Thorntail Documentation

The Thorntail Team

Version 4.0.0-SNAPSHOT

Table of Contents

Thorntail is the next generation of WildFly Swarm.

Introduction

Chapter 1. Lessons Learned

> Don’t cling to a mistake just because you spent a lot of time making it.

Mangling artifacts is dangerous

When you mangle and repackage a user’s artifacts and dependencies, it can many times go awry.

Don’t replace Maven

Let Maven (or Gradle) handle the entirety of pulling dependencies. We cannot predict the topology of someone’s repository managers, proxies and network.

Don’t get complicated with uberjars

The more complex our uberjar layout is, the harder it is to support Gradle or other non-Maven build systems.

Classpaths are tricky

If different codepaths are required for executing from Maven, an IDE, a unit-test, and during production, you will have a **bad time**.

Don’t insist on uberjars

For Linux containers, people want layers that cleanly separate application code from runtime support code.

Testability is important

A slow test is a test that is never willingly executed. PRs take forever to validate. Users like to be able to test their own code quickly and iteratively.

Easily extensible means ecosystem

If it’s entirely too difficult to extend the platform, the ecosystem will not grow. New integrations should be simple.

Related: Core things should not be any more first-class than community contributions

For instance, auto-detection in WildFly Swarm only worked with core fractions; user-provided wouldn’t auto-detect.

Ensure the public-vs-private API guarantees are clear.

Intertwingly code (and javadocs) make finding the delineation between public API and private implementations difficult.

Allow BYO components

We don’t want to decide *all* of the implementations, and certainly not versions, of random components we support.

Be a framework, not a platform

Frameworks are easier to integrate into an existing app; a platform becomes the target with (generally too many) constraints.

Maintain tests & documentation

Ensure the definition of "done" includes both tests and documentation.

Productization complexity

The greater divergence between community and product, the more effort is required for productization. Complicating any process to automate productization from community.

BOM complexity

Related to productization as well, but of itself having a handful of BOMs made life confusing for us and for users. There were often times where fractions would be "Unstable" or "Experimental" for months with no real reason other than we forgot to update it.

Concepts

Chapter 2. Microservice

A microservice is small application with a *bounded domain*. A microservice is intended to solve a semantically constrained problem related to a larger system. In a microservice-based architecture, an *application* is made from a collection of many *microservices*.

Chapter 3. CDI-native

Thorntail is built from the from ground-up to be CDI-native. Building applications of any notable size benefit from the usage of a dependency-injection framework.

Chapter 4. MicroProfile-native

Thorntail is built from the from ground-up to be MicroProfile-native. MicroProfile addresses many of the needs and requirements of microservices-centric applications. Instead of bolting MicroProfile facilities on, Thorntail natively supports the various MicroProfile specifications directly.

Chapter 5. Flat Classpath

While Java application servers previously have had the ability to support multiple disparate applications, when building microservices, a runtime need only support a single application, or service. With a microservices architecture, significantly fewer resources and capabilities may be required for each service. Freely mixing service and application implementations becomes significantly less problematic and certainly less cumbersome.

That being said, the Java Platform Module System (JPMS) may become beneficial in the future after further adoption by other upstream projects.

Tools

Chapter 6. Maven Plugin

The `thorntail-maven-plugin` exists to make packaging your application easier.

Basic Configuration

As with any Maven plugin, configuration occurs within your project's `pom.xml`

The plugin has one available goal: `package`. The behavior of this goal is controlled by the plugin configuration, described below.

```
<plugin>
  <groupId>io.thorntail</groupId>
  <artifactId>thorntail-maven-plugin</artifactId>
  <version>4.0.0-SNAPSHOT</version>
  <configuration>
    <!-- global configuration -->
  </configuration>
  <executions>
    <execution>
      <goals>
        <goal>package</goal>
      </goals>
      <configuration>
        <!-- execution-specific configuration -->
      </configuration>
    </execution>
  </executions>
</plugin>
```

6.1. Modes

The plugin can operate in two modes: *fat* and *thin*, with *fat* being the default. The mode is selected by a `<mode>⋯</mode>` block within the plugin configuration, or by the `thorntail.mode` property.

fat

Produces an executable build that includes all dependencies and your application artifact.

thin

Produces an executable build that includes all dependencies but *not* your application artifact.

Mode is an independent concept from *format*, described below.

```
<configuration>
  <mode>thin</mode>
</configuration>
```

6.2. Formats

The plugin can produce three different types of executable distributions: *jar*, *dir*, and *zip*, with *jar* being the default. The format is selected by a `<format>⋯</format>` block within the plugin configuration, or by the `thorntail.format` property. These contents of any of these formats is still defined by the *mode*, described above.

jar

Produces a fat jar (or *überjar*) containing the cotnents defined by the *mode* above. The jar may be executed using normal `java -jar` commands.

dir

Produces a directory containing the contents defined by the `mode` above, along with scripts to easily execute it. The *dir* layout may be best suited for container-related pipelines, where all runtime support aspects are added to a base layer, and the topmost layer contains only the vanillia application artifact. To achieve this method, `mode` should be configured to be `thin`.

zip

Produces the same content as the *dir* format, but as a `.zip` file.

```
<configuration>
  <format>dir</format>
</configuration>
```

6.3. `main()`

The plugin will attempt to discover an existing non-ambiguous `main(⋯)` within your application. If it finds none, a default `main(⋯)` will be configured. If it finds a single application-provided `main(⋯)`, it will be used. If it finds multiple application-provided `main(⋯)` methods, an error will result. To resolve an ambiguous `main(⋯)` error, a `mainClass` may be configured using a `<mainClass>⋯</mainClass>` block within the plugin configuration, or by the `thorntail.mainClass` property.

```
<configuration>
  <mainClass>com.mycorp.myapp.Main</mainClass>
</configuration>
```

6.4. Other configuration

Naming

The artifact produced will include the Maven classifier of `-bin`. This classifier may be changed using the `<classifier>⋯</classifier>` configuration element, or `thorntail.classifier` property.

