Cheat Sheet: Wildfly Swarm

RED HAT DEVELOPERS

Table of Contents

1. Introduction

Microservices architectures (MSA) are a departure from the traditional application composition as they emphasize the development of smaller individual services centered around a finite purpose instead of single deployable unit (commonly known as a monolith). By breaking down a larger application into compartmental services, each can participate in their own development and deployment lifecycle, allowing for faster turnaround, reduced time to market.

One of the challenges when adopting new architectural approaches is the need to balance and use the latest and greatest technologies, while continuing to leverage well established frameworks and patterns that provide stability and reliability. Traditional applications have made use of application servers, such as Wildfly, as their deployment platform. However, most applications only use a fraction of the available Java EE features.

WildFly Swarm is a framework for building enterprise Java microservices. By deconstructing the WildFly Application Server into finer-grained components, applications can be designed to make use of only the enterprise Java EE API's they need. The application and the supporting components are packaged together into an executable runtime, known as an "uber jar", for development.

2. Three ways to create a Wildfly Swarm application

2.A Developing a Swarm Application

A new or existing Java EE Maven application can be configured as a WildFly Swarm application in three simple steps within the application's pom.xml file:

1. Include the Java EE 7 API (If not already present by way of another dependency. Most existing applications will already reference these APIs.)

```
<dependencies>
...
    <!-- Java EE 7 dependency -->
    <dependency>
        <groupId>javax</groupId>
        <artifactId>javaee-api</artifactId>
        <version>7.0</version>
        <scope>provided</scope>
    </dependency>
...
</dependencies>
```

2. Add Swarm to the DependencyManagement section of your Maven POM

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.wildfly.swarm</groupId>
            <artifactId>bom</artifactId>
            <version>${version.wildfly.swarm}</version>
            <scope>import</scope>
            <type>pom</type>
        </dependency>
    </dependencies>
</dependencyManagement>
```

3. Add the Swarm Maven Plugin

```
<plugins>
...
    <plugin>
        <groupId>org.wildfly.swarm</groupId>
        <artifactId>wildfly-swarm-plugin</artifactId>
        <version>${version.wildfly.swarm}</version>
        <executions>
            <execution>
                <goals>
                    <goal>package</goal>
                </goals>
            </execution>
        </executions>
    </plugin>
...
</plugins>
```

2.B Swarm Generator

A preconfigured maven project can be created using the WildFly Swarm Generator:
http://wildfly-swarm.io/generator/

3 steps to generate a project:

1. Provide a Maven GroupId, ArtifactId and Version (GAV)
2. Select from available dependencies, such as JAX-RS, to include in the application
3. Download the complete, generated application

2.C Swarmtool

Swarmtool is a standalone jar that can be used to convert an existing jar or war into a swarm application without requiring any other tooling. Existing projects can be run as WildFly Swarm applications in three steps:

1. Download Swarmtool

Swarmtool is available for download in Maven Central by matching the WildFly Swarm version that is referenced in your application's pom.xml file:

```
curl -O https://repo1.maven.org/maven2/org/wildfly/swarm/swarmtool/<WILDFLY_SWARM_
VERSION>/swarmtool-<WILDFLY_SWARM_VERSION>-standalone.jar
```

2. Determine options to apply (see table below and apply any option as necessary.)

3. Execute the swarm tool jar via the command line

Since the swarmtool jar is executable, it can be renamed to a value which is easier to execute, such as `swarmtool`. The following command can then be executed:

```
swarmtool <options> <artifact>
```

Alternatively, it can be executed using `java -jar`

