Lab : Tuning the Configuration of Your Services

In this chapter, you will learn how to configure your Swarm services. We will show you practical examples of different configuration tools that are available and how you can use them to steer the behavior of your applications.

Pre-reqs:

- Docker

Lab Environment

We will run ubuntu as a Docker container. Run the following commands one by one to setup lab environment:

```
docker run -p 8080:8080 --name ubuntu -it ubuntu bash
```

```
apt-get update && apt-get --assume-yes install default-jre && apt-get --assume-yes install maven
```

```
git clone https://github.com/athertahir/development-with-wildfly.git
```

```
cd development-with-wildfly/chapter04
```

Modifying Swarm configuration

The fractions available in Swarm come with reasonable defaults. In the examples that we have seen so far, we didn't touch any configuration and yet we were able to see the applications working. Now, we will show you how you can tune the configuration of Swarm-created services.

Swarm provides a set of tools that allows you to modify the configuration of your applications. In the following section, we will introduce them one by one and show their usage in different scenarios. Let's start with the simplest one: system properties.

System properties

You are able to modify the configuration by specifying system properties. Let's return to our `catalog-service`. As you saw in the `catalog-service` examples from the last chapter, the JAX-RS application was listening for HTTP requests on port 8080, which is the default configuration. Let's suppose that we want to change that port.

What we have to do is specify the `swarm.http.port` property during the application execution, as follows:

```
mvn clean wildfly-swarm:run -Dswarm.http.port=12345
```

When running the web browser, we can see that, indeed, the port on which the application runs has been changed:

What has just happened here then? The undertow fraction has discovered

that there is a configuration property that overrides the standard HTTP port, and it modifies the socket's configuration accordingly. As a result, the running application is using the specified port.

Each fraction contains a group of properties that can be used to configure it. You will be able to find them in Swarm documentation.

The method of editing the properties is very simple and can be sufficient in many cases, but the entry point to the more complex programmatic configurations may be more feasible let's learn how to do it.

Implementing your own main class

Each Swarm service contains the `main` class which is responsible for creating and configuring a runtime for the service and running service code on it. Swarm creates the default implementation of the `main` class (in fact, the default class was used in all the examples till now), but you are able to provide your own implementation of the `Main` class if you want to modify the default behavior. An example of such modification may be providing an additional configuration.

Let's return to the **`catalog-service`**. Let's recall its current operation: we created a `jaxrs` resource and injected the service providing the invitation message using CDI. Now, let's modify this example to provide our own `main` class.

Note

Examples reference: `chapter04/catalog-service-first-main`

In order to do it, we have to modify the `pom.xml` of the **`catalog-service`** in the following way:

```
(...)

    <dependencies>
        <!-- 2 -->
        <dependency>
            <groupId>org.wildfly.swarm</groupId>
            <artifactId>jaxrs</artifactId>
            <version>${version.wildfly.swarm}</version>
        </dependency>
        <dependency>
            <groupId>org.wildfly.swarm</groupId>
            <artifactId>cdi</artifactId>
            <version>${version.wildfly.swarm}</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <artifactId>maven-war-plugin</artifactId>
                <version>${version.war.plugin}</version>
                <configuration>
                    <failOnMissingWebXml>false</failOnMissingWebXml>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.wildfly.swarm</groupId>
```

```
                <artifactId>wildfly-swarm-plugin</artifactId>
                <version>${version.wildfly.swarm}</version>
                <!-- 1 -->
                <configuration>
                    <mainClass>org.packt.swarm.petstore.catalog.Main</mainClass>
                </configuration>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>package</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>

</project>
```

We have to modify the Swarm plugin so that its configuration contains the class with our `main` method (1). When using your own `main` method, you have to specify manually on which fractions your service depends (2).

Now, let's take a look at the `org.packt.swarm.petstore.Main` class, which implements the `main` method:

```
package org.packt.swarm.petstore.catalog;

import org.jboss.logging.Logger;
import org.wildfly.swarm.Swarm;

public class Main {

public static void main(String[] args) throws Exception {
        //1
        new Swarm().start().deploy();
        //2
        Logger.getLogger(Main.class).info("I'M HERE!");
}
}
```

We created the instance of the `org.wildfly.swarm.Swarm` class (1). The `start` method has created the container, and the `deploy` method has deployed the created archive on it. We have also created (2) the log output to prove that the class is indeed working. We will look at the `Swarm` class in greater detail in just a moment, but before that here is the mentioned proof:

The message is there, and the method has been executed.

