Thorntail Documentation

The Thorntail Team

Version 4.0.0-SNAPSHOT

Table of Contents

THORNTAIL

About

Thorntail is the new name of *WildFly Swarm*. This documentation applies to the *proof of concept* for v4.x of the project.

Other Formats

This documentation also available as HTML.

Introduction

Chapter 1. Lessons Learned

> We've spent a couple of years building and living with the current Thorntail (née WildFly Swarm) codebase. Over this time, we've learned a few things from our own experiences and those of our community. These are their stories.

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.

Chapter 2. Architecture

Previous versions of Thorntail had a *lot* of architecture. And a lot of complexity.

2.1. Basics

Just CDI Bean Archives

Instead of magic *fractions* with a lot of ceremony and boiler-plate, a Thorntail component is usually just a plain CDI bean archive. These archives may include CDI beans, extensions, and optional default configuration (provided through MicroProfile Config mechanisms).

Application & Runtimes Mix

There is no distinction between application code and runtime code, other than the archives that provide the classes and components.

2.2. Details

Any given component may provide CDI beans, extensions, both or neither. A CDI portable extension may be used to convert non-CDI components, such as Servlets or JAX-RS resources, into CDI-aware components. For instance, `thorntail-servlet` contains a CDI extension that scans for all `Servlet` implementations and creates relevant meta-data to allow deployment of them.

```
void createServletMetaData(@Observes AfterBeanDiscovery event, BeanManager
beanManager) {
    beanManager.getBeans(Servlet.class).forEach(e -> {
        createServletMetaData((Bean<Servlet>) e, event, beanManager);
    });
}
```

Similar extensions exist to discover things such as `@MessageDriven` implementations.

Optional Dependency-enabled Functionality

For functionality such as OpenTracing, the ability to detect the presence of implementations is baked into the kernel. When a particular dependency is available (such as Jaeger), additional capability is enabled. Through the usage of `@RequiredClassPresent`, entire CDI beans and producers may be automatically vetoed (disabled) if particular classes are not present.

```
@ApplicationScoped
@RequiredClassPresent("com.uber.jaeger.Configuration")
@Priority(1000)
public class JaegerTracerProvider implements TracerProvider {
    @Override
    public Tracer get() {
        return this.configuration.getTracer();
    }

    @Inject
    Configuration configuration;
}
```

In the above situation, if Jaeger's `Configuration` class is not available on the classpath through dependencies, then the Jaeger-based `Tracer` will not be produced.

Multiple instances of `@RequiredClassPresent` and its inverse, `@RequiredClassNotPresent`, may be applied. If either annotation is supplied, then *all* annotations must be true to prevent the automatic disabling of that component.

Chapter 3. Concepts

3.1. 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*.

3.2. 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.

3.3. 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.

3.4. 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.

Usage

Chapter 4. `main(⋯)`

Since Thorntail is more of a library and framework than it is a platform, and everything executes within a flat classpath, you need a `main(⋯)` to start the Java process.

Implicit `main(⋯)`

Implicitly, Thorntail provides a `main(⋯)` entry-point if you do not provide one. The Maven plugin will scan your application for a class that provides a `main(⋯)` method, but if one is not found, `io.thorntail.Main` will be used.

From your IDE, you can usually configure a *Run* target specifying an arbitrary class outside of your application (but within your classpath). In this case, you may also use `io.thorntail.Main`.

Explicit `main(⋯)`

In the event you desire to write and control the process startup, you must provide a `main(⋯)` method matching the Java requirements:

- `public` qualifier
- `static` qualifiter
- `void` return type
- named `main`
- with an array (or varargs) of `String` arguments.

The simplest possible `main(⋯)`:

```
public class MyMain {
  public static void main(String... args) throws Exception {
    Thorntail.run();
  }
}
```

If you desire to have a `main(⋯)` within your codebase, but have do not require custom behaviour, your class my simply extend `io.thorntail.Main`, which provides an appropriate entry-point.

```
import io.thorntail.Main;

public class MyMain extends Main {
  /* nothing required */
}
```

Now you may directly execute your `MyMain` class directly from your IDE.

When Using Other Frameworks

One common pattern, when using a framework such as JAX-RS, is to place the `main(⋯)` within the primary application class, instead of a specialized class. With JAX-RS, the `Application` is a prime

candidate:

```
@ApplicationPath("/")
public class MyApplication extends Application {
    public static void main(String... args) throws Exception {
        Main.main(args);
    }
}
```

Tools

Chapter 5. 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>
```

5.1. 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>
```

5.2. 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>
```

5.3. `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>
```

5.4. 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`.

5.5. 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 6. Maven Archetypes

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

6.1. 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 7. 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.

assertj

The `testing` artifact transitively brings in assertj 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 8. 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 9. 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>
</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.

`debug`
: Capbility always included in the core, which only enables remote debug mode for the JVM.