```
swarmtool <options> <artifact>
```

Commonly used Swarmtool options:

Name	Description
`-Dkey=value`	Set a property to be used at runtime by the -swarm.jar. Overrides values from --properties-file.
`--executable`	Causes the -swarm.jar to be generated as a executable
`-f,--fractions=x,y,z`	A list of extra fractions to include when auto-detection is used. Each fraction can be of the form group:name:version, name:version, or name.
`--modules=dir1,dir2`	A list of paths to dirs containing any required JBoss Modules
`-n, --name=name`	Name of the resulting jar
`--no-bundle-deps`	Disables bundling dependencies within the artifact. These dependencies will be resolved from a Maven repository at runtime
`--no-fraction-detect`	Disable fraction autodetection
`-o, --output-dir=path`	Directory the resulting jar will be written
`--properties-file`	System properties to apply at runtime
`--repos=url1,url2`	Maven repositories to resolve dependencies in addition to Maven Central and the JBoss Public Repository

3. Customizing the Runtime

There are several approaches for configuring WildFly swarm:

- src/main/resources/project-stages.yml

The recommended approach for customizing the runtime is to define configuration parameters in a file called project-stages.yml which is located by default on the classpath in the src/main/resources folder.

```
logger:
    level: DEBUG
swarm:
  port:
    offset: 10
```

- WildFly standalone.xml file

Custom configuration can also be specified using the WildFly Standalone XML file, a declarative XML model (equivalent to a WildFly Application Server configuration).

```
<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/ExampleDS” pool-name=”ExampleDS”
       enabled=”true” use-java-context=”true”>
           <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>
```

This file is typically stored on the classpath (in the src/main/resources folder) and must be referenced when booting up the container in a main class.

```
...
ClassLoader cl = Main.class.getClassLoader();
URL xmlConfig = cl.getResource(“standalone.xml”);
Swarm swarm = new Swarm(false)
        .withXmlConfig(xmlConfig);
...
```

4. Ways to Run a Swarm Application

WildFly Swarm applications can be compiled and executed either by building an Uber Jar or using the WildFly Swarm Maven Plugin

- Build an Uber Jar

```
$ mvn package
$ java -jar <project>-swarm.jar
```

- Maven Plugin

```
$ mvn wildfly-swarm:run
```

- Integrated Development Environment (IDE)

Execute the main class of the application or the org.wildfly.swarm.Swarm class from an IDE

5. Configuring a Swarm Application

A wide range of configuration options are available to tailor the execution of WildFly Swarm applications.

5.A Java System Properties

These properties are specified when launching the application on the command line:

```
$ java -jar <project>-swarm.jar -Dproperty=value -Dproperty2=value2
```

Name	Description	Default
`swarm.export.deployment`	Deployed artifact to be dumped to disk when swarm starts, for debugging	False
`swarm.port.offset`	Global port offset	0
`swarm.bind.address`	Interface to bind servers	0.0.0.0
`swarm.http.eager`	Eagerly open HTTP listeners	false
`swarm.http.port`	HTTP server port	8080
`swarm.https.port`	HTTPS server port	8443
`swarm.context.path`	Context path for the deployed application	/
`swarm.debug.port`	If provided, the swarm process will pause for debugging on the given port. This option is only available when running an Arquillian test or `mvn wildfly-swarm:run`, not when executing `java -jar`	none

Name	Description	Default
`swarm.debug.bootstrap`	Context path for the deployed application	false
`swarm.project.stage`	Activates a stage in project-stages. yaml	none
`swarm.project.stage.file`	A URL reference to a project state file	none
`swarm.deployment.timeout`	Timeout, in seconds, to wait for a deployment to occur	300
`swarm.ds.username`	Username for database access	none
`swarm.ds.password`	Password for database access	none
`swarm.ds.connection.url`	JDBC Connection URL	none
`swarm.jdbc.driver`	Database Driver name	none

5.B Command Line

When a container main class is not provided, WildFly Swarm supports a number of command line arguments that can be passed to customize the execution

Argument	Description	Default
`-D<name> or -D<name>=value`	Defines a Java system property (See table above.)	none
`-P<file-or-url>`	Location of a java .properties file to use as system properties (See table above.)	none
`-b <addr>`	Bind the public listeners to an address	0.0.0.0 (all)
`-c <file-or-url>`	Specify an XML configuration file (such as standalone.xml or a fragment)	none
`-s <file-or-url>`	Specify a project-stages.yml	project-stages.yml from the application classpath, if provide
`-S <active-stage`	Name of the stage to activate from a project-stages.yml file	none
`--help or -h`	Display the relevant help, including any known project-stages.yml keys	N/A
`--version or -v`	Display the version of the WildFly Swarm being used	N/A

5.C Project Stages

It may be desirable to externalize configuration variables and enabled their usage per runtime environment. This is accomplished by including each environment configuration as a set of combined yml files separated by a --- within the project-stages.yml file. Each named stage must have project: stage: <name> which denotes the name of the stage.