The artifact will be named the same as the primary project artifact (according to `${project.finalName}`), unless a plugin configuration of `<finalName>⋯</finalName>` or a property of `thorntail.finalName` is provided.

Attaching

If the format is `jar` or `zip`, it will be attached to the Maven project, causing it to be built or deployed to the repository. If the format is `dir`, it can not be attached.

To disable attaching of a `jar` or `zip` build, a configuration block of `<attach>⋯</attach>` or property of `thorntail.attach` may be set to `false`.

6.5. Distribution Structure

Directory and Zip

When `dir` or `zip` formats are selected, the layout of the resulting tree is relatively simple:

`bin/`
: Directory containing platform-specific scripts to execute the application.

`bin/run.sh`
: A Unix-compatible shell script for launching the application. If the distribution was built as a `thin` distribution, the application archive must be provided in one of two ways:

  - As an argument to the `run.sh` command.
  - By placing it within the `app/` directory.

`bin/run.bat`
: A Windows-compatible batch script for launching the application. If the distribution was built as a `thin` distribution, the application archive must be provided in one of two ways:

  - As an argument to the `run.bat` command.
  - By placing it within the `app/` directory.

`app/`
: A directory to contain the application archive. If the distribution was built as a `thin` distribution, this directory will be empty. When using containers, the top-most layer may be responsible for placing the application archive in this location, or may mount the archive into this directory when run.

`lib/`
: Contains all dependencies for the application. Care is taken to ensure last-modified timestamps of the contents of this directly do not change needlessly.

Jar

When the `jar` format is selected, the contents of the jar are also relatively simple:

`*.jar`
: All `.jar` archives are placed within the root of the resulting `-bin.jar`.

`bin/Run.class`
: A bootstrapping class is provided which can set up the classpath given the contents at the root of the jar. The bootstrap class will extract all of the `.jar` artifacts from the root to a cache directory at `$HOME/.thorntail-cache`. The extracted jars will have a SHA-1 hash added to their names in

order to disambiguate any identically named jars from this or other applications, as the cache is shared.

`META-INF/MANIFEST.MF`

The Jar manifest is configured to run the `bin.Run` main bootstrapping class when `java -jar` is used.

Chapter 7. Maven Archetypes

Maven archetypes are provided to make it quick to get started with new Thorntail projects.

7.1. JAX-RS

Maven Coordinates

```
<dependency>
  <groupId>io.thorntail.archetypes</groupId>
  <artifactId>thorntail-jaxrs-archetype</artifactId>
</dependency>
```

Create a new project

```
mvn archetype:generate \
    -DarchetypeGroupId=io.thorntail.archetypes \
    -DarchetypeArtifactId=thorntail-jaxrs-archetype \
    -DarchetypeVersion=4.0.0-SNAPSHOT \
    -DgroupId=com.mycorp \
    -DartifactId=my-app \
    -Dversion=1.0-SNAPSHOT
```

Your project will be created in the `my-app/` directory, and contain stubs to get your started. These stubs include a JUnit-based test, along with appropriate configuration of your `pom.xml`.

Chapter 8. Testing with JUnit

Thorntail provides a JUnit `TestRunner` implementation which allows your JUnit tests to execute within the context of your full application. To use the `TestRunner`, you must include the `testing` artifact with `<scope>test</scope>`.

Maven Coordinates

```
<dependency>
  <groupId>io.thorntail</groupId>
  <artifactId>thorntail-testing</artifactId>
  <scope>test</scope>
</dependency>
```

Use the `ThorntailRunner`

Write your JUnit test as usual, but include a class-level annotation of `@RunWith(ThorntailRunner.class)`

```
@RunWith(ThorntailRunner.class)
public class MyTest {
  // tests go here
}
```

Participate in CDI

Your test class will be instantiated and injected for each test method. You may use `@Inject` to inject any component available to your application. The entirety of your application will be booted and available.

```
@RunWith(ThorntailRunner.class)
public class MyTest {

  public void testSomething() throws Exception {
    assertThat(myLunch.getCheese()).isEqualTo("cheddar");
  }

  @Inject
  private Lunch myLunch;
}
```

Use `@EphemeralPorts`

If the annotation `@EphemeralPorts` is applied at the class level, and your application uses a servlet container, then arbitrary emphemeral ports will be selected and used. This may be useful when running tests on a CI machine or if you wish to parallelize your tests.

In order to know what port are actually selected and in-use, you may `@Inject` either a `@Primary` or `@Management URL` or `InetSocketAddress` component. These instances are made available throw the

Servlet component.

fest-assert

The `testing` artifact transitively brings in fest-assert for making fluent assertions in your tests.