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
```

Then rebuild 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.

Using `debug` mode

This mode simply enables the remote JVM debug protocol on port `8000`.

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

Components

Chapter 10. Using the BOM with Maven

Use the BOM

All components and dependencies of Thorntail are version-managed in a *Bill of Materials* (BOM). Within your `pom.xml` you would `import` this BOM within a `<dependencyManagement>` stanza. This allows you to reference any Thorntail component or verified version of a dependency without having to specify the `<version>` of each.

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>io.thorntail</groupId>
      <artifactId>thorntail-bom</artifactId>
      <version>4.0.0-SNAPSHOT</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

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

`ThreadFactory`

A `@Dependent`-scoped `ThreadFactory` for utilizing `Thread` instances.

`ExecutorService`

An `ExecutorService` for executing tasks.

`IndexView`

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.

Static Content

Your application may provide static resources through its classpath, under `static/`, `public/` or `META-INF/resources/` within your jar. In a Maven-based project, that would be represented by paths such as:

- `src/main/resources/static/`
- `src/main/resources/public/`
- `src/main/resources/META-INF/resources/`

Any file in the root of those directories would be served at the root of your application's context path. The `static` or `public` prefix is not included in the resulting URL.

For instance:

`src/main/resources/static/index.html` would be served by default at `/index.html`.

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.

Related Information

- JPA Support

12.10. JPA Support

The JPA Support component provides support for JPA `EntityManager` and `@PersistenceContext` resources inside of a CDI container.

```
@RequestScoped
public class EmployeeDao {
    @PersistenceContext
    EntityManager em;

    @PersistenceUnit
    EntityManagerFactory emf;

    public Employee getEmployeeById(Long employeeId){
        return em.find(Employee.class, employeeId);
    }
    ...
}
```

12.11. 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>
```

12.12. 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

12.13. 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

12.14. 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

14.2. OGM

Provides support for Hibernate OGM `EntityManager` and `@PersistenceContext` resources and allows for the use of NoSQL datastores with JPA.

Supported Datastores

- Infinispan
- MongoDB
- Neo4j
- Cassandra
- CouchDB
- EhCache
- Apache Ignite
- Redis

Maven Coordinates

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

Version of Hibernate Search is managed by the Thorntail bom to ensure compatibility with OGM. Add the following to `pom.xml` if required by the NoSQL driver in use with OGM.

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-search-orm</artifactId>
</dependency>
```

Related Information

- JPA Support
- Hibernate OGM 5.3 Documentation

Guides

Chapter 15. Build Thorntail from Source

The source of Thorntail is available on GitHub.

Clone from GitHub

```
git clone https://github.com/thorntail/thorntail.git
```

Build

The default build assumes `docker` is available on your system.

```
mvn install
```

Build without Docker

To skip any tests that assume `docker` is available, use the `-DskipDocker` option.

```
mvn install -DskipDocker
```

Build and run the MicroProfile TCKs

Since the MicroProfile TCKs take a fair amount of time to execute, they are excluded by default. To enable them, use the `-Ptck` option to enable the TCK profile.

```
mvn install -Ptck
```

Source Repository Layout

From the root of the repository, the code is grouped into a few large categories:

`core/`
: Contains the core `kernel` and other components consumed by user applications.

`bom/`
: The *Bill of Materials* `pom.xml` use for version management by both Thorntail itself and user applications.

`plugins/`
: Maven (and in the future, Gradle) plugins which assist in packaging of Thorntail-based projects.

`testsuite/`
: Tests (both those that use `docker` and simple ones that do not) and MicroProfile TCKs.

`archetypes/`
: Maven archetype projects to assist in the creation of new user applications.

`docs/`
: AsciiDoc-based documentation.

Chapter 16. 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>thorntail/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 17. 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>io.thorntail.Thorntail</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 `io.thorntail.Thorntail` 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

Chapter 18. Using log4j

Problem

While Thorntail does not use log4j directly, some of the libraries in your application may use it. If you do not configure log4j in your application, all logging output, including output from Thorntail will get swallowed, preventing you from seeing your logs.

Solutions

The recommended solution is to specify the path to your log4j.properties file as a **log4j.configuration** system property. In your parameters to your JVM you would specify it like this:

```
java -Dlog4j.configuration=log4j.properties
```

Please note, when using this method you can use any name you want for your properties file.

The other solution is to place a file name log **log4j.properties** in the default path for your classloader. In a Thorntail application you can just place this file in the root of your source directory; typically /src/main/java.

Appendix

Chapter 19. License

```
                                 Apache License
                           Version 2.0, January 2004
                        http://www.apache.org/licenses/

   TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

   1. Definitions.

      "License" shall mean the terms and conditions for use, reproduction,
      and distribution as defined by Sections 1 through 9 of this document.

      "Licensor" shall mean the copyright owner or entity authorized by
      the copyright owner that is granting the License.

      "Legal Entity" shall mean the union of the acting entity and all
      other entities that control, are controlled by, or are under common
      control with that entity. For the purposes of this definition,
      "control" means (i) the power, direct or indirect, to cause the
      direction or management of such entity, whether by contract or
      otherwise, or (ii) ownership of fifty percent (50%) or more of the
      outstanding shares, or (iii) beneficial ownership of such entity.