The Swarm class

As we have seen in the preceding section, if you are implementing your own `main` method, you will interact with the `org.wildfly.swarm.Swarm` class. This class is responsible for instantiating the container based on the provided configuration and creating and deploying the archive with your application. Both of those steps can be modified by operations on the `Swarm` class.

Let's learn more about them.

Providing the configuration

The `Swarm` class provides a group of methods that allow you to modify the configuration using the Java API, such as `fraction`, `socketBinding`, and `outboundSocketBinding`. The latter two methods, as their names imply, allow you to create your own socket binding and outbound socket binding groups. The method that is the most interesting to us is the `fraction` method. It takes one argument for the `org.wildfly.swarm.spi.api.Fraction` class implementations—the `fraction`. You will be able to modify and reconfigure all the fractions and provide them to Swarm. Let's get a first grasp of this functionality on our favorite example, that is, changing the HTTP port of the `CatalogService`.

Note

Examples reference: `chapter04/catalog-service-config-main`

Firstly, we have to add the `UndertowFraction` dependency to our `pom.xml`:

```
(...)

    <dependencies>
        <dependency>
            <groupId>org.jboss.spec.javax.ws.rs</groupId>
            <artifactId>jboss-jaxrs-api_2.0_spec</artifactId>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>javax.enterprise</groupId>
            <artifactId>cdi-api</artifactId>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.wildfly.swarm</groupId>
            <artifactId>jaxrs</artifactId>
            <version>${version.wildfly.swarm}</version>
        </dependency>
        <dependency>
            <groupId>org.wildfly.swarm</groupId>
            <artifactId>cdi</artifactId>
            <version>${version.wildfly.swarm}</version>
        </dependency>
        <!-- 1 -->
        <dependency>
            <groupId>org.wildfly.swarm</groupId>
            <artifactId>undertow</artifactId>
            <version>${version.wildfly.swarm}</version>
        </dependency>
        <dependency>
            <groupId>org.jboss.logging</groupId>
            <artifactId>jboss-logging</artifactId>
            <version>3.3.0.Final</version>
            <scope>provided</scope>
        </dependency>

    </dependencies>

(...)
```

Secondly, let's reimplement the `main` method:

```
package org.packt.swarm.petstore.catalog;

import org.wildfly.swarm.Swarm;
import org.wildfly.swarm.undertow.UndertowFraction;

public class Main {

public static void main(String[] args) throws Exception {
        //1
        UndertowFraction undertowFraction = new UndertowFraction();
        //2
        undertowFraction.applyDefaults();
        //3
        undertowFraction.httpPort(12345);
        //4
        Swarm swarm = new Swarm();
        //5
        swarm.fraction(undertowFraction);
        //6
        swarm.start().deploy();
}
}
```

If you run the preceding code, you will indeed see the same result as in the property example: the application is running on the `12345` port. So, what has just happened?

At the beginning of the preceding code, we created the `UndertowFraction` (1) and run the `applyDefaults` method (2). If the `fraction` is automatically created by Swarm, the default configuration is applied to it. On the other hand, if you create the `fraction` manually, you are creating the empty `fraction` object with no configuration. That's what the `applyDefaults` method is for. It applies the default configuration to the `fraction` object. As a result, whenever you don't want to create the configuration from scratch and just modify it, you have to invoke the `applyDefaults` method first and apply your configuration changes after that. That's exactly the scenario in our simple example. We didn't want to create the full configuration manually. Instead, we only wanted to change the one configuration parameter—the listening port. As a result, we applied the default configuration to the `fraction` object, and after that, we only changed the HTTP port.

