MICROSERVICES

Fast, flexible data access in Java using the Helidon microservices platform

Helidon SE and Helidon MP provide a very diverse array of methods for accessing data sources.

by Paul Parkinson

February 12, 2021

Helidon is a collection of Java libraries for writing microservices. Helidon 2.2.0 is out and provides a very diverse and flexible array of methods for accessing data. In this article, I'll provide an overview of those data-access methods with references to lower-level material and examples.

First, though, here's a bit of history.

The techniques and technologies used to access databases are based on various criteria that take into account the best fit for data access needs as well as the platform being run. **Figure 1** is a time line of relevant events for Java platforms and data access.

Figure 1. A time line of Java platforms and data access

Jakarta EE is the current and future home of the Java EE platform, and Eclipse MicroProfile was created to extend the enterprise Java environment for developing microservices. Alongside these standards are frameworks such as Spring Boot, Helidon, and Micronaut. **Figure 2** shows the wide range of technologies supported by Helidon. You can use these technologies based on the platform you are running or what you are most comfortable with for whatever reason.

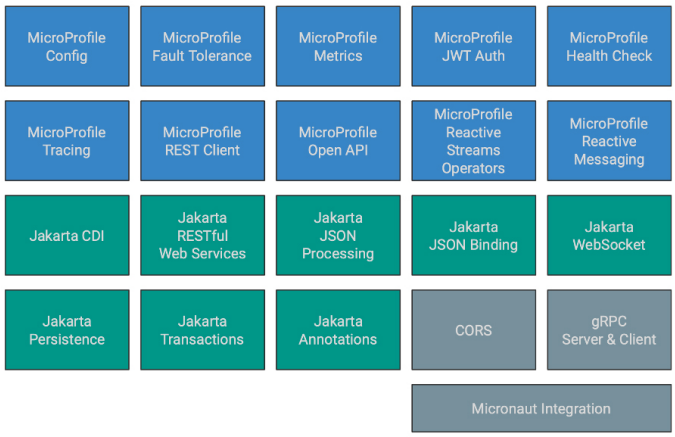

Figure 2. Helidon supports a wide range of technologies.

Helidon is an open source project funded by Oracle, so it integrates and aligns with Oracle database technologies and features extremely well. This includes support for the Oracle Universal Connection Pool (UCP) for JDBC.

For Oracle Database, UCP provides performance (via connection pooling and tagging), scalability (via a front end for Database Resident Connection Pool [DRCP], a shared pool for multitenant databases, a swim lane for sharded databases, and sharding data sources), and high availability (via the Transaction Guard, Application Continuity, and Transparent Application Continuity features of Oracle Database).

Helidon SE and Helidon MP

There are Java SE and MicroProfile (MP) versions of Helidon. Helidon SE is designed to be transparent without using contexts and dependency integration (CDI), reactive programming, and functional-style programming. Helidon MP is for those familiar with Java EE; this version uses CDI.

Helidon MP's CDI integration supports both HikariCP and UCP JDBC connection pools via the following Maven dependencies:

```
<dependency>
    <groupId>io.helidon.integrations.cdi</gro
    <artifactId>helidon-integrations-cdi-data
    <scope>runtime</scope>
</dependency>

<dependency>
    <groupId>io.helidon.integrations.cdi</gro
    <artifactId>helidon-integrations-cdi-data
    <scope>runtime</scope>
</dependency>
```

The following is a sample `META-INF/microprofile-config.properties` file for a Helidon microservice that connects to an Oracle Autonomous

Transaction Processing database. Note the MicroProfile naming convention of *[objectype].[objectname].[objectproperty]*:

```
oracle.ucp.jdbc.PoolDataSource.orderpdb.URL =
oracle.ucp.jdbc.PoolDataSource.orderpdb.user
oracle.ucp.jdbc.PoolDataSource.orderpdb.passw
oracle.ucp.jdbc.PoolDataSource.orderpdb.conne
oracle.ucp.jdbc.PoolDataSource.orderpdb.inact
```

If you don't need data source/UCP-specific APIs, you could use the more generic `javax.sql.DataSource`, `javax.sql.DataSource.orderpdb` naming convention.

Here's an example of microservice code where the data source reference configured above is automatically injected.

```
@Inject
@Named("orderpdb")
PoolDataSource atpOrderPdb;
```

These simple steps make data source access simple and dynamic and a perfect fit for cloud environments.

