext context;

    @Override
    public String getDefaultTenantId() {
        return "base";
    }

    @Override
    public String resolveTenantId() {
        String path = context.request().path();
        String[] parts = path.split("/");

        if (parts.length == 0) {
            // resolve to default tenant config
            return getDefaultTenantId();
        }

        return parts[1];
    }

}
```

From the implementation above, tenants are resolved from the request path so that in case no tenant could be inferred, the default tenant identifier is returned.

Configuring the application

In general it is not possible to use the Hibernate ORM database generation feature in conjunction with a multitenancy setup. Therefore you have to disable it and you need to make sure that the tables are created per schema. The following setup will use the Flyway extension to achieve this goal.

SCHEMA approach

The same data source will be used for all tenants and a schema has to be created for every tenant inside that data source. CAUTION: Some databases like MariaDB/MySQL do not support database schemas. In these cases you have to use the DATABASE approach below.

```
# Disable generation
quarkus.hibernate-orm.database.generation=none

# Enable SCHEMA approach and use default datasource
quarkus.hibernate-orm.multitenant=SCHEMA
# You could use a non-default datasource by using the following
setting
# quarkus.hibernate-orm.multitenant-schema-datasource=other

# The default data source used for all tenant schemas
quarkus.datasource.db-kind=postgresql
quarkus.datasource.username=quarkus_test
quarkus.datasource.password=quarkus_test
quarkus.datasource.jdbc.url=jdbc:postgresql://localhost:5432/quarku
s_test

# Enable Flyway configuration to create schemas
quarkus.flyway.schemas=base,mycompany
quarkus.flyway.locations=classpath:schema
quarkus.flyway.migrate-at-start=true
```

Here is an example of the Flyway SQL (V1.0.0__create_fruits.sql) to be created in the configured folder src/main/resources/schema.

```sql
CREATE SEQUENCE base.known_fruits_id_seq;
SELECT setval('base."known_fruits_id_seq"', 3);
CREATE TABLE base.known_fruits
(
   id   INT,
   name VARCHAR(40)
);
INSERT INTO base.known_fruits(id, name) VALUES (1, 'Cherry');
INSERT INTO base.known_fruits(id, name) VALUES (2, 'Apple');
INSERT INTO base.known_fruits(id, name) VALUES (3, 'Banana');

CREATE SEQUENCE mycompany.known_fruits_id_seq;
SELECT setval('mycompany."known_fruits_id_seq"', 3);
CREATE TABLE mycompany.known_fruits
(
   id   INT,
   name VARCHAR(40)
);
INSERT INTO mycompany.known_fruits(id, name) VALUES (1, 'Avocado');
INSERT INTO mycompany.known_fruits(id, name) VALUES (2,
'Apricots');
INSERT INTO mycompany.known_fruits(id, name) VALUES (3,
'Blackberries');
```

DATABASE approach

For every tenant you need to create a named data source with the same identifier that is returned by the `TenantResolver`.

```
# Disable generation
quarkus.hibernate-orm.database.generation=none

# Enable DATABASE approach
quarkus.hibernate-orm.multitenant=DATABASE

# Default tenant 'base'
quarkus.datasource.base.db-kind=postgresql
quarkus.datasource.base.username=quarkus_test
quarkus.datasource.base.password=quarkus_test
quarkus.datasource.base.jdbc.url=jdbc:postgresql://localhost:5432/q
uarkus_test

# Tenant 'mycompany'
quarkus.datasource.mycompany.db-kind=postgresql
quarkus.datasource.mycompany.username=mycompany
quarkus.datasource.mycompany.password=mycompany
quarkus.datasource.mycompany.jdbc.url=jdbc:postgresql://localhost:5
433/mycompany

# Flyway configuration for the default datasource
quarkus.flyway.locations=classpath:database/default
quarkus.flyway.migrate-at-start=true

# Flyway configuration for the mycompany datasource
quarkus.flyway.mycompany.locations=classpath:database/mycompany
quarkus.flyway.mycompany.migrate-at-start=true
```

Following are examples of the Flyway SQL files to be created in the configured folder `src/main/resources/database`.

Default schema (`src/main/resources/database/default/V1.0.0__create_fruits.sql`):

```
CREATE SEQUENCE known_fruits_id_seq;
SELECT setval('known_fruits_id_seq', 3);
CREATE TABLE known_fruits
(
  id   INT,
  name VARCHAR(40)
);
INSERT INTO known_fruits(id, name) VALUES (1, 'Cherry');
INSERT INTO known_fruits(id, name) VALUES (2, 'Apple');
INSERT INTO known_fruits(id, name) VALUES (3, 'Banana');
```

Mycompany   schema
(`src/main/resources/database/mycompany/V1.0.0__create_fruits.sql`):

```
CREATE SEQUENCE known_fruits_id_seq;
SELECT setval('known_fruits_id_seq', 3);
CREATE TABLE known_fruits
(
   id   INT,
   name VARCHAR(40)
);
INSERT INTO known_fruits(id, name) VALUES (1, 'Avocado');
INSERT INTO known_fruits(id, name) VALUES (2, 'Apricots');
INSERT INTO known_fruits(id, name) VALUES (3, 'Blackberries');
```

Programmatically Resolving Tenants Connections

If you need a more dynamic configuration for the different tenants you want to support and don't want to end up with multiple entries in your configuration file, you can use the `io.quarkus.hibernate.orm.runtime.tenant.TenantConnectionResolver` interface to implement your own logic for retrieving a connection. Creating an application scoped bean that implements this interface will replace the current Quarkus default implementation `io.quarkus.hibernate.orm.runtime.tenant.DataSourceTenantConnectionResolver`. Your custom connection resolver would allow for example to read tenant information from a database and create a connection per tenant at runtime based on it.