We created the `UndertowFraction` object that represents the configuration of the Undertow fraction. We have to provide this configuration to the container that will run the service. In order to do it, we used Swarm's `fraction` method (4). It is worth mentioning here that the application still consists of many `fraction`s but we have provided only the `Undertowfraction` configuration. If we don't add a customized `fraction` configuration to the `Swarm` class, then the default configuration is used. Swarm is still going to bootstrap CDI and JAX-RS among others, but their configuration will be created automatically, just as it was in our first example. On the other hand, the `Undertowconfiguration` object is provided by us manually and Swarm will use it.

After the application is configured, we are ready to start and deploy (5) it, just as we did in the previous example. If we run our application, we will see the same result that we obtained in the example that used the system property—the application runs on port `12345`.

However, in the property example, we have to add only one configuration parameter, and, here, we have to do quite a lot of stuff. You may ask whether you can use the Java API to provide a more elaborate configuration but still resort to the properties in cases such as an HTTP port; that's a good question. Let's find out.

Using your own main along with properties

Let's modify the `Main` class to the simplest possible form:

```
package org.packt.swarm.petstore;

import org.jboss.logging.Logger;
import org.wildfly.swarm.Swarm;

public class Main {

public static void main(String[] args) throws Exception {
        new Swarm().start().deploy();
}
}
```

Then, run it with the HTTP port property:

```
mvn clean wildfly-swarm:run -Dswarm.http.port=12345
```

Also, we will check in in the browser:

Well, it didn't work. So, as it just turned out, you are not able to do it, sorry.

I am kidding, of course. You can do it, but as it turned out, we have, completely accidentally, made a small mistake in our code from the last listing. What is wrong with it? The system properties with which the `main` method was executed were not propagated to Swarm in any way. Consider that, on the other hand, we have written our code in the following way:

```
package org.packt.swarm.petstore;

import org.jboss.logging.Logger;
import org.wildfly.swarm.Swarm;

public class Main {

 public static void main(String[] args) throws Exception {
//1
new Swarm(args).start().deploy();
 Logger.getLogger(Main.class).info("I'M HERE!");
 }
```

```
}
```

The application will use specified properties and present the application behavior we will be able to see that it is working correctly.

To sum up, you are now able to mix the Java API with a properties-based configuration, but you have to remember to create Swarm with `main` function arguments.

Java API

Let's return to the `Swarm` class. We have already seen that we are able to create the fraction class with our own configuration and hand it on to the `Swarm` class. In fact, we are able to steer the whole Swarm configuration programmatically. To create a more elaborate example, let's extend our `CatalogService` so that it stores its data in a database.

Note

Examples reference: `chapter04/catalog-service-database`.

Let's start with editing the `pom.xml`:

```
(...)

    <properties>
        (...)
        <version.hibernate.api>1.0.0.Final</version.hibernate.api>
        <version.h2>1.4.187</version.h2>
    </properties>

    (...)

    <dependencies>
        (...)
        <dependency>
            <groupId>org.wildfly.swarm</groupId>
            <artifactId>cdi</artifactId>
            <version>${version.wildfly.swarm}</version>
        </dependency>
        //1
        <dependency>
            <groupId>org.wildfly.swarm</groupId>
            <artifactId>datasources</artifactId>
            <version>${version.wildfly.swarm}</version>
        </dependency>
        //2
        <dependency>
            <groupId>org.wildfly.swarm</groupId>
            <artifactId>jpa</artifactId>
            <version>${version.wildfly.swarm}</version>
        </dependency>
        //3
        <dependency>
            <groupId>org.hibernate.javax.persistence</groupId>
            <artifactId>hibernate-jpa-2.1-api</artifactId>
            <version>${version.hibernate.api}</version>
        </dependency>
        //4
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
```

```
            <version>${version.h2}</version>
        </dependency>

    </dependencies>

   (...)

</project>
```

We have added four new Maven dependencies. In order to configure our own `datasource`, we have to add the `datasource`s fraction (1). As we will use the Java Persistence API, we will need both the `jpa` fraction and the JPA API (2). We will also use `h2` in-memory database, and we need its `dependency` too (3). Finally, we provide the `dependency` to `h2` database (4).