RestAssured

The `testing` artifact transitively brings in RestAssured to enable easily testing of HTTP endpoints. Additionally, it preconfigures the `RestAssured.baseURI` to the URL for the primary web endpoint, if available. The preconfiguration of the `baseURI` is especially useful when you use `@EphemeralPorts`.

Related Information

Testing with Arquillian

Chapter 9. Testing with Arquillian

Arquillian is a framework which assists with both blackbox and *in-container* testing of your components. The MicroProfile TCKs use the Arquillian framework in order to verify compliance with the specifications.

Maven Coordinates

To use the Arquillian integration, include the `testing-arquillian` artifact in your project with a `<scope>test</scope>` block.

```
<dependency>
  <groupId>io.thorntail</groupId>
  <artifactId>thorntail-testing-arquillian</artifactId>
  <scope>test</scope>
</dependency>
```

Arquilian Deployable Container

Thorntail provides an Arquillian-compatible *deployable container* which allows a developer to deploy only the components they wish to test. Additionally, the tests themselves may either be blackbox (`@RunAsClient`) or *in-container* where they can directly interact with the components under test.

Writing an in-container Test

Using JUnit, write a test as you normally would, but include a class-level annotation of `@RunWith(Arquillian.class)`.

Additionally, to specify the components you wish to be tested, you must provide a method marked `@Deployment` which produces a ShrinkWrap archive to be consider as the application.

```
@RunWith(Arquillian.class)
public class MyTest {

  @Deployment
  public static JavaArchive myDeployment() {
    JavaArchive archive = ShrinkWrap.create(JavaArchive.class);
    // set up archive
    return archive;
  }

}
```

For in-container tests, the test class itself (`MyTest` in this case) is considered an injectable CDI bean. Any components your application creates, or which are normally available from Thorntail may be injected.

```
@RunWith(Arquillian.class)
public class MyTest {

  @Deployment
  public static JavaArchive() {
    JavaArchive archive = ShrinkWrap.create(JavaArchive.class);
    // set up archive
    return archive;
  }

  public void testSomething() throws Exception {
    assertThat(myLunch.getCheese()).isEqualTo("cheddar");
  }


  @Inject
  private Lunch myLunch;

}
```

Related Information

Testing with JUnit

Chapter 10. Developer Tools

Thorntail provides a set of developer tools to allow for restarting or reloading classes when developing a Thorntail-based application. The simple ability to restart a running process when compiled `.class` files or packaged `.jar` files are changed is built in to the core. To gain the ability to hot reload classes into an running executable, an additional dependency is required.

Maven Coordinates

```
<dependency>
  <groupId>io.thorntail</groupId>
  <artifactId>thorntail-devtools</artifactId>
  <scope>test</scope>
</dependency>
```

Setting the `THORNTAIL_DEV_MODE`

To enable either the restarting of processes or hot-reloading of classes, the environment variable `THORNTAIL_DEV_MODE` must be set.

`restart`
: Capability always included in the core, which will watch the contents of the classpath. Upon noticing changes, the process will be terminated and restarted, causing the JVM to load new versions of all classes.

`reload`
: Capbility only available if the above `thorntail-devtools` dependency is added to the project. It will watch for changes to the contents of the classpath (only `.class` files, not packaged `.jar` files) and attempt to load and redefine the classes within the running process.

Using `restart` mode

Restart works primarily with directory layouts. The provided `bin/run.sh` will use either the application's own packaged `.jar` if built using `<mode>fat</mode>` or will attempt to use `target/classes/` if built with `<mode>thin</mode>`. Start the process with the environment variable set to `restart`

```
$ THORNTAIL_DEV_MODE=restart ./target/myapp-bin/bin/run.sh
```

The rebuilt your project as appropriate

```
$ mvn compile
```

Within the console of the running process, you should see, within a few seconds, the process stop and restart automatically.

Using `reload` mode

Add the above Maven `<dependency>` block to your project.

Then follow the same steps as for `restart`, but setting the mode to `reload`.

```
$ THORNTAIL_DEV_MODE=reload ./target/myapp-bin/bin/run.sh
```

The rebuilt your project as appropriate

```
$ mvn compile
```

Additionally, the same behavior is available if you execute your `main()` directly from your IDE with the environment variable set appropriately. Triggering a recompilation from within your IDE should also cause hot reloading of your classes within the running process.

Components

Chapter 11. Kernel

Maven Coordinates

The core of Thorntail is usually brought in transitively through other dependencies. It's Maven coordinates are:

```
<dependency>
  <groupId>io.thorntail</groupId>
  <artifactId>thorntail-kernel</artifactId>
</dependency>
```

CDI Components

`ThreadFactoryProducer`

Produces a `Dependent` scoped `ThreadFactory` for utilizing `Thread` instances.

`IndexProducer`

Produces an `Application` scoped `IndexView` representing the jandex'd files of the Deployment, read from `META-INF/thorntail.idx` which is created by the plugin. If not found, it produces an empty `IndexView` instance.

11.1. Configuration

Configuration of applications built on Thorntail is performed using MicroProfile-config mechanisms. The default `microprofile-config.properties` file located within the `META-INF/` directory of the application can be used to set or override default configuration values. The same file may be used to provide application-specific configuration which does not directly affect the Thorntail behavior.

Additionally, other files, both within `META-INF/` and on the filesystem may contribute to the final configuration, with various degrees of priority. The priority may be controlled on a file-by-file basis using the MicroProfile-config `config_ordinal` property within each file. Files with larger priorities will override values set in files with lower priorities.

