Deployment options for OSGi applications in the cloud/edge
Deployment options for OSGi applications in the cloud/edge

Speaker

Dirk Fauth
Research Engineer
Eclipse Committer

ETAS GmbH
Borsigstraße 24
70469 Stuttgart

dirk.fauth@etas.com
www.etas.com

blog.vogella.com/author/fipro/
Twitter: fipro78
Deployment options for OSGi applications in the cloud/edge

Overview

1. Deployment Variants
2. Container
3. Benchmark
4. Conclusion
Deployment Variants
Deployment Variants

**General**

- **Multiple JARs in a folder**
  - Multiple JARs
  - JRE

- **Executable JAR**
  - Single JAR
  - JRE

- **Custom JRE (jlink)**
  - Custom JRE
  - Java

- **Native Executable**
  - Native executable
  - GraalVM
Deployment Variants

Multiple JARs in a folder

- Multiple JAR files (OSGi bundles) inside a folder
- Additional configuration file
- Launcher

```java
java -jar org.eclipse.osgi-3.17.200.jar
```

- Build
  - maven-dependency-plugin
  - maven-resources-plugin

Deployment Variants

Executable JAR

- Executable JAR that includes each required bundle as embedded JAR file
- Configuration also included in the executable JAR
- Launcher
  aQute.launcher.pre.EmbeddedLauncher

```
java -jar equinox-app.jar
```

- Build
  - bnd-maven-plugin
  - bnd-export-maven-plugin

https://bnd.bndtools.org/
https://bndtools.org/
https://github.com/bndtools/bnd/tree/master/maven
Deployment Variants

Custom JRE via jlink

- Create a custom JRE with `jlink` command of the JDK
  - *assemble and optimize a set of modules and their dependencies into a custom runtime image*

  https://docs.oracle.com/en/java/javase/17/docs/specs/man/jlink.html

- Folder layout like JRE
- Launcher: `java` command

```java
java [options] -m <module>[/<mainclass>]
```

- Issue with OSGi and jlink
  
  Most available OSGi bundles do not contain a `module-info.class` → automatic module cannot be used with jlink

JPMS
Deployment Variants
Native Executable with GraalVM

- **Native Image** is a technology to compile Java code ahead-of-time to a binary – a native executable. A native executable includes only the code required at run time, that is the application classes, standard-library classes, the language runtime, and statically-linked native code from the JDK.

- Can be created using the GraalVM native-image tool
  - From a **Class**, a **JAR (classpath)** or a **Module (modulepath)**

- “Closed world assumption”
  - all the bytecode in your application that can be called at run time must be known at build time

- Issue with OSGi and native-image
  Dynamic classloading per bundle managed by OSGi Framework (Module Layer)
  ```java
  java.lang.NullPointerException: A null service reference is not allowed.
  ```

Deployment Variants

OSGi

- Multiple JARs in a folder
- Executable JAR
- Custom JRE (jlink)
- Native Executable
Deployment Variants

Custom JRE via jlink - OSGi

– Add `module-info.class`

– ModiTect
  https://github.com/moditect/moditect
  ➔ Intrusive change that adds an artifact to an existing published JAR
  OSS license compatibility?
  Checksum?
  ➔ Requires knowledge on internals for generation
  Maintenance?

– Bndtools JPMS Support
  https://bnd.bndtools.org/chapters/330-jpms.html
Enable creation of `module-info.class` for each bundle, e.g. via `bnd-maven-plugin`

```xml
<plugin>
  <groupId>biz.aQute.bnd</groupId>
  <artifactId>bnd-maven-plugin</artifactId>
  <configuration>
    <bnd>
      <![CDATA[
        Bundle-SymbolicName: ${project.groupId}.${project.artifactId}
        -sources: true
        -contract: *
        -jpms-module-info:org.fipro.service.command;modules='org.apache.felix.configadmin'
        -jpms-module-info-options: org.osgi.service.cm;ignore="true"
      ]]>)
    </bnd>
    </configuration>
  </plugin>
```
Enable creation of `module-info.class` for **executable jar** via `.bndrun` file

```
-jpms-module-info: \
 ${project.groupId}.equinox.${project.artifactId};\n    version=${project.version};\n    ee=JavaSE-${java.specification.version}
-jpms-module-info-options: jdk.unsupported;static=false
```