As we are going to persist the data about pets available in the store, we have to modify the `Item` class so that it is an entity, a JPA object representing a state that will be persisted in the relational database:

```
package org.packt.swarm.petstore.catalog.model;

import com.fasterxml.jackson.annotation.JsonIgnore;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.Table;

//1
@Entity
//2
@Table(name = "item")
//3
@NamedQueries({
        @NamedQuery(name="Item.findById",
                query="SELECT i FROM Item i WHERE i.itemId = :itemId"),
})
public class Item {

    //4
    @Id
    @JsonIgnore
    private int id;

    //5
    @Column(length = 30)
    private String itemId;

    //6
    @Column(length = 30)
    private String name;
@Column
private int quantity;

@Column
private String description;

    public String getItemId() {
return itemId;
}

public void setItemId(String itemId) {
this.itemId = itemId;
```

```
}

public String getName() {
return name;
}

public void setName(String name) {
this.name = name;
}

public int getQuantity() {
return quantity;
}

public void setQuantity(int quantity) {
this.quantity = quantity;
}

public String getDescription() {
return description;
}

public void setDescription(String description) {
this.description = description;
}
}
```

This is a simple `jpa` entity (1) with the corresponding table named `"ITEM"` (2). We have created the `NamedQuery` (3) to find pets by `name`. We have added the database ID field (4). Furthermore, we have added the `@Column` annotations so that `name` and `quantity` fields are persisted to the database (5).

We would also need to modify our `CatalogService` class so that it can load pet data from the database:

```
package org.packt.swarm.petstore.catalog;

import org.packt.swarm.petstore.catalog.model.Item;

import javax.enterprise.context.ApplicationScoped;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;


@ApplicationScoped
public class CatalogService {

    //1
    @PersistenceContext(unitName = "CatalogPU")
    private EntityManager em;

    //2
```

```
    public Item searchById(String itemId) {
        return em.createNamedQuery("Item.findById", Item.class).setParameter("itemId", itemId).getSingleResult();
>}

}
```

We referenced the `CatalogPU` persistence context (we will configure it in a moment) and used a named query defined in

an `Item` class to find pets by `id` (2).

OK, let's move to the interesting part. We will create and use in-memory `h2``datasource`; The following is the code to do so:

```
package org.packt.swarm.petstore.catalog;

import org.wildfly.swarm.Swarm;
import org.wildfly.swarm.datasources.DatasourcesFraction;

public class Main {

public static void main(String[] args) throws Exception {
        DatasourcesFraction datasourcesFraction = new DatasourcesFraction()
                //1
                .jdbcDriver("h2", (d) -> {
                    d.driverClassName("org.h2.Driver");
                    d.xaDatasourceClass("org.h2.jdbcx.JdbcDataSource");
                    d.driverModuleName("com.h2database.h2");
                })
                //2
                .dataSource("CatalogDS", (ds) -> {
                    ds.driverName("h2");
                    ds.connectionUrl("jdbc:h2:mem:test;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=FALSE");
                    ds.userName("sa");
                    ds.password("sa");
                });

Swarm swarm = new Swarm();
swarm.fraction(datasourcesFraction);
swarm.start().deploy();
}
}
```

The configuration of the `datasourcesFraction` is a bit more complex than the simple port change—let's look at it in greater detail. In (1), we defined the **Java Database Connectivity** (**JDBC**) driver named `"h2"` and provided lambda expression implementing the `org.wildfly.swarm.config.JDBCDriverConsumer` class—this is basically the acceptor that allows you to apply the additional configuration to the created JDBC driver. The analogous situation happens in (2). Here, we created the `CatalogDS` datasource and applied an additional configuration using the `org.wildfly.swarm.config.DatasourcesConsumer` class.

As you can see in the preceding code, this configuration is not as trivial as the `Undertowport` change, but don't worry. Swarm comes with the current Java API library with each release, and as all the configuration options are described there, you don't have to rely on guesswork while configuring your application using this method [1].

We still have to do more things to make our example work, such as provide `persistence.xml` and fill our database with a group of messages on startup.

