aicas GmbH

JamaicaAMS User Documentation

Version 1.0.0
17 November 2022

© 2018–2022 aicas GmbH, Karlsruhe. All rights reserved.

Every effort has been made to ensure that all statements and information contained in this document are accurate. However, aicas GmbH accepts no liability for any error or omission therein.

This product includes software developed by IAIK of Graz University of Technology. This software is based in part on the work of the Independent JPEG Group.

Portions of this software are copyright © 2020 The FreeType Project (www.freetype.org). All rights reserved.

Java and all Java-based trademarks are registered trademarks of Oracle America, Inc. All other brands or product names are trademarks or registered trademarks of their respective holders.

The software included in this product contains copyrighted software that is licensed under the GNU General Public License (GPL) or GNU Lesser General Public License (LGPL). You may obtain the complete corresponding source code from us for a period of three years after our last shipment of this product. We will charge 30 EUR for the creation and shipment of a physical machine-readable copy of the source code.

Please contact us at the following address for payment instructions:

aicas GmbH
Emmy-Noether-Straße 9
76131 Karlsruhe
Germany

Email: `support@aicas.com`

This offer is valid to anyone in receipt of this information.

Contents

1 JamaicaAMS Framework **5**

2 Getting Started **9**
2.1 System Requirements . . . 10
2.1.1 Host System . . . 10
2.1.2 Target System . . . 11
2.2 Installation . . . 11
2.3 Execution . . . 12
2.3.1 Launching . . . 12
2.3.2 Environment Variables . . . 13
2.3.3 Interaction . . . 14
2.3.4 Exit Codes . . . 15
2.3.5 Example Bundles . . . 16
2.3.5.1 How to install the Examples . . . 16
2.3.5.2 Importing the Examples into Eclipse . . . 16
2.3.5.3 How to build the Examples . . . 17

3 Tools and Components **19**
3.1 Development . . . 19
3.1.1 Emulator . . . 20
3.1.2 PMD and SpotBugs . . . 20
3.2 Deployment . . . 23
3.2.1 jarsigner . . . 23
3.3 Configuration . . . 24
3.3.1 Profiling and JamaicaAMS . . . 24
3.3.2 Bundle Configuration . . . 25
3.3.2.1 Configurator . . . 25
3.3.2.2 Configuration Admin Service . . . 26

4 Security **29**
4.1 Terms and Definitions 29
4.1.1 Public/Private Key Pair 29
4.1.2 Certificates and Chains 30
4.2 Signed JAR File 31
4.3 Authentication and Permissions 33
4.4 Activating Security in JamaicaAMS 33
4.5 An Example of Security Configuration 34
4.5.1 Sign the Example Bundle 35
4.5.2 Configure the System Properties 35
4.5.3 Configure the Policy 36
4.5.4 Set-Up and Operation 36

5 OSGi Framework and Bundles **37**
5.1 Framework Layers 37
5.2 Bundle Lifecycle 38
5.3 Service Orientation 38
5.4 Controlling the Bundles 39

6 How to write a Bundle with Eclipse **41**
6.1 Prerequisites 41
6.2 Bundle Tutorial 41
6.2.1 Create a new Plug-In Project 41
6.2.2 Make Yourself Familiar with the UI 42
6.2.3 Implement the Functionality 42
6.2.4 Run the Bundle on the Integrated Framework 42
6.2.5 Deployment 43

7 Debugging Bundles with Eclipse **45**
7.1 Prerequisites 45
7.2 Background 45
7.3 Setup of the Debugging Environment 46
7.3.1 Run the Debug Server 47
7.3.2 Run the Debug Client 47

8 JamaicaAMS Runtime Reference **49**
8.1 JamaicaAMS Properties 49
8.1.1 Config Properties 49
8.1.2 System Properties 51
8.1.3 Logger Properties 52
8.2 Budgets 52
8.3 Thread Count 54
8.3.1 Bundle Threads 55
8.4 Usage of the Java Native Interface (JNI) 57

9 Information for Specific Targets **59**
9.1 Linux 59
9.1.1 Shared Libraries 59
9.1.2 Random Number Generator 60

Chapter 1

JamaicaAMS Framework

JamaicaAMS is a modular and extensible application framework, especially designed and tailored for Industrial IoT use cases. It provides a powerful runtime environment for Java-based applications and components, thereby supporting not only static but also highly dynamic and distributed application scenarios.

JamaicaAMS targets in particular at heterogeneous embedded and mobile devices with sparse resources, providing performance guarantees for their applications during runtime. However, high-end servers and computational ecosystems are feasible target platforms as well, which also can take advantage of the integrated framework features and technology in a larger scaled fashion.

The strength of JamaicaAMS results from the combination of two solid open standards: OSGi™[1], that specifies a software architecture to create modular applications and services, called bundles, and the Real-Time Specification for Java (RTSJ).

The JamaicaAMS Application Management System (AMS) extends Apache Felix (currently version 7.0.3), which is an open source implementation of the OSGi specification, in its Core Release 8. The Apache framework implements standardized mechanisms for lifecycle and dependency management of components, service abstractions as well as standardized security and isolation mechanisms.

By implementing the OSGi open technology, JamaicaAMS allows for a seamless and automated integration, configuration and update of application components on a big amount of remote devices, at once and during runtime. By eliminating system downtime for maintenance and updates, it vastly improves the availability of services and avoids disruption, for instance in production, automation and consumer processes.

Moreover, JamaicaAMS relies on the robustness of JamaicaVM, aicas' hard realtime Java-based Virtual Machine. Based on the RTSJ specification, JamaicaVM implements mechanisms for realtime programming and execution. Therefore JamaicaAMS permits to explicitly configure and enforce resource budgets for applications and their components during runtime. Thus, even if the system might be overloaded and misbehaving, the framework and the resource-isolated applications are ensured to operate in a reliable fashion. This is especially useful for mixed critical- and control automation systems where, for example, the sampling of sensors and the control of actuators must be guaranteed in either case.

[1] OSGi™ is a trademark, registered trademark or service mark of the OSGi Alliance.

JamaicaAMS can be complemented, when used together with a set of specialized tools and features (see Figure 1.1) which support development, analysis, build, deployment, configuration and maintenance of Java-based applications and components. Integration with well established development environments, like the Eclipse IDE, speeds up the implementation process, shortening the time-to-market of future IoT applications.

Cloud connectivity features can easily be integrated into JamaicaAMS to allow for a central platform and device management by remote operators.

Industrial IoT use cases are especially supported by automation bus interfaces, like Modbus. Finally, a custom component repository can be easily deployed and integrated, providing a common code-base for a multitude of devices in an app-store fashion. Comprising those features and functionalities based on open standards, JamaicaAMS provides a powerful and portable platform for a variety of innovative, performance critical, highly distributed and dynamic industrial and IoT scenarios.

This document provides information about particular technologies and components included in the JamaicaAMS. It assumes from its readers a certain knowledge about the OSGi standard. However, for those who are not familiar with the specification, Section 5 offers a brief introduction to the OSGi layers and the core concept of bundle lifecycle, which are central to the development and management of modular applications.

For additional information or conceptual clarification, please see the OSGi Core Release 8 Specification [5].