```
logger:
    level: DEBUG
swarm:
  port:
    offset: 10
---
project:
    stage: development
logger:
    level: DEBUG
swarm:
  port:
    offset: 50
---
project:
    stage: production
logger:
    level: INFO
swarm:
  port:
    offset: 100
```

5.D Swarm Maven Plugin

The `swarm-maven-plugin` is responsible for assembling and packaging the uberjar containing the application. Additional properties can be defined within the plugin to drive the execution of the resulting application.

```
<plugin>
    <groupId>org.wildfly.swarm</groupId>
    <artifactId>wildfly-swarm-plugin</artifactId>
    <configuration>
        <properties>
            <swarm.port.offset>1</swarm.port.offset>
        </properties>
    </configuration>
</plugin>
```

6. The Main Class

The main class is the root of every WildFly Swarm application and includes an API for managing the full lifecycle of starting, stopping and its application deployment.

A main method defined within a class can be provided to explicitly define the configuration of the deployment, but is not required.

```
public class MyMain {
    public static void main(String...args) {

        // Instantiate the container
        Swarm swarm = new Swarm();

        // Create one or more deployments
        JAXRSArchive deployment = ShrinkWrap.create(JAXRSArchive.class);
        ...

        // Add resource to deployment
        deployment.addClass(MyResource.class);

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

Once defined, the class must be configured for use via the swarm-maven-plugin:

```
<configuration>
    <mainClass>org.wildfly.swarm.examples.jaxrs.Main</mainClass>
</configuration>
```

If a main class is not provided, one will be constructed automatically based on the dependencies included with the application.

7. Fractions

Fractions (or features) are a set of defined capabilities that can be added to WildFly Swarm application, such as support for RESTful services, or provide integration qualities such as persistence or messaging. They map directly to an existing subsystem within WildFly or add functionality from external resources.

7.A Java EE

The following are a non exhaustive list of fractions which are associated with functionality found in a WildFly subsystem

CDI: Support for CDI (Contexts and Dependency Injection)

Maven Dependencies	`<dependency>`
	`  <groupId>org.wildfly.swarm</groupId>`
	`  <artifactId>cdi</artifactId>`
	`</dependency>`

Description	Define CDI based annotations, such as @Inject within the application

Datasources: Managed connections to relational databases

Maven Dependencies	`<dependency>`
	`  <groupId>org.wildfly.swarm</groupId>`
	`  <artifactId>datasources</artifactId>`
	`</dependency>`

project-stages.yml File	`swarm:`
	`  datasources:`
	`    data-sources:`
	`      ### [datasource]`
	`      MyDS:`
	`        driver-name: myh2`
	`        connection-url: jdbc:h2:mem:test;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=FALSE`
	`        user-name: sa`
	`        password: sa`
	`      ### [datasource]`
	`    jdbc-drivers:`
	`      ### [driver]`

```
myh2:
  driver-class-name: org.h2.Driver
  xa-datasource-name: org.h2.jdbcx.JdbcDataSource
  driver-module-name: com.h2database.h2
```

JAX-RS: Java EE API for web services that can be used as a replacement for servlet based applications. In addition to base JAX-RS functionality, integration is also available for CDI and multipart forms.