Kubernetes deployments

Here is a Kubernetes deployment YAML file for a microservice, which sets the environment variables (with values acquired from Kubernetes secrets, vault, and so on) that will override the defaults set in the Helidon configuration:

```
containers:
- name: order
  image: %DOCKER_REGISTRY%/order-helidon:0.1
  imagePullPolicy: Always
  env:
  - name: oracle.ucp.jdbc.PoolDataSource.orde
    value: "ORDERUSER"
  - name: oracle.ucp.jdbc.PoolDataSource.orde
    valueFrom:
      secretKeyRef:
        name: atp-user-cred-orderuser
        key: password
  - name: oracle.ucp.jdbc.PoolDataSource.orde
value: "jdbc:oracle:thin:@%ORDER_PDB_NAME%_tp
```

Java EE/Jakarta EE Persistence API

The Java EE/Jakarta EE Persistence API (JPA), first released in 2009, is still the most widely used API for object-relational mapping. JPA is used not only in Java EE and Jakarta EE applications but also in other frameworks such as Spring Boot and Helidon MP.

Hibernate and Eclipse are the most popular implementations of JPA and are both supported by Helidon MP. Therefore, it is simple to migrate the use of JPA in applications that run JPA (whether they are from an application server, Spring Boot, or some other platform) to the lighter-weight Helidon.

More details on Helidon and JPA can be found in "Helidon and JPA" by Laird Nelson, as well as in "Data persistence with Helidon and Native Image" by Tomáš Kraus.

Micronaut Data

Micronaut is a JVM-based, full-stack framework for building modular microservices and serverless applications. The Micronaut framework was released in late 2018, around the same time as Helidon, and has been very successful in providing a smooth transition from Spring Boot to its platform by providing the ability to do the following:

- Integrate Spring components into a Micronaut application
- Run Spring applications as Micronaut applications
- Expose Micronaut beans to a Spring application

Helidon has an integration layer that allows the use of Micronaut features from within a Helidon microservice. These features include Micronaut singleton injection, Micronaut interceptors, Micronaut bean validation and, of particular interest to the current topic, Micronaut Data.

The Micronaut Data database access toolkit precomputes queries and executes them with a thin runtime. Micronaut Data provides a general API for translating a query model into a query at compile time and provides runtime support for JPA/Hibernate and SQL/JDBC back ends.

Inspired by GORM and Spring Data, Micronaut Data improves on these two technologies by eliminating the runtime model that uses reflection, eliminating query translation that uses regular expressions and pattern matching, and adding type safety. The use of reflection in GORM and Spring Data for modeling relationships between entities leads to more memory consumption.

Because Micronaut Data does not perform query translation at runtime—it's all precomputed—the performance gain can be significant. Micronaut Data JDBC provides nearly 2.5 times the performance of Spring Data; Micronaut Data JPA provides up to 40% better performance than Spring Data JPA. Also, startup times are at least 1.5 times faster than that of Spring Boot.

Micronaut Data supports GraalVM native images for both the JPA and JDBC implementations. The currently supported databases are H2, PostgreSQL, Oracle Database, MariaDB, and Microsoft SQL Server.

Some considerations for the use of direct Micronaut Data JDBC compared to JPA, aside from the performance and memory efficiencies mentioned, include the fact that JDBC has fewer dialects than JPA, is optimized for reads instead of writes (the opposite of JPA), and is better for startup times and, thus, serverless applications.

By integrating with Micronaut in this way, Helidon also inherits the simplicity of porting Spring Boot applications to Helidon. Tomas Langer has written a detailed article on this subject: "Helidon with Micronaut Data repositories."

Helidon DB Client

Helidon SE is a compact toolkit that embraces the latest Java SE features, such as reactive streams, asynchronous and functional programming, and fluent-style APIs. Helidon DB Client API, designed for Helidon SE, simplifies how you work with databases by abstracting the type of the database. The API can be used for both relational and nonrelational databases.