Profiles

Configuration files may be conditionally activated using *profiles*. Profiles are activated by setting the Java system property of `thorntail.profiles` or the system environment variable of `THORNTAIL_PROFILES` to a comma-separated list of names.

Search Paths & Explicit Configuration Files

To externalize configuration, the Java system property of `thorntail.config.location` or the system environment variable of `THORNTAIL_CONFIG_LOCATION` may be set to a system-dependent delimited set of paths. Each path is considered in turn, withi increasing priority. If a path is a directory, it will be searched for appropriate configuration files matching any activated profiles. If a path is a regular file, it will be loaded, regardless of name or activated profiles.

YAML

If the application includes a dependency on `snakeyaml`, then YAML-based configuration files will also be located and loaded.

Environment Variables

All configuration items may be set through environment variables. As the format used for many configuration keys may include characters not allowed as environment variable names, a mechanical translation is performed. A requested configuration key is converted to uppercase, and each dot is replaced with an underscore. For example, a configuration key of `web.primary.port` may be configured through an environment variable named `WEB_PRIMARY_PORT`.

Framework Defaults

Each framework component may include default values for any required configuration item. These defaults have a priority of `-1000` to allow easy overriding of them.

Table 1. Configuration Sources

Path	Priority	Notes
`META-INF/framework-defaults.propertes`	-1000	Located via classloader and provided by framework components.
`META-INF/microprofile.propertes`	100	Located via classloader.
`META-INF/application.properties`	200	Located via classloader.
`META-INF/application.yaml`	200	Located via classloader, if SnakeYAML is available
`META-INF/application-profile.properties`	250+	Located via classloader, in order specified, with increasing priority.
`META-INF/application-profile.yaml`	250+	Located via classloader, in order specified, with increasing priority.
`application-profile.properties`	250+	Located via filesystem from specified search paths, in order, with increasing priority.
`application-profile.yaml`	250+	Located via filesystem from specified search paths, in order, with increasing priority.
`path`	275	Located via filesystem, through explicit property or environment variable.
environment variables	300	Converted from all available system environment variables.
system properties	400	All available system properties.

Interpolation

Configuration values may be interpreted and assembled from other values. Interpolation expressions are wrapped within delimeters of `${' and `}'. Additionally, expressions may provide a default value, which may in turn be another expression or a literal. All interpolation is performed before using the value converters to convert to the desired type.

As with normal usage of `Config`, if an interpolation expression references a configuration key and provides no default, if that key does not exist, a `NoSuchElementException` will be thrown.

In the event that a literal `${` is desired within a value, without interpolation, a `\` character may be used to escape it.

All other `\` which appear before any other character will be included literally in the value, not as an escape.

`${web.primary.port}`
: Will be replaced with the current value of the configuration item `web.primary.port` if it exists. If no such value exists, an exception will be thrown.

`${web.primary.port:8080}`
: Will be replaced with the current value of the configuration item `web.primary.port` if it exists. If no such value exists, the value of `8080` will be provided, and converted as appropriate.

`${web.management.port:${web.primary.port:8080}}`
: Will be replaced with the current value of the configuration item `web.management.port` if it exists. If not such value exists, will be replaced by the current value of the configuration item `web.primary.port` if it exists. If no such value exists, the value of `8080` will be provided, and converted as appropriate.

`thing-${thing.type:default}-impl`
: Will be a combination of the literal `thing-` text, the value of `thing.type` configuration item` if present, using the word 'default' if not, with a suffix of `-impl`.

`%40-$`
: Will result in a string literal of `%40-$`

`\${foo}`
: Will result in a string literal of `${foo}` without interpolation, removing the escape character.

`foo\,bar`
: Will result in a string literal of `foo\,bar` without removal of the escape character.

Related Information

- MicroProfile Configuration Spec

Chapter 12. Java EE

12.1. Bean Validation

The Bean Validation component provides for using *bean validation* according to JSR 380.

Maven Coordinates

```
<dependency>
  <groupId>io.thorntail</groupId>
  <artifactId>thorntail-bean-validation</artifactId>
</dependency>
```

CDI Components

Injectable components are defined by the Bean Validation specification.

Related Information

- DataSources
- JMS

12.2. Servlet

The Servlet component of Thorntail enables basic Java Web Servlet processing and serving.

Maven Coordinates

To include the servlet component, add a dependency:

```
<dependency>
  <groupId>io.thorntail</groupId>
  <artifactId>thorntail-servlet</artifactId>
</dependency>
```

Implicit Deployment

An application archive will be scanned for all `Servlet` implementations and added to a default deployment. The `@WebServlet` annotation should be used to configure the servlet as desired.

Explicit Deployments

To have more control over the deployment, the application may use normal CDI facilities to produce instances of `DeploymentMetaData`. Each instance of `DeploymentMetaData` will be individually deployed to the underlying servlet container.

Management Deployments

Various other components, such as *Metrics* and *Health* produce additional web endpoints. Each of these are marked as *management* deployments. By default, these management deployments will be automatically deployed alongside the application deployment. The servlet component may be

configured (see below) to separate application endpoints from management endpoints.

Configuration of Primary Endpoints

If the management endpoints (see below) are not configured separately, then the primary configuration applies to all endpoints.

web.primary.host

Sets the host or interface to bind the primary endpoint connection listener.

web.primary.port

Sets the port to bind the primary endpoint connection listener. If this value is set to `0`, a random available port will be used.