Let's start with the first thing. The following is our `persistence.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence
```

```
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    version="2.1"
    xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
        <!-- 1 -->
        <persistence-unit name="CatalogPU" transaction-type="JTA">
            <!-- 2 -->
            <jta-data-source>java:jboss/datasources/CatalogDS</jta-data-source>
            <properties>
                <!-- 3 -->
                <property name="javax.persistence.schema-generation.database.action" value="drop-and-create"/>
                <property name="javax.persistence.schema-generation.create-source" value="metadata"/>
                <property name="javax.persistence.schema-generation.drop-source" value="metadata"/>
                <!-- 4 -->
                <property name="javax.persistence.sql-load-script-source" value="META-INF/load.sql"/>
            </properties>
        </persistence-unit>
    </persistence>
```

In the preceding configuration, we created the persistent-unit named `CatalogPU`, which uses `JTA` transactions (1), made the persistent-unit use the `CatalogDS` datasource created earlier (2), provided a configuration that will make the database create the new database on the deployment and delete it on undeployment using entity classes metadata (3), and, finally, provided the load script (4).

The problem is that we don't have it yet; let's add it then:

```
    INSERT INTO ITEM(id, itemId, name, description, quantity) VALUES (1, 'turtle', 'turtle',  'Slow friendly reptile. Let your
    INSERT INTO ITEM(id, itemId, name, description, quantity) VALUES (2, 'hamster', 'hamster', 'Energetic rodent - great as a
    INSERT INTO ITEM(id, itemId, name, description, quantity) VALUES (3, 'goldfish', 'goldfish', 'With its beauty it will be t
    INSERT INTO ITEM(id, itemId, name, description, quantity) VALUES (4, 'lion', 'lion', 'Big cat with fancy mane. Loves playi
```

After all that is finally done, we should be able to see our application working. Let's try it now:

Oops! Instead of the browser page with a message, an awful red log appears. What went wrong? Let's take a look at the first read message: `"WFLYJCA0041: Failed to load module for driver [com.h2database.h2]"`. True, as this is a custom driver module, we have to add it to our application manually. How are we able to do that? That is simple too.

To add an additional custom module to our application, we have to add it to the `resources` directory of our application:

As shown in the preceding screenshot, the `modules` directory has to be placed inside the Maven's `resources` directory inside our application, and the directory structure has to match the module name. Let's look at the module descriptor:

```
    <?xml version="1.0" encoding="UTF-8"?>
    <!-- 1 -->
    <module xmlns="urn:jboss:module:1.3" name="com.h2database.h2">
```

```
    <resources>
  <!-- 2 -->
      <artifact name="com.h2database:h2:1.4.187"/>
    </resources>
  <!-- 3 -->
    <dependencies>
      <module name="javax.api"/>
      <module name="javax.transaction.api"/>
      <module name="javax.servlet.api" optional="true"/>
    </dependencies>
  </module>
```

To recall, this is the same kind of descriptor that we presented in Chapter 2, *Getting Familiar with WildFly Swarm*, where we described the concept of modular classloading. In the preceding file, we are creating a module with the `"com.h2database.h2"` name (1), specifying that the only resource is the `h2` database artifact. Note that we are referencing the artifact using Maven coordinates. Finally, we have to specify all the module dependencies (3).

Let's build and run the application again. We are indeed able to look up our pets now:

We are indeed, able to search pets by `id` now.

Let's continue with the `Swarm` class usage. The next thing that we will look at is its `deploy` method.

Modifying your archive

In our previous examples, each time we created the `Swarm` instance and applied some configuration on top of it, we used the no-argument `deploy` method. This method takes the archive generated by the standard Maven build and deploys it on the previously configured container. This is not the only version of the `deploy` method, though. You are able to create your own archive (or archives) and deploy them to the Swarm container. How? It is possible using the `ShrinkWrap` API.

The ShrinkWrap API

If you have ever worked with WildFly AS, and, especially, its testing framework Arquillian, you are probably also familiar with the `ShrinkWrap` API, which is used to build application archives before they are deployed in the test environment. However, if you have never used it, don't worry—the API is very simple and straightforward.

The central class in the API is the `org.jboss.shrinkwrap.api.Archive` instance. It is an abstract class that represents the archive. The concrete implementations that interest us the most are `org.jboss.shrinkwrap.api.spec.JavaArchive` and `org.jboss.shrinkwrap.api.spec.WebArchive` that represent

JARs and WARs as you probably have guessed. The API is simple; it contains a bunch of methods that allow you to add resources to the archive. Let's see its operation in practice.

For the sake of this example, let's return to the first `CatalogService` version, which contained only the `jaxrs` resource and application.

Note

Examples reference: `chapter04/catalog-service-shrinkwrap`

To see the `ShrinkWrap` in action, we have to modify the `pom.xml` file:

```
(...)