This makes the executable jar itself a module!
Deployment Variants

Custom JRE via jlink with Bndtools JPMS support

Build

```bash
$JAVA_HOME/bin/jlink \  
  --add-modules org.fipro.service.equinox.app \  
  --module-path equinox-app.jar \  
  --no-header-files \  
  --no-man-pages \  
  --output /app/jre
```

Launch

```bash
/app/jre/bin/java \
  -m org.fipro.service.equinox.app/aQute.launcher.pre.EmbeddedLauncher
```
Deployment Variants

OSGi

- Multiple JARs in a folder
- Executable JAR
- Custom JRE (jlink)
- Native Executable
Deployment Variants

OSGi Connect

– *OSGi Connect allows for bundles to exist and be installed into the OSGi Framework from the flat class path, the module path (Java Platform Module System), a jlink image, or a native image.*

→ Allows to start an OSGi application without the full OSGi Module Layer

OSGi Core R8 – Connect Specification
https://docs.osgi.org/specification/osgi.core/8.0.0/framework.connect.html

Apache Felix Atomos
https://github.com/apache/felix-atomos

Ubiquitous OSGi - Android, Graal Substrate, Java Modules, Flat Class Path
https://www.youtube.com/watch?v=KxmtzjHBumU

OSGi R8, Felix 7, Atomos and the future of OSGi@Eclipse
https://www.youtube.com/watch?v=oitFMBztf5s
Deployment Variants

GraalVM Native Image with OSGi Connect

– Preparation
  1. Add/use Atomos to be able to start the OSGi application from the flat classpath
  2. Generate reachability metadata via tracing agent (reflection, resources, …)
  3. Update generated metadata

– Build
  – Via GraalVM build plugins (Maven/Gradle)
  – Docker multi-stage build using GraalVM container images

– Notes/Remarks
  – *native-image* build only worked with flat classpath and listing all jars explicitly
  – Build result is platform-dependent
  – atomos_lib folder or index file needed for Atomos to discover bundles and load bundle entries
  – Still not everything is working as expected (e.g. *scr:list* produces an empty output)
Deployment Variants

OSGi Connect / Apache Felix Atomos

Multiple JARs in a folder + Custom JRE (jlink)

Executable JAR + Native Executable
## Deployment Variants

### Deployment (plain OSGi)
- Multiple JARs in folder
- Executable JAR
- Custom JRE (`jlink`)
- GraalVM Native Image

### Deployment (OSGi Connect)
- Multiple JARs in folder
- Executable JAR
- Custom JRE (`jlink`)
- GraalVM Native Image
Container
“Size matters” – Find the right base image

### Alpine vs. Debian vs. Ubuntu

<table>
<thead>
<tr>
<th>Image</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpine:3</td>
<td>5.54 MB</td>
</tr>
<tr>
<td>debian:bullseye-slim</td>
<td>80.50 MB</td>
</tr>
<tr>
<td>ubuntu:jammy</td>
<td>77.84 MB</td>
</tr>
</tbody>
</table>

### Eclipse Temurin vs. IBM Semeru

<table>
<thead>
<tr>
<th>Image</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>eclipse-temurin:17-jdk-jammy</td>
<td>~ 455 MB</td>
</tr>
<tr>
<td>eclipse-temurin:17-jdk-alpine</td>
<td>~ 356 MB</td>
</tr>
<tr>
<td>eclipse-temurin:17-jre-jammy</td>
<td>~ 266 MB</td>
</tr>
<tr>
<td><strong>eclipse-temurin:17-jre-alpine</strong></td>
<td>~ 168 MB</td>
</tr>
<tr>
<td>ibm-semeru-runtimes:open-17-jdk-jammy</td>
<td>~ 477 MB</td>
</tr>
<tr>
<td>ibm-semeru-runtimes:open-17-jre-jammy</td>
<td>~ 272 MB</td>
</tr>
</tbody>
</table>
Interlude: Distroless

"Distroless" images contain only your application and its runtime dependencies. They do not contain package managers, shells or any other programs you would expect to find in a standard Linux distribution.