Configuration of Management Endpoints

Two configuration properties control which host and port management endpoints are served from. By default, they match the primary host and port, and serve from the same connection.

To change the management host or port, use the following configuration properties:

web.management.host

Sets the host or interface to bind the management endpoint connection listener.

web.management.port

Sets the port to bind the management endpoint connection listener. If this value is set to `0`, a random available port will be used.

Configuration of Undertow

The servlet componet includes a variety of configuration options related to the default Undertow-based implementation.

undertow.io-threads

The number of I/O threads to use by the web server. By default it is calculated as the maximum of `2` or the number of available CPUs.

undertow.worker-threads

The number of worker threads used by the web server. By default it is calculated as 8 times the number of I/O threads.

undertow.high-water

The high water mark for a server's connections. Once this number of connections have been accepted, accepts will be suspended for that server.

undertow.low-water

The low water mark for a server's connections. Once the number of active connections have dropped below this number, accepts can be resumed for that server.

undertow.tcp-nodelay

Configure a TCP socket to disable Nagle's algorithm.

undertow.cork

Specify that output should be buffered. The exact behavior of the buffering is not specified; it may flush based on buffered size or time.

CDI Components

To enable creation of well-integrated applications, the Servlet component provides access to several CDI components.

`@Primary URL`

A `URI` with the qualifier of `@Primary` is available for injection. It specifies the URL of the primary endpoint.

`@Primary InetSocketAddress`

An `InetSocketAddress` with the qualifier of `@Primary` is available for injection. It specifies the address and port of the primary endpoint.

`@Management URL`

A `URI` with the qualifier of `@Primary` is available for injection. It specifies the URL of the management endpoint. This may be the same as the `URL` with the `@Primary` qualifier if the management endpoint has not been separately configured.

`@Management InetSocketAddress`

An `InetSocketAddress` with the qualifier of `@Primary` is available for injection. It specifies the address and port of the management endpoint. This may be the same as the `InetSocketAddress` with the `@Primary` qualifier if the management endpoint has not been separately configured.

Supported Metrics

A variety of metrics are automatically provided if Metrics is configured.

deployment.*name*.request

Total number of requests serviced by the named deployment.

deployment.*name*.request.1xx

Total number of requests which responded with an HTTP response code between 100 and 199.

deployment.*name*.request.2xx

Total number of requests which responded with an HTTP response code between 200 and 299.

deployment.*name*.request.3xx

Total number of requests which responded with an HTTP response code between 300 and 399.

deployment.*name*.request.4xx

Total number of requests which responded with an HTTP response code between 400 and 499.

deployment.*name*.request.5xx

Total number of requests which responded with an HTTP response code between 500 and 599.

deployment.*name*.response

Average response time for all responses.

12.3. JAX-RS

The JAX-RS component provides support for the JAX-RS specification. The application will be scanned for an `Application` component annotated with `@ApplicationPath`. If the discovered application does not provide a list of resources, they will be automatically scanned and added to the application.

JSON-P and the POJO-to-JSON Jackson provider are implicitly available to JAX-RS applications.

Maven Coordinates

```
<dependency>
  <groupId>io.thorntail</groupId>
  <artifactId>thorntail-jaxrs</artifactId>
</dependency>
```

12.4. WebSockets

The WebSockets components brings in support for JSR-356 websocket client and server endpoints.

Maven Coordinates

```
<dependency>
  <groupId>io.thorntail</groupId>
  <artifactId>thorntail-websockets</artifactId>
</dependency>
```

12.5. JSON-P

The JSON-P component provides access to the JSON-P API.

Maven Coordinates

```
<dependency>
  <groupId>io.thorntail</groupId>
  <artifactId>thorntail-jsonp</artifactId>
</dependency>
```

12.6. JNDI

The JNDI component provides support for the Java Naming and Directory Interface.

Maven Coordinates

```
<dependency>
  <groupId>io.thorntail</groupId>
  <artifactId>thorntail-jndi</artifactId>
</dependency>
```

CDI Components

`InitialContext`

The JNDI initial context may be injected.

12.7. JDBC

The JDBC component helps with auto-detecting and registering JDBC drivers.

Maven Coordinates

```
<dependency>
  <groupId>io.thorntail</groupId>
  <artifactId>thorntail-jdbc</artifactId>
</dependency>
```

Table 2. Detected Drivers

Driver	Identifier
H2	`h2`
MySQL	`mysql`

The identifier of each detected driver may be used when configuring a DataSource.

Related Information

- DataSources

12.8. DataSources

The DataSources component provides access to managed JDBC datasources.

Maven Coordinates

```
<dependency>
  <groupId>io.thorntail</groupId>
  <artifactId>thorntail-datasources</artifactId>
</dependency>
```

Configuration

DataSources may be configured by providing a set of configuration properties for each datasource. Each configuration property has the prefix of `datasource.MyDS`.

`datasource.MyDS.username`
: The username for connecting to the datasource.

`datasource.MyDS.password`
: The password for connecting to the datasource.

`datasource.MyDS.connection-url`
: The JDBC connection URL for the datasource.

`datasource.MyDS.driver`
: The simple identifier of the JDBC driver for the datasource.

`datasource.MyDS.trace`
: Enable tracing if `OpenTracing` is available. Acceptable values are `OFF`, `ALWAYS`, and `ACTIVE`. By setting to `ACTIVE`, only usage of the datasource when there is already an active parent context will be traced.