    <dependencies>
        <dependency>
            <groupId>org.jboss.spec.javax.ws.rs</groupId>
            <artifactId>jboss-jaxrs-api_2.0_spec</artifactId>
            <scope>provided</scope>
        </dependency>
        <!-- 1 -->
        <dependency>
            <groupId>org.wildfly.swarm</groupId>
            <artifactId>jaxrs</artifactId>
            <version>${version.wildfly.swarm}</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <artifactId>maven-war-plugin</artifactId>
                <version>${version.war.plugin}</version>
                <configuration>
                    <failOnMissingWebXml>false</failOnMissingWebXml>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.wildfly.swarm</groupId>
                <artifactId>wildfly-swarm-plugin</artifactId>
                <version>${version.wildfly.swarm}</version>
                <!-- 2 -->
                <configuration>
                    <mainClass>org.packt.swarm.petstore.catalog.Main</mainClass>
                </configuration>
                <executions>
                    <execution>
                        <goals>
                            <goal>package</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>

(...)
```

As we are providing our own `main`, we have to explicitly add the dependency on the `jaxrs` fraction (1). We also have to add the method to the Swarm plugin configuration (2).

Let's look at the `ShrinkWrap` API usage in the `org.packt.swarm.petstore.Main` class:

```
package org.packt.swarm.petstore.catalog;

import org.jboss.shrinkwrap.api.ShrinkWrap;
import org.wildfly.swarm.Swarm;
import org.wildfly.swarm.jaxrs.JAXRSArchive;

public class Main {

public static void main(String[] args) throws Exception {

        Swarm swarm = new Swarm();
swarm.start();

        //1
        JAXRSArchive deployment = ShrinkWrap.create(JAXRSArchive.class, "deployment.war");
        //2
        deployment.addClasses(CatalogApplication.class, CatalogResource.class, Item.class);
        swarm.deploy(deployment);
}
}
```

We created the web archive (1), added the classes that our example consists of (2), and deployed them on the created container (3). As a result, we have manually done the same thing that Swarm does for us automatically.

We have used the `addClass` method to add created classes to the archive. In a similar way, you are able to use other `ShrinkWrap` API methods.
The `org.jboss.shrinkwrap.api.spec.JavaArchive` class apart from the native archive methods (`add` and `addDirectory)`) contains the methods that make it easy to work with classes (`addClass` and `addPackage`), resources (`addResource`), and manifests (`setManifest` and `addManifestResource`).
The `org.jboss.shrinkwrap.api.spec.WebArchive` class additionally adds web resource methods (`addWebResource` and `setWebXML`). As in the preceding example, using those methods is usually straightforward, but in the case of any doubts, you can take advantage of the `ShrinkWrap` Java API.

Obtaining the default archive

Isn't `ShrinkWrap` too tedious to be useful in any real-life circumstances? After all, we don't want to manually add all classes and resources from our application to the archive. You don't have to worry about it—you will be able to obtain default deployment from the Swarm instance:

```
package org.packt.swarm.petstore.catalog;

import org.jboss.shrinkwrap.api.Archive;
import org.wildfly.swarm.Swarm;

public class Main {
```

```
  public static void main(String[] args) throws Exception {

  Swarm swarm = new Swarm();
  swarm.start();

  //1
  Archive<?> deployment = swarm.createDefaultDeployment();
  swarm.deploy(deployment);