Helidon DB Client provides

- Database configuration abstraction: Using a Helidon configuration allows database implementation-specific configuration options without the need to use database implementation-specific APIs. This allows for seamless switching between databases based on configuration.
- Statement configuration abstraction: Using a Helidon configuration allows the use of database-specific statements. This enables the use of different databases on different environments without changing code.
- A unified API for data access and querying: Thanks to the statement configuration abstraction, you can invoke a statement against relational or nonrelational databases (such as MySQL and MongoDB) without modifying source code.
- Reactive database access with backpressure: Currently the client supports a natively reactive driver for MongoDB and an executor service-wrapped support for any JDBC driver. This allows for seamless use of JDBC drivers in a reactive nonblocking environment, including support for backpressure (the result set is processed as requested by the query subscriber).
- Observability: The API offers support for health checks, metrics, and tracing.

Using the API with MongoDB simply requires adding the following Maven dependency:

```
<dependency>
            <groupId>io.helidon.dbclient</gro
                 <artifactId>helidon-dbclient
      </dependency>
```

And a configuration such as this:

```
db:
  source: "mongoDb"
  connection:
    url: "mongodb://127.0.0.1:27017/pokemon"
  statements:
    # Insert operation contains collection na
    # Name variable is stored as MongoDB prim
    insert2: '{
            "collection": "pokemons",
            "value": {
                "_id": $name,
                "type": $type
            }
        }'
```

Here's how you can code the Helidon DB Client and register the endpoints to access the data source:

```
Config dbConfig = config.get("db");
        DbClient dbClient = DbClient.builder(
                // add an interceptor to name
                .addService(DbClientMetrics.c
                // add an interceptor to stat
                .addService(DbClientMetrics.t
                                        .statemen
                // add an interceptor to all
                .addService(DbClientTracing.c
                .build();

        HealthSupport health = HealthSupport.
                .addLiveness(DbClientHealthCh
                .build();

        return Routing.builder()
                .register(health)
                .register(MetricsSupport.crea
                .register("/db", new PokemonS
                .build();
```

You can learn more from "Helidon DB Client" by Tomáš Kraus.

The Neo4j graph database

Helidon works with relational and nonrelational databases, SQL and NoSQL databases, and many more databases. These include JDBC, MongoDB via the MongoDB client, and Oracle Database JSON database via Oracle's Simple Oracle Document Access (SODA) API – and recently, the Helidon project added integration with the graph database Neo4j. The Neo4j integration can be enabled with the following Maven dependencies:

```
<dependency>
                    <groupId>io.helidon.integ
                    <artifactId>helidon-integ
                    <version>${helidon.versio
```

```
                </dependency>
                <dependency>
                    <groupId>io.helidon.integ
                    <artifactId>helidon-integ
                    <version>${helidon.versio
                </dependency>
                <dependency>
                    <groupId>io.helidon.integ
                    <artifactId>helidon-integ
                    <version>${helidon.versio
        </dependency>
```

As with all Helidon features, configuration may be done in the `application.yaml` file:

```
neo4j:
  uri: bolt://localhost:7687
  authentication:
    username: neo4j
    password: secret
  pool:
metricsEnabled: true #should be explicitly en
```

Or it can be done via a MicroProfile configuration:

```
neo4j.uri=bolt://localhost:7687
neo4j.authentication.username=neo4j
neo4j.authentication.password: secret
neo4j.pool.metricsEnabled: true #should be ex
```

Here's how to use Neo4j with Helidon SE:

```
Neo4JSupport neo4j = Neo4JSupport.builder()
        .config(config)
        .helper(Neo4JMetricsSupport.create())
        .helper(Neo4JHealthSupport.create())
        .build();

 Routing.builder()
        .register(health)                   /
        .register(metrics)                  /
        .register(movieService)
        .build();
```

Neo4j can be used in Helidon SE by simply injecting the driver, for example:

```
@Inject
Driver driver;
```

Coherence Community Edition

Coherence Community Edition (CE) is a reliable and scalable platform for state management. It integrates with Helidon, GraalVM, Oracle Database, and Oracle Database cloud services.

Coherence CE contains the in-memory data grid functionality necessary to write microservices applications. Its features include

- Fault-tolerant automatic sharding
- Scalable caching, querying, aggregation, transactions, and in-place processing
- Polyglot programming on the grid side with GraalVM
- Persistence and data source integration
- Creating events, sending messages, and streaming
- A comprehensive security model
- Unlimited clients in polyglot languages and over REST
- Docker and Kubernetes support, with Kibana and Prometheus dashboards

Helidon 2.2.0 supports the MicroProfile GraphQL specification, which is an open source data query and manipulation language for APIs. A recent article, "Access Coherence using GraphQL," by Tim Middleton, shows how to create and use GraphQL endpoints to access data in Coherence CE seamlessly with Helidon MP.