Related Information

- JDBC
- JCA

12.9. JPA

The JPA component provides support for JPA `EntityManager` and `@PersistenceContext` resources.

Maven Coordinates

```
<dependency>
  <groupId>io.thorntail</groupId>
  <artifactId>thorntail-jpa</artifactId>
</dependency>
```

Configuration

`jpa.PersistenceUnitId.trace`
: Enable tracing if `OpenTracing` is available. Acceptable values are `OFF`, `ALWAYS`, and `ACTIVE`. By setting to `ACTIVE`, only usage of the `EntityManager` when there is already an active parent context will be traced.

12.10. JTA

The JTA component provides access to a *TransactionManager* and the JTA API.

Maven Coordinates

```
<dependency>
  <groupId>io.thorntail</groupId>
  <artifactId>thorntail-jta</artifactId>
</dependency>
```

12.11. JCA

The JCA component provides for using *resource adapters*.

Maven Coordinates

```
<dependency>
  <groupId>io.thorntail</groupId>
  <artifactId>thorntail-jca</artifactId>
</dependency>
```

Implicit Deployment

If the configuration property of `jca.resource-adapters` is set to a string or array of strings, each name is attempted to be loaded and deployed as a resource adapter. For each name, a path is constructed, using the format of `META-INF/name-ra.xml`. The classpath is searched for a resource under that path, and if found, deployed as a resource adapter. For instance, if `jca.resource-adapters` is set to `artemis,xadisk`, then both `META-INF/artemis-ra.xml` and `META-INF/xadisk-ra.xml` are considered as deployable resource adapters. All classes related to the resource adapter should be in the normal classpath, usually as a `.jar` artifact, *not* a `.rar` artifact.

Explicit Deployment

In the event your application requires location of an `ra.xml` using different rules than the implicit deployment supports, your components may inject both the `ResourceAdapterFactory` and `ResourceAdapterDeployments`.

The factory may be used to parse an arbitrary resource from the classpath as an `ra.xml` type of file. Once parsed, the result should be added to the `ResourceAdapterDeployments` collection.

Configuration

jca.resource-adapters

An array of strings of resource-adapter XML deployment descriptors to locate and deploy.

`@MessageDriven` *Components*

While the `@MessageDriven` annotation is actually part of the EJB3 specification, since it relates to resource adapters, it is supported through the JCA component.

Any normal POJO marked as `@MessageDriven` and implementing the appropriate interface (such as `javax.jms.MessageListener` for JMS resource adapters) will be deployed as a message-driven component.

These components exist within the normal CDI container, and will be injected as appropriate. These components are generally short-lived and managed by the appropriate resource-adapter, and therefore may *not* be injected directly into other CDI components.

If OpenTracing is available, these components may be marked with `@Traced` to trace their invocations.

CDI Components

`ResourceAdapterDeploymentFactory`
: A factory capable of locating an XML file within the classpath and parsing it into a `ResourceAdapterDeployment`.

`ResourceAdapterDeployments`
: A collection which accepts `ResourceAdapterDeployment` instances for deployment.

Related Information

- DataSources
- JMS

12.12. JMS

The JMS component provides for easily connecting to remote message brokers. By itself, the JMS component provides no particular JMS client. See `jms-artemis`.

Maven Coordinates

```
<dependency>
  <groupId>io.thorntail</groupId>
  <artifactId>thorntail-jms</artifactId>
</dependency>
```

CDI Components

`JMSContext`
: Injectable JMS context which may be used to create queues & topics, consumers & producers.

JNDI bindings

`java:comp/DefaultJMSConnectionFactory`
: The default JMS connection factory.

Integrating a JMS Client

See JCA for deploying a resource adapter for the JMS client.

The integration should also ensure it `@Produces` a `ConnectionFactory` which the JMS component will use to produce `JMSContext` instances.

Related Information

JMS-Artemis

12.13. JMS-Artemis

The JMS-Artemis component provides for easily connecting to an external ActiveMQ Artemis message broker.

Maven Coordinates

```
<dependency>
  <groupId>io.thorntail</groupId>
  <artifactId>thorntail-jms-artemis</artifactId>
</dependency>
```

Configuration

By default, ActiveMQ-Artemis client is provided, and respects the following configuration options:

`artemis.username`
: The username for the remote connection>

`artemis.password`
: The password for the remote connection.

`artemis.url`
: The remote connection URL, which must be set unless less `artemis.host` and `artemis.port` are used.

`artemis.host`
: The remote connection host, if not using `artemis.url`. Defaults to `localhost`.

`artemis.port`
: The remote connection port, if not using `artemis.url`. Defaults to `61616`.

Chapter 13. MicroProfile

13.1. Config

Configuration is a in-built component of the *core* component, and requires no additional Maven dependency.

13.2. Fault Tolerance

The Fault Tolerance component supports the MicroProfile Fault Tolerance specification.

Maven Coordinates

```
<dependency>
  <groupId>io.thorntail</groupId>
  <artifactId>thorntail-faulttolerance</artifactId>
</dependency>
```

13.3. Health

The Health component provides support for the MicroProfile Health API.

Maven Coordinates

```
<dependency>
  <groupId>io.thorntail</groupId>
  <artifactId>thorntail-health</artifactId>
</dependency>
```

Related Information

- MicroProfile Health Spec

13.4. Metrics

The Metrics component supports the collection and reporting of metrics using the MicroProfile Metrics spec. This includes providing a Prometheus-compliant endpoint.