<table>
<thead>
<tr>
<th>Image</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcr.io/distroless/static-debian11</td>
<td>2.36 MB</td>
</tr>
<tr>
<td>gcr.io/distroless/base-debian11</td>
<td>20.32 MB</td>
</tr>
<tr>
<td>gcr.io/distroless/java17-debian11</td>
<td>230.88 MB</td>
</tr>
</tbody>
</table>

Distroless Java image is based on Debian and glibc, therefore bigger than an Alpine Temurin image.

Can be interesting in production for security reasons, but not for size.

[https://github.com/GoogleContainerTools/distroless](https://github.com/GoogleContainerTools/distroless)
Container
Java Best Practices

– Install only what you need
  – Use JRE instead of JDK
  – Use multi-stage builds (e.g. to create JRE or Native Image)

– Don’t run Java apps as root

– Properly shutdown and handle events to terminate a Java application

– Take care of “container-awareness”

https://snyk.io/blog/best-practices-to-build-java-containers-with-docker/
https://developers.redhat.com/articles/2022/04/19/java-17-whats-new-openjddks-container-awareness#
https://blog.openj9.org/2021/06/15/innovations-for-java-running-in-containers/
Container

Building Docker Images

– Use dedicated Docker files instead of generation tools

– Integrate image creation as part of the build via fabric8io/docker-maven-plugin

Maven/Gradle first

https://github.com/fabric8io/docker-maven-plugin

– Use multi-stage build to checkout sources and build in one container, then create new production container with build result only

Docker first

```xml
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>docker-maven-plugin</artifactId>
  <extensions>true</extensions>
  <configuration>
    <images>  ... </images>
  </configuration>
  <executions>  ... </executions>
</plugin>
```
## Container

### Deployment Variant – Base Image – Image Size

<table>
<thead>
<tr>
<th>Deployment (plain OSGi)</th>
<th>Base Image</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple JARs in folder</td>
<td>eclipse-temurin:17-jre-alpine</td>
<td>~ 171 MB</td>
</tr>
<tr>
<td>Executable JAR</td>
<td>eclipse-temurin:17-jre-alpine</td>
<td>~ 174 MB</td>
</tr>
<tr>
<td>Custom JRE (jlink)</td>
<td>alpine:3</td>
<td>~ 75 MB</td>
</tr>
<tr>
<td>Custom JRE (jlink/compressed)</td>
<td>alpine:3</td>
<td>~ 53 MB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Deployment (OSGi Connect)</th>
<th>Base Image</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple JARs in folder</td>
<td>eclipse-temurin:17-jre-alpine</td>
<td>~ 171 MB</td>
</tr>
<tr>
<td>Custom JRE (jlink)</td>
<td>alpine:3</td>
<td>~ 75 MB</td>
</tr>
<tr>
<td>Custom JRE (jlink/compressed)</td>
<td>alpine:3</td>
<td>~ 53 MB</td>
</tr>
<tr>
<td>GraalVM Native Image</td>
<td>scratch</td>
<td>~ 38 MB</td>
</tr>
<tr>
<td></td>
<td>alpine:3</td>
<td>~ 43 MB</td>
</tr>
</tbody>
</table>
Benchmark
### Benchmark Bundle / Immediate Component

- Get start timestamp from system property
- Get current timestamp
- Send POST request via `java.net.http.HttpClient`
- Shutdown

### Shell Script

- Execute application multiple times in for-loop (clean/cache)
- Pass start timestamp as system property
## Benchmark Images