Messaging for Oracle Advanced Queuing

Due to the nature of microservices environments, messaging is often used for interservice communications, and that's what the MicroProfile Reactive Messaging specification was designed for.

The Oracle Advanced Queuing (AQ) messaging system has been part of Oracle Database since 2002. The system, which supports Java Message Service (JMS), has features that make it perfect for microservices development, including

- Transactional queues and an "exactly once" delivery guarantee so you're not forced to code logic for idempotency
- The ability to conduct database work and produce and consume messages within the same local transaction. This facilitates event sourcing, sagas, and general transaction communication patterns used in microservices with atomic (and, again, exactly-once delivery) guarantees not possible with other messaging and database systems

The integration of Oracle AQ with Helidon is powerful and simple to use. Here is an example where an Oracle AQ JMS ("order-placed") message is received, the underlying JDBC connection is obtained and used to do database work (check inventory), and a response message ("inventory-exists") is sent.

These three actions are conducted within the same local transaction such that all either fail or succeed, thus relieving an administrator or developer from needing to intervene and rectify a system due to a failure or add logic to a microservice to handle failures such as duplicate deliveries or inconsistent data.

```
@Incoming("orderplaced")
@Outgoing("inventoryexists")
@Acknowledgment(Acknowledgment.Strategy.NONE)
public CompletionStage<Message<String>> reser
        return CompletableFuture.supplyAsync(
            Connection jdbcConnection = msg.g
   String inventoryStatus = getInventoryForOr
            return Message.of(inventoryStatus
        });
}
```

See “Helidon messaging with Oracle AQ,” by Daniel Kec, for a deeper look.

GraalVM Native Image

All the features mentioned in this article are compatible with GraalVM, which means that Helidon microservices using those features can be built into a GraalVM Native Image, a technology that performs an ahead-of-time compilation of Java code to create a standalone executable.

With the new Oracle Database 21c, GraalVM Native Image support also works with Oracle Universal Connection Pool (UCP) wallets and the Oracle Autonomous Transaction Processing cloud database service.

You can learn more about this and the JDBC 21.1.0.0 reconfiguration for GraalVM Native Image in “New Year goodies —Oracle JDBC 21.1.0.0 on Maven Central,” by Kuassi Mensah.

Integration with sagas and MicroProfile LRA

Applications that require data coordination between multiple microservices create challenges for data consistency and integrity. Those challenges necessitate changes in the transaction processing and data patterns used by them.

Traditional systems rely on two-phase commit or other extended architecture (XA) protocols that use synchronous communication, resource locking, and recovery via rollback or commit. While those protocols provide strong consistency and isolation, they do not scale well in a microservices environment due to the latency of held locks. That means such methods are suitable for only a small subset of microservices use cases—generally those with low throughput requirements.

The saga design pattern, by contrast, uses asynchronous communication and local resources only (thus, no distributed

locks) and recovery via compensating actions. The saga pattern scales well, so it is well suited for long running transactions in a microservices environment. Additional application design considerations are necessary, however, for read isolation and compensation logic and debugging can be tricky.

That's where the MicroProfile Long Running Actions (LRA) API comes in. You can run MicroProfile LRA in Helidon. See more in my article, "Long running actions for MicroProfile on Helidon… Data integrity for microservices."

Conclusion

Helidon is an extremely versatile Java library for accessing data in microservices. This article briefly described the capabilities of Helidon SE and Helidon MP to give you an overall picture of the landscape.

Dig deeper

- Helidon: A simple cloud native framework
- Reactive streams programming over WebSockets with Helidon SE
- Hello, Coherence Community Edition, Part 1: Creating cloud native stateful applications that scale
- Transition from Java EE to Jakarta EE
- Building Microservices with Micronaut
- GraalVM: Native Images in Containers

Paul Parkinson

As the Data and Transaction Processing Development Lead for the Helidon Microservices Cloud Platform, **Paul Parkinson** works with a wide variety of languages and technologies such as Oracle Database, Java MicroProfile, Oracle Cloud Infrastructure, Sagas, Command and Query Responsibility Segregation (CQRS), Event Sourcing, Kubernetes, Istio, Jaeger, Grafana, Kiali, Open Service Broker, Kafka, and Event Broker. Follow him @paulparkinson.

Share this Page