ORACLE CODE ONE

Break New Ground

San Francisco
September 16–19, 2019

DEV5376
Helidon MicroProfile: Managing Persistence with JPA

Laird Nelson

Consulting Member of Technical Staff
Oracle
September 19, 2019

Safe Harbor

Resources

- **https://github.com/ljnelson/dev5376** (This talk and its code)
- https://helidon.io (Helidon)
- https://github.com/oracle/helidon (Helidon source code)
- https://jakarta.ee/specifications/persistence/2.2/ (JPA)
- https://jakarta.ee/specifications/cdi/2.0/ (CDI)
- https://microprofile.io/ (MicroProfile)
- https://lairdnelson.wordpress.com/ (My blog)

Overview

- What is MicroProfile?
- What is Helidon MicroProfile?
- What is JPA?
- How might JPA fit into a MicroProfile application?
- Helidon MicroProfile meets JPA
- Sample code and demonstration

What is MicroProfile?

- **"Just enough" specification for Java microservices**
- Umbrella specification of Java microservices-focused component specifications
- Inspired by Java EE, but not part of Java EE or Jakarta EE
  - No reference implementations, just specifications and compatibility tests
  - Cloud-native in focus: think health checks, metrics, fault tolerance, etc.
- Borrows familiar Java EE and Jakarta EE component specifications and adds new ones
- **CDI 2.0, JAX-RS 2.1 and MicroProfile Config form the backbone specifications**
- Everything else bolts on, mostly via CDI portable extensions

What is Helidon MicroProfile?

- **Java libraries that form a compliant MicroProfile implementation**
  - Free and open source; Oracle- and community-authored
  - Apache 2.0 license
  - MicroProfile version 3.0 with ongoing adoption of new versions
  - **https://helidon.io/docs/latest/#/microprofile/01_introduction**
  - **https://github.com/oracle/helidon**
- **Not an application server** but Java EE programmers will feel at home

What is JPA?

- Java Persistence Architecture specification, currently at version 2.2 and now under the stewardship of the Eclipse Foundation
- **Standardizes how Java objects map to databases**
- A pragmatic standard: emerged from Hibernate and TopLink
- Part of Jakarta EE, but implementations also run standalone
- **Not part of MicroProfile**

What is JPA?

- **SE mode**: usable from within any Java SE program
  - …but *you* are in charge of (resource-local) transactions, closing sessions, database connections, thread safety, etc.
- **EE mode**: usable from within a Java EE application server
  - The application server handles JTA transactions and also takes care of everything else, but you need an app server*

**Maybe; stay tuned….*

JPA: SE mode

```
// Create an EntityManagerFactory. It's not thread safe and needs to be closed properly.
EntityManagerFactory emf = Persistence.createEntityManagerFactory("test");
// Create your EntityManager when needed.  It's not thread-safe. Don't create too many.
EntityManager em = emf.createEntityManager();
// Handle transactions, exceptions, etc. Suppress exceptions properly and use finally blocks.
EntityTransaction et = em.getTransaction();
et.begin();
PersistenceException e = null;
try {
  em.persist(someEntity); // Here's the thing you want to do
  et.commit();
} catch (PersistenceException timeToRollback) {
  e = timeToRollback;
  et.rollback();
} finally {
  try {
    em.close(); // …in most situations
  } catch (PersistenceException closeException) {
    if (e == null) {
      e = closeException;
    } else {
      e.addSuppressed(closeException);
    }
    throw e;
  }
}
```

JPA: SE mode

```
<!-- Usually RESOURCE_LOCAL transactions, not JTA. -->
<persistence-unit transaction-type="RESOURCE_LOCAL">

<!-- Pooling or not is now up to the JPA implementation (no mention of DataSource below). -->
<property name="javax.persistence.jdbc.driver" value="org.h2.Driver"/>
<property name="javax.persistence.jdbc.url" value="jdbc:h2:mem:test"/>
<property name="javax.persistence.jdbc.user" value="sa"/>
<property name="javax.persistence.jdbc.password" value=""/>
```

JPA: SE mode

- A bit complicated; with great power comes great responsibility
- *You* manage:
  - lifecycles of `EntityManager` and `EntityManagerFactory` instances
  - transactions (via JPA’s `EntityTransaction` interface, not JTA)
  - exceptions (rollback)
  - thread safety
- No real integration needed, just do All The Things™ in The Right Order™

JPA: EE mode

```
// Ask for a transaction-scoped EntityManager.
@PersistenceContext private EntityManager em;

// In a transactional EJB method somewhere:
em.persist(someEntity); // Here’s the thing you want to do
```

JPA: EE mode

```
<!-- JTA transactions are "built in". -->
<persistence-unit transaction-type="JTA"…>

<!-- Tell the server the JTA data source name. -->
<jta-data-source>java:comp/env/jdbc/test<!-- (Hmm; JNDI…) --></jta-data-source>
```

JPA: EE mode

- Very simple to use
- Transactions, thread safety, JPA object lifecycles and so on are managed for you
- JNDI implicitly required; `java:comp/env/jdbc` etc.
- Integration follows the inversion-of-control pattern
- Technically requires an application server 🏋️

EE Mode JPA + MicroProfile?