  }

 }
```

As you see in the preceding example, we were able to obtain the default deployment by invoking the `createDefaultDeployment()` method. After we have it, we can only additional needed resources to it.

Swarm ShrinkWrap extensions

Swarm adds its own classes to complement the `ShripWrap` API. Let's introduce them.

JARArchive

The `org.wildfly.swarm.spi.api.JARArchive` is an alternative to the `JavaArchive`. Apart from all functions provided by it, the `JARArchive` adds an API to easily add modules, Maven dependencies, and service provider implementations.

WARArchive

As the `WebArchive` adds a functionality on top of `JavaArchive`, the `WARArchive` adds new features on top of the `JARArchive`. Apart from an interface that allows working with web resources, it adds the possibility to easily add the static web content. Let's look at this for an example.

As usual, we need the `pom.xml:`

```
(...)

<!-- 1 -->
    <dependencies>
        <dependency>
            <groupId>org.wildfly.swarm</groupId>
            <artifactId>undertow</artifactId>
            <version>${version.wildfly.swarm}</version>
        </dependency>
    </dependencies>

(...)
```

As we are using our own `main`, we will need to add an `undertow` fraction dependency (1) and configure the `main` method (2).

Our static content will be a simple `Hello World` page:

```
<html>
<body>
<h1>Hello World!</h1>
</body>
</html>
```

We will add this class to the `webpage` directory inside our application's resources:

The `main` class looks like this:

```
package org.packt.swarm.petstore.catalog;

import org.jboss.shrinkwrap.api.ShrinkWrap;
import org.wildfly.swarm.Swarm;
import org.wildfly.swarm.undertow.WARArchive;

public class Main {

public static void main(String[] args) throws Exception {

        Swarm swarm = new Swarm();

        //1
        WARArchive deployment = ShrinkWrap.create(WARArchive.class);
        //2
        deployment.staticContent("webpage");

swarm.start().deploy(deployment);

}
}
```

We have created the `WARArchive` and invoked the `staticContent` method. When we open the web browser, we will see the `Hello World` page:

What has happened? The static content method has copied all non-Java files from the `webpage` directory (one file in our example) to the created archive so that they can be seen by `undertow`.

JAXRSArchive

The last type of Swarm archive that we want to look at right now is the `org.wildfly.swarm.JAXRSArchive`. This archive adds the ability to create a default JAX-RS application with the application path set to `"/"`. Till now, we have been doing this manually in all our examples. With the JAX-RS Archive, this class will be added automatically.

XML configuration

Although Java API is convenient, this is not the only option that we have. If you are familiar with the WildFly XML configuration, or if you are migrating your application to Swarm and have a working XML file, you

don't have to translate it to Java API as you can use it directly.

Note

Examples reference: `chapter04/catalog-service-xmlconfig`

Let's return to our database example. You may configure the datasource using XML. In such a case, the XML configuration will look like this:

```
<subsystem xmlns="urn:jboss:domain:datasources:4.0">
    <datasources>
        <drivers>
            <driver name="h2" module="com.h2database.h2">
                <driver-class>org.h2.Driver</driver-class>
                <xa-datasource-class>org.h2.jdbcx.JdbcDataSource</xa-datasource-class>
            </driver>
        </drivers>
        <datasource jndi-name="java:jboss/datasources/CatalogDS" pool-name="CatalogDS" enabled="true" use-java-context="tr
            <connection-url>jdbc:h2:mem:test;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=FALSE</connection-url>
            <driver>h2</driver>
        </datasource>
        <datasource jndi-name="java:jboss/datasources/ExampleDS" pool-name="ExampleDS" enabled="true" use-java-context="tr
            <connection-url>jdbc:h2:mem:test;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=FALSE</connection-url>
            <driver>h2</driver>
            <security>
                <user-name>sa</user-name>
                <password>sa</password>
            </security>
        </datasource>
    </datasources>
</subsystem>
```

We have to add this configuration file to the `resources` directory:

Finally, we also have to tell Swarm to use the configuration file. The following is the modified `Main` class:

```
package org.packt.swarm.petstore.catalog;

import org.jboss.shrinkwrap.api.Archive;
import org.wildfly.swarm.Swarm;
import org.wildfly.swarm.datasources.DatasourcesFraction;
import org.wildfly.swarm.jaxrs.JAXRSArchive;
import org.wildfly.swarm.undertow.UndertowFraction;
import org.wildfly.swarm.undertow.WARArchive;

import java.net.URL;

public class Main {

public static void main(String[] args) throws Exception {

        Swarm swarm = new Swarm();