### Deployment Variant – Base Image – Image Size – Benchmark Image Size

<table>
<thead>
<tr>
<th>Deployment (plain OSGi)</th>
<th>Base Image</th>
<th>Size</th>
<th>Size Benchmark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple JARs in folder</td>
<td>eclipse-temurin:17-jre-alpine</td>
<td>~ 171 MB</td>
<td>~ 173 MB</td>
</tr>
<tr>
<td>Executable JAR</td>
<td>eclipse-temurin:17-jre-alpine</td>
<td>~ 174 MB</td>
<td>~ 176 MB</td>
</tr>
<tr>
<td>Custom JRE (jlink)</td>
<td>alpine:3</td>
<td>~ 75 MB</td>
<td>~ 78 MB</td>
</tr>
<tr>
<td>Custom JRE (jlink/compressed)</td>
<td>alpine:3</td>
<td>~ 53 MB</td>
<td>~ 55 MB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Deployment (OSGi Connect)</th>
<th>Base Image</th>
<th>Size</th>
<th>Size Benchmark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple JARs in folder</td>
<td>eclipse-temurin:17-jre-alpine</td>
<td>~ 171 MB</td>
<td>~ 173 MB</td>
</tr>
<tr>
<td>Custom JRE (jlink)</td>
<td>alpine:3</td>
<td>~ 75 MB</td>
<td>~ 78 MB</td>
</tr>
<tr>
<td>Custom JRE (jlink/compressed)</td>
<td>alpine:3</td>
<td>~ 53 MB</td>
<td>~ 55 MB</td>
</tr>
</tbody>
</table>

GraalVM Native Image
- scratch
- alpine:3
  - ~ 38 MB
  - (~ 43 MB)
  - (~ 46 MB)
  - ~ 53 MB

+ coreutils
+ nanosecond support
+ benchmark bundle
+ java.net.http module
+ shell script support
# Benchmark Results

## Deployment (plain OSGi)

<table>
<thead>
<tr>
<th>Base Image</th>
<th>Size</th>
<th>Size Benchmark</th>
<th>Startup Clean</th>
<th>Startup Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple JARs in folder</td>
<td>~ 171 MB</td>
<td>~ 173 MB</td>
<td>~ 982 ms</td>
<td>~ 901 ms</td>
</tr>
<tr>
<td>Executable JAR</td>
<td>~ 174 MB</td>
<td>~ 176 MB</td>
<td>~ 1087 ms</td>
<td>~ 1099 ms</td>
</tr>
<tr>
<td>Custom JRE (jlink)</td>
<td>~ 75 MB</td>
<td>~ 78 MB</td>
<td>~ 1336 ms</td>
<td>~ 1345 ms</td>
</tr>
<tr>
<td>Custom JRE (jlink/compressed)</td>
<td>~ 53 MB</td>
<td>~ 55 MB</td>
<td>~ 1497 ms</td>
<td>~ 1505 ms</td>
</tr>
</tbody>
</table>

## Deployment (OSGi Connect)

<table>
<thead>
<tr>
<th>Base Image</th>
<th>Size</th>
<th>Size Benchmark</th>
<th>Startup Clean</th>
<th>Startup Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple JARs in folder</td>
<td>~ 171 MB</td>
<td>~ 173 MB</td>
<td>~ 1122 ms</td>
<td>~ 973 ms</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>~ 1194 ms</td>
<td>~ 1052 ms</td>
</tr>
<tr>
<td>Custom JRE (jlink)</td>
<td>~ 75 MB</td>
<td>~ 78 MB</td>
<td>~ 1439 ms</td>
<td>~ 1326 ms</td>
</tr>
<tr>
<td>Custom JRE (jlink/compressed)</td>
<td>~ 53 MB</td>
<td>~ 55 MB</td>
<td>~ 1593 ms</td>
<td>~ 1445 ms</td>
</tr>
<tr>
<td>GraalVM Native Image</td>
<td>~ 38 MB</td>
<td>(~ 46 MB)</td>
<td>(~ 43 MB)</td>
<td>(~ 34 MB)</td>
</tr>
</tbody>
</table>
Conclusion
Conclusion

– All Java deployment variants possible for OSGi applications via
  – Bndtools JPMS support
  – OSGi Connect (Felix Atomos)

– Different deployment variants have different startup & runtime behaviors

– Make decision about variant dependent on the use case,
  e.g. short running executables in container vs. long running application servers

– Further optimizations possible by configuring the Java runtime,
  e.g. Container-awareness, Garbage Collection, Checkpoint & Restore, etc.
Benchmark Sources

https://github.com/fipro78/osgi_deployment_options
Thank you

Dirk Fauth
ETAS/ENA
dirk.fauth@etas.com