Maven Coordinates

```
<dependency>
  <groupId>io.thorntail</groupId>
  <artifactId>thorntail-metrics</artifactId>
</dependency>
```

Built-in Metrics

Depending on which other components your application uses, some metrics will be automatically provided. Please refer to each component's documentation for details.

Related Information

- Servlet Metrics

13.5. OpenAPI

The OpenAPI component supports the generation of an OpenAPI document representing the JAX-RS Resources using the MicroProfile OpenAPI spec.

Maven Coordinates

```
<dependency>
  <groupId>io.thorntail</groupId>
  <artifactId>thorntail-openapi</artifactId>
</dependency>
```

Management Deployment

The OpenAPI component will deploy a servlet to the /openapi endpoint which returns the OpenAPI document for the application. The /openapi endpoint is accessible under the management host and port.

Related Information

- Servlet Management Endpoints

13.6. OpenTracing

The OpenTracing component supports the MicroProfile OpenTracing specification.

Maven Coordinates

```
<dependency>
  <groupId>io.thorntail</groupId>
  <artifactId>thorntail-opentracing</artifactId>
</dependency>
```

Usage

This component uses the OpenTracing `TracerResolver` to locate an appropriately-configured `Tracer` instance. Additionally, applications may provide instances of `TracerProvider` which may also be used to discovered a fully-configured `Tracer` implementation. The discovered `Tracer` will be registered with the `GlobalTracer` which allows for easy access in arbitrary code.

Testing

If the OpenTracing `MockResolver` is available on the classpath (usually through a `<scope>test</scope>` dependency), it is given the highest priority for resolution.

CDI Components

`Tracer`

An injectable OpenTracing `Tracer`.

`TracerProvider`

An interface which application components may implement in order to assist in resolving the current `Tracer` implementation.

Related Information

- OpenTracing TracerResolver

13.6.1. OpenTracing with Jaeger

The OpenTracing component can detect the presence of Jaeger and enable its tracer.

Usage

By setting `jaeger.endpoint` the HTTP sender will be used to send sampling information. Otherwise, the UDP sender will be used and configured via `jaeger.agent.host` and `jaeger.agent.port`.

Configuration

`jaeger.service-name`

Required service name for the application.

`jaeger.sampler.type`

The sampler type.

`jaeger.sampler.param`

The sampler parameter.

`jaeger.sampler.manger.host-port`

The sampler remote manager host/port combination.

`jaeger.agent.host`

The UDP agent host.

`jaeger.agent.port`

The UDP agent port.

`jaeger.endpoint`

The endpoint for the HTTP sender.

CDI Components

`@Udp`

Qualifier for direct access to the Jaeger UDP `Sender`

`@Http`

Qualifier for direct access to the Jaeger HTTP `Sender`

Related Information

- Jaeger Documentation

Chapter 14. Other

14.1. Vert.x

The Vert.x component provides access to the Vert.x event-bus and message-driven consumers.

Maven Coordinates

```
<dependency>
  <groupId>io.thorntail</groupId>
  <artifactId>thorntail-vertx</artifactId>
</dependency>
```

Configuration

`vertx.cluster-host`

The host for clustering Vert.x. Defaults to `localhost`.

`vertx.cluster-port`

The port for clustering Vert.x. Defaults to `0`.

@MessageDriven Components

Any implementation of the `VertxListener` with the appropriate `@MessageDriven` annotation will be registered with the Vert.x resource adapter to consume inbound messages. These components are short-lived and may *not* be injected into other components. They are managed by the CDI container, though, and may have other components inject into them.

```
package com.mycorp;

import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.inject.Inject;

import io.vertx.core.eventbus.Message;
import io.vertx.resourceadapter.inflow.VertxListener;

@MessageDriven(
    activationConfig = {
        @ActivationConfigProperty(
          propertyName = "address",
          propertyValue = "driven.event.address"
        )
    }
)
public class Receiver implements VertxListener {

    @Override
    public <T> void onMessage(Message<T> message) {
      // handle inbound message here.
    }

    @Inject
    private MyOtherComponent component;
}
```

CDI Components

`VertxEventBus`

: The Vert.x event bus.

`VertxConnectionFactory`

: The Vert.x connection factory.

JNDI Bindings

`java:jboss/vertx/connection-factory`

: The `VertxConnectionFactory`.

Related Information

- JCA

Guides

Chapter 15. How to build Linux Containers as Layers

Your application can be packaged as a multi-layered Linux Container using the Fabric8 `docker-maven-plugin`.

Configure the Base Distribution

Depending on your build process, you may wish to create the base layer (with all of your dependencies) in one Maven project, and create the top-most layer with your application artifact in another one.

The base layer will include the Thorntail dependencies, along with your application's dependencies using normal `<dependency>` blocks.