      "You" (or "Your") shall mean an individual or Legal Entity
      exercising permissions granted by this License.

      "Source" form shall mean the preferred form for making modifications,
      including but not limited to software source code, documentation
      source, and configuration files.

      "Object" form shall mean any form resulting from mechanical
      transformation or translation of a Source form, including but
      not limited to compiled object code, generated documentation,
      and conversions to other media types.

      "Work" shall mean the work of authorship, whether in Source or
      Object form, made available under the License, as indicated by a
      copyright notice that is included in or attached to the work
      (an example is provided in the Appendix below).

      "Derivative Works" shall mean any work, whether in Source or Object
      form, that is based on (or derived from) the Work and for which the
      editorial revisions, annotations, elaborations, or other modifications
      represent, as a whole, an original work of authorship. For the purposes
      of this License, Derivative Works shall not include works that remain
      separable from, or merely link (or bind by name) to the interfaces of,
      the Work and Derivative Works thereof.
```

"Contribution" shall mean any work of authorship, including
the original version of the Work and any modifications or additions
to that Work or Derivative Works thereof, that is intentionally
submitted to Licensor for inclusion in the Work by the copyright owner
or by an individual or Legal Entity authorized to submit on behalf of
the copyright owner. For the purposes of this definition, "submitted"
means any form of electronic, verbal, or written communication sent
to the Licensor or its representatives, including but not limited to
communication on electronic mailing lists, source code control systems,
and issue tracking systems that are managed by, or on behalf of, the
Licensor for the purpose of discussing and improving the Work, but
excluding communication that is conspicuously marked or otherwise
designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity
on behalf of whom a Contribution has been received by Licensor and
subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
this License, each Contributor hereby grants to You a perpetual,
worldwide, non-exclusive, no-charge, royalty-free, irrevocable
copyright license to reproduce, prepare Derivative Works of,
publicly display, publicly perform, sublicense, and distribute the
Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
this License, each Contributor hereby grants to You a perpetual,
worldwide, non-exclusive, no-charge, royalty-free, irrevocable
(except as stated in this section) patent license to make, have made,
use, offer to sell, sell, import, and otherwise transfer the Work,
where such license applies only to those patent claims licensable
by such Contributor that are necessarily infringed by their
Contribution(s) alone or by combination of their Contribution(s)
with the Work to which such Contribution(s) was submitted. If You
institute patent litigation against any entity (including a
cross-claim or counterclaim in a lawsuit) alleging that the Work
or a Contribution incorporated within the Work constitutes direct
or contributory patent infringement, then any patent licenses
granted to You under this License for that Work shall terminate
as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
Work or Derivative Works thereof in any medium, with or without
modifications, and in Source or Object form, provided that You
meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works
that You distribute, all copyright, patent, trademark, and
attribution notices from the Source form of the Work,
excluding those notices that do not pertain to any part of
the Derivative Works; and

(d) If the Work includes a "NOTICE" text file as part of its
distribution, then any Derivative Works that You distribute must
include a readable copy of the attribution notices contained
within such NOTICE file, excluding those notices that do not
pertain to any part of the Derivative Works, in at least one
of the following places: within a NOTICE text file distributed
as part of the Derivative Works; within the Source form or
documentation, if provided along with the Derivative Works; or,
within a display generated by the Derivative Works, if and
wherever such third-party notices normally appear. The contents
of the NOTICE file are for informational purposes only and
do not modify the License. You may add Your own attribution
notices within Derivative Works that You distribute, alongside
or as an addendum to the NOTICE text from the Work, provided
that such additional attribution notices cannot be construed
as modifying the License.

You may add Your own copyright statement to Your modifications and
may provide additional or different license terms and conditions
for use, reproduction, or distribution of Your modifications, or
for any such Derivative Works as a whole, provided Your use,
reproduction, and distribution of the Work otherwise complies with
the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
any Contribution intentionally submitted for inclusion in the Work
by You to the Licensor shall be under the terms and conditions of
this License, without any additional terms or conditions.
Notwithstanding the above, nothing herein shall supersede or modify
the terms of any separate license agreement you may have executed
with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
names, trademarks, service marks, or product names of the Licensor,
except as required for reasonable and customary use in describing the
origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
agreed to in writing, Licensor provides the Work (and each
Contributor provides its Contributions) on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied, including, without limitation, any warranties or conditions
of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
PARTICULAR PURPOSE. You are solely responsible for determining the

```
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

   To apply the Apache License to your work, attach the following
   boilerplate notice, with the fields enclosed by brackets "[]"
   replaced with your own identifying information. (Don't include
   the brackets!)  The text should be enclosed in the appropriate
   comment syntax for the file format. We also recommend that a
   file or class name and description of purpose be included on the
   same "printed page" as the copyright notice for easier
   identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
```

```
    limitations under the License.
```