        //1
        ClassLoader cl = Main.class.getClassLoader();
        URL xmlConfig = cl.getResource("datasources.xml");

        //2
        swarm.withXmlConfig(xmlConfig);
```

```
swarm.start().deploy();

}
```

We have obtained the classloader to be able to locate the configuration file(1). After reading the file, we instructed Swarm to use the configuration from it (2).

However, we have used the whole configuration file—will Swarm use all the subsystems now? The answer is no; only the fractions, whose dependencies have been specified will be added to the container. Swarm, given the XML file, will read only the configuration of those subsystems whose fractions constitute it. You are also able to provide a file with only those subsystems that you want to configure using XML.

YAML configuration

Another way in which you can provide Swarm configuration is YAML data serialization language.

Once more, let's start with the port-change example. We will start again with JAX-RS example and modify it to use the YAML configuration.

First, let's create the HTTP-`port.yml` configuration file inside the `resources` directory:

```
swarm:
  http:
    port: 12345
```

The nested properties are translated to flat properties by Swarm. So, the property specified by the preceding file is translated to `swarm.http.port`, which we know well already.

To use the following configuration, we have to modify our `Main` class:

```
package org.packt.swarm.petstore.catalog;

import org.wildfly.swarm.Swarm;

import java.net.URL;

public class Main {

public static void main(String[] args) throws Exception {

        Swarm swarm = new Swarm();

//1
        ClassLoader cl = Main.class.getClassLoader();
        URL yamlConfig = cl.getResource("http-port.yml");

//2
        swarm.withConfig(yamlConfig);

swarm.start().deploy();
}
}
```

After obtaining the configuration from the `classpath` (1), we informed Swarm to use it using the `withConfig` method. That's it; now, Swarm will use the `12345` port.

Project stages

The strength of the YAML configuration is its ability to provide different groups properties for different project stages. Again, let's take a look at the example first.

The new configuration file looks like this:

```
swarm:
  http:
    port: 8080
---
project:
    stage: test
swarm:
    http:
        port: 12345
---
project:
    stage: QA
swarm:
    http:
        port: 12346
```

The different parts of the file gather the configuration for different project stages. The first group is the default configuration. It is used when no stage name is provided. The other two specify the configurations for test and `QA` stages. However, how do you know the stage in which the application currently runs? You have to provide the `swarm.project.stage` property. So, consider that, for example, we run the preceding example with the following command:

```
mvn wildfly-swarm:run -Dswarm.project.stage=QA
```

Then, we will be able to access our application on the `12346` port.

As you will have noticed in the preceding code, the YAML configuration makes it easy to create the configuration for different environments and choose what group of properties should be used using a simple command-line argument.

YAML database configuration

As an another YAML config example, we are going to show you how to configure the datasources with the YAML configuration file. Let's take a look:

Note

Examples reference:
`chapter 4/catalog-service-database-ymlconfig`

The example is very similar to the XML configuration example. We have to exchange the configuration file for its YAML equivalent:

```
swarm:
  datasources:
    data-sources:
      CatalogDS:
        driver-name: h2
connection-url: jdbc:h2:mem:test;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=FALSE
user-name: sa
password: sa
jdbc-drivers:
          h2:
            driver-class-name: org.h2.Driver
xa-datasource-name: org.h2.jdbcx.JdbcDataSource
driver-module-name: com.h2database.h2
```

And also need to make the `Main` class use it (1):

```
package org.packt.swarm.petstore.catalog;

import org.wildfly.swarm.Swarm;

import java.net.URL;

public class Main {

public static void main(String[] args) throws Exception {
        Swarm swarm = new Swarm();

        //1
        ClassLoader cl = Main.class.getClassLoader();
        URL ymlConfig = cl.getResource("datasources.yml");

        swarm.withConfig(ymlConfig);

swarm.start().deploy();
}
}
```

We are going to use such configurations a lot in the examples throughout the book.