Maven Dependencies

```
<dependency>
  <groupId>org.wildfly.swarm</groupId>
  <artifactId>jaxrs</artifactId>
</dependency>

<dependency>
  <groupId>org.wildfly.swarm</groupId>
  <artifactId>jaxrs-multipart</artifactId>
</dependency>
```

JPA: Java EE API to access relational databases, such as H2, MySQL, PostgreSQL

Maven Dependencies

```
<dependency>
  <groupId>org.wildfly.swarm</groupId>
  <artifactId>jpa</artifactId>
</dependency>
```

Notes

There is no need to add any specific JPA API's as they will be added automatically. However, you will still need to add the dependency for the JDBC driver being utilized.

Logging: Additional capabilities to configure logging outside of the default behavior to print to the console

Maven Dependencies

```
<dependency>
  <groupId>org.wildfly.swarm</groupId>
  <artifactId>logging</artifactId>
</dependency>
```

Transactions: JTA and JTS support. Uses the Narayana transaction manager to expose TCP ports 4712 and 4713 (recovery and status).

Maven Dependencies

```
<dependency>
  <groupId>org.wildfly.swarm</groupId>
  <artifactId>transactions</artifactId>
</dependency>
```

project-stages.yml File

```
swarm:
  transactions:
    port: 4712
```

7.B Beyond Java EE

Flyway: Open source database migration tool that strongly simplicity and convention over configuration

Maven Dependencies

```
<dependency>
  <groupId>org.wildfly.swarm</groupId>
  <artifactId>flyway</artifactId>
</dependency>
```

project-stages.yml File

```
swarm:
  flyway:
    jdbc-url: jdbc:oracle:oci8:@
    jdbc-user: admin
    jdbc-password: password
```

All SQL migration should should be placed within the src/main/resources/db/migration folder to be packaged in the uberjar

Logstash: Redirect logs to the Logstash centralized log management server.

Maven Dependencies

```
<dependency>
  <groupId>org.wildfly.swarm</groupId>
  <artifactId>logstash</artifactId>
</dependency>
```

Configuration Properties

```
swarm.logstash.hostname
swarm.logstash.port
```

Netflix OSS: Netflix has created several projects focused on Microservices that can be integrated into WildFly Swarm

Ribbon: Library providing a method for discovering services by name, and allowing clients to invoke those services. One of the primary facilities of Ribbon is client-side load-balancing. With Ribbon, each client discovers possibly many instances of a given service, and uses a strategy (such as round-robin) to balance requests to the service across the many providers.

Hystrix: To support the circuit breaker integration pattern, Hystrix is a library designed to allow microservices to satisfy requests even when remote services are unavailable.

Maven Dependencies

```
<dependency>
  <groupId>org.wildfly.swarm</groupId>
  <artifactId>ribbon</artifactId>
</dependency>
```

Configuration Properties

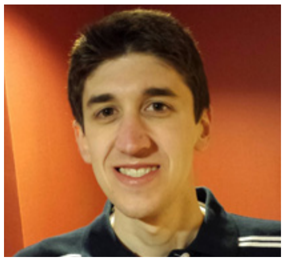
`swarm.ribbon.context.path -` /ribbon by default

Service Advertising using annotations

```
@Advertise(“recommendations”)
public class Recommendations {
…
}
```

Note: The ribbon dependency will transitively provide dependent libraries, such as Hystrix, RxJava and Netty

8. About the Author



Andrew Block is a Principal Consultant with Red Hat Consulting. He specializes on delivering cloud based and integration solutions for customers along with emphasizing the importance of Continuous Integration and Continuous Delivery methodologies. Andrew is a contributor to a variety of open source projects, blogs about his experiences, and speaks at conferences.

Twitter @sabre1041   Linkedin