- What if we could enable EE-mode JPA in MicroProfile?
- Could we get automatic transactions in the absence of EJBs?
- How about extended-mode persistence contexts in the absence of stateful session beans?
- Bean validation?
- Can we avoid JNDI?

EE Mode JPA + MicroProfile?

- We’ll need a `DataSource` provider.
- We’ll need a JTA provider.
- We’ll need various tools to eliminate JNDI.
- We need all these things to be aware of each other.
- Helidon MicroProfile extensions to the rescue!

Helidon HikariCP Extension

- A **CDI portable extension that combines the Hikari connection pool with MicroProfile Config**
- `DataSources` injectable by name:

  -
    ```
    @Inject
    @Named("test")
    private DataSource testDataSource;
    ```
- More importantly, **`DataSources` discoverable via lookup without JNDI:**

  -
    ```
    beanManager.getBeans(DataSource.class,
                         NamedLiteral.of("test"));
    ```

Helidon HikariCP Extension

- Configurable via any MicroProfile Config `ConfigSource`
- `META-INF/microprofile-config.properties` example:

```
javax.sql.DataSource.test.dataSourceClassName=\
org.h2.jdbcx.JdbcDataSource

javax.sql.DataSource.test.dataSource.password=

javax.sql.DataSource.test.dataSource.url=\
jdbc:h2:mem:test

javax.sql.DataSource.test.dataSource.user=sa
```

- In this example, `test` is the name of the data source, the teal bits are Hikari properties and keywords, and the green bits are properties of the `org.h2.jdbcx.JdbcDataSource` implementation class itself

Helidon Oracle UCP Extension

- A **CDI portable extension that combines the Oracle Universal Connection Pool with MicroProfile Config**
- `DataSource`s injectable by name:

  -
    ```
    @Inject
    @Named("test")
    private DataSource testDataSource;
    ```
- More importantly, **`DataSource`s discoverable via lookup without JNDI:**

  -
    ```
    beanManager.getBeans(DataSource.class,
                         NamedLiteral.of("test"));
    ```

Helidon Oracle UCP Extension

- Configurable via any MicroProfile Config `ConfigSource`
- `META-INF/microprofile-config.properties` example:

```
javax.sql.DataSource.test.connectionFactoryClassName=\
org.h2.jdbcx.JdbcDataSource

javax.sql.DataSource.test.password=

javax.sql.DataSource.test.URL=\
jdbc:h2:mem:test

javax.sql.DataSource.test.user=sa
```

- In this example, `test` is the name of the data source and the green bits are properties of the `oracle.ucp.jdbc.PoolDataSourceImpl` implementation class itself

Helidon JTA Extension

- A CDI portable extension that **integrates JTA into Helidon MicroProfile**
- Supplies `TransactionManager`, `Transaction`, `TransactionSynchronizationRegistry`, `UserTransaction` as injectable CDI beans (i.e. **no JNDI needed**)
- Implemented behind the scenes by the Narayana transaction engine
- **Supplies `@Transactional` annotation and related interceptors**
- (As a handy side effect, also enables CDI transactional event observers)

Helidon JTA Extension

```
// Annotate a business method on any CDI bean with
// @Transactional. Here’s a JAX-RS resource method:
@GET
@Path("response")
@Produces("text/plain")
@Transactional
public String hello() {
  // You are in a JTA transaction here. Nice!
  return "Hello, yourself.";
}
```

Helidon EclipseLink Extension

- A simple library and CDI portable extension that **makes EclipseLink think it is running in a Java EE application server even though it's not**
- Lets EclipseLink find the current JTA transaction without using JNDI
  - (Remember the Helidon JTA extension that supplies the `Transaction` object?)
- Lets EclipseLink find a named `DataSource` by name without using JNDI
  - (Remember the Helidon Hikari extension that enables `DataSource`s to be discoverable in CDI by name?)
- Zero configuration; just works

Helidon Hibernate Extension

- A simple library and CDI portable extension that **makes Hibernate think it is running in a Java EE application server even though it's not**
- Lets Hibernate find the current JTA transaction without using JNDI
  - (Remember the Helidon JTA extension that supplies the `Transaction` object?)
- Lets Hibernate find a named `DataSource` by name without using JNDI
  - (Remember the Helidon Hikari extension that enables `DataSource`s to be discoverable in CDI by name?)
- Zero configuration; just works

EE-Mode JPA + MicroProfile: Putting It Together

- **Put these extensions on your runtime classpath.**
- **The end.**
  - (Well, almost the end. Then you Do Ordinary EE-Mode JPA Things™ as you normally would.)
    - Configure your data source
    - Set up your `META-INF/persistence.xml`
    - Design your entities and tables

Example Code and Demo