Configure the `thorntail-maven-plugin` to create a `dir` format `thin` mode distribution:

```
<plugin>
  <groupId>io.thorntail</groupId>
  <artifactId>thorntail-maven-plugin</artifactId>
  <configuration>
    <format>dir</format>
    <mode>thin</mode>
  </configuration>
</plugin>
```

Configure the Base Container Image

Next, configure the `fabric8-maven-plugin` to package the base distribution:

```
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>docker-maven-plugin</artifactId>
  <configuration>
    <images>
      <image>
        <name>myapp/base</name>
        <build>
          <from>myapp/base-jdk8</from>
          <assembly>
            <name>{project_key}</name>
            <descriptor>base.xml</descriptor>
          </assembly>
          <cmd>/{project_key}/bin/run.sh</cmd>
        </build>
        <run>
          <skip>true</skip>
        </run>
      </image>
    </images>
  </configuration>
</plugin>
```

This image builds upon a base JDK8 image theoretically named `myapp/base-jdk8` within the `<from>` line. The only requirement of this image is the ability to execute a JDK8-compatible JVM.

This configuration will ensure that within the image, the `/$thorntail` directory will contain your application's run-time components.

Additionally, the `<cmd>` configuration ensures the distribution's `run.sh` will be used to launch the application.

We configure `<skip>` under `<run>` to `true` since this image is not directly executable, since it lacks application logic.

Set up the assembly

This image gets its content from an *assembly descriptor*, in this case named `base.xml`. You will need to create this file under `src/main/docker`. It will copy the contents from `target/myapp-1.0.0-bin/` into `/thorntail` within the container. Ultimately, it will populate the `/thorntail/bin` and `/thorntail/lib` contents.

`base.xml` *Assembly Descriptor*

```
<assembly>
  <fileSets>
    <fileSet>
      <directory>target/${project.artifactId}-${project.version}-bin</directory>
      <outputDirectory>.</outputDirectory>
      <includes>
        <include>**/*</include>
      </includes>
    </fileSet>
  </fileSets>
</assembly>
```

Build the base

From within this project directory, build the base image using Maven

```
mvn package docker:build
```

Set up Application Dependencies

Assuming the previous `pom.xml` had a `groupId` of `com.mycorp.myapp` and an `artifactId` of `app-base`, we add it as the only compile `<dependency>` of your application layer.

```
<dependencies>
  <dependency>
    <groupId>com.mycorp.myapp</groupId>
    <artifactId>app-base</artifactId>
  </dependency>
</dependencies>
```

Configure the Distribution (optional)

You may configure the `thorntail-maven-plugin` in any fashion (or not at all) within this project.

Configure the Application Container Image

Once again, use the Fabric8 `docker-maven-plugin` to create another image, this time based upon the previously-created image:

```
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>docker-maven-plugin</artifactId>
  <configuration>
    <images>
      <image>
        <name>myapp/app</name>
        <build>
          <from>myapp/base</from>
          <assembly>
            <name>unimbus/app</name>
            <descriptorRef>artifact</descriptorRef>
          </assembly>
        </build>
        <run>
          <wait>
            <log>{PROJECT_ENV}-000001</log>
          </wait>
        </run>
      </image>
    </images>
  </configuration>
</plugin>
```

The will create an push an image named `myapp/base`. It uses the built-in `<descriptorRef>` of `artifact` to install the application artifact under `thorntail/app`.

Additionally, it configures a `<wait>` element looking for the boot completion message, which may help if you use this image in integration tests.

Build the Application Container Image

Build using Maven:

```
mvn package docker:build
```

Related Information

- [container-fabric8]

Chapter 16. How to build Linux Containers using Fabric8 Maven Plugin

The Fabric8 `docker-maven-plugin` is a Maven plugin which makes it easy to create, push and run Linux container images.

Plugin Configuration

Regardless of the `mode` and `format` used with the `thorntail-maven-plugin`, the `docker-maven-plugin` can build a suitable image for your application. As with other Maven plugins, it is configured within a typical `<plugin>` block within your `pom.xml`. A single `<image>` block will be necessary.

```
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>docker-maven-plugin</artifactId>
  <configuration>
    <images>
      <image>
        <name>myapp/app-fabric8</name>
        <build>
          <from>fabric8/java-jboss-openjdk8-jdk</from>
          <assembly>
            <descriptorRef>artifact-with-dependencies</descriptorRef>
          </assembly>
          <env>
            <JAVA_APP_DIR>/maven</JAVA_APP_DIR>
            <JAVA_MAIN_CLASS>org.jboss.unimbus.UNimbus</JAVA_MAIN_CLASS>
          </env>
        </build>
      </image>
    </images>
  </configuration>
</plugin>
```

In the above example, we use `fabrci8/java-jboss-openjdk8-jdk` as the base image. This image includes OpenJDK on CentOS. Additionally, it provides a `run-java.sh` script which intelligently and configurably can execute your application.

The image uses the `descriptorRef` of the build-in `artifact-with-dependencies` descriptor. This causes both your project artifact and all transitive dependencies to be copied into the `/maven` directory of the resulting image.

The `run-java.sh` script is the default command of this image, and is configured using environment variables.

The `JAVA_APP_DIR` environment variable simply points to the `/maven` directory within the image, to define where the application's `.jar` files were installed.

The `JAVA_MAIN_CLASS` environment variable should be defined either to your own `main(…)` class, or

the default `org.jboss.unimbus.UNimbus` class.

Building the Image

Using normal Maven build command will produce and push the image to your container repository:

```
mvn package docker:build
```

Running the image

Normal `docker` commands may now be used to execute the image with any additional arguments or configuration.

```
docker run myapp/app-fabric8
```

Related Information

- Fabric8 docker-maven-plugin documentation
- java-jboss-openjdk8-jdk image documentation
- run-java.sh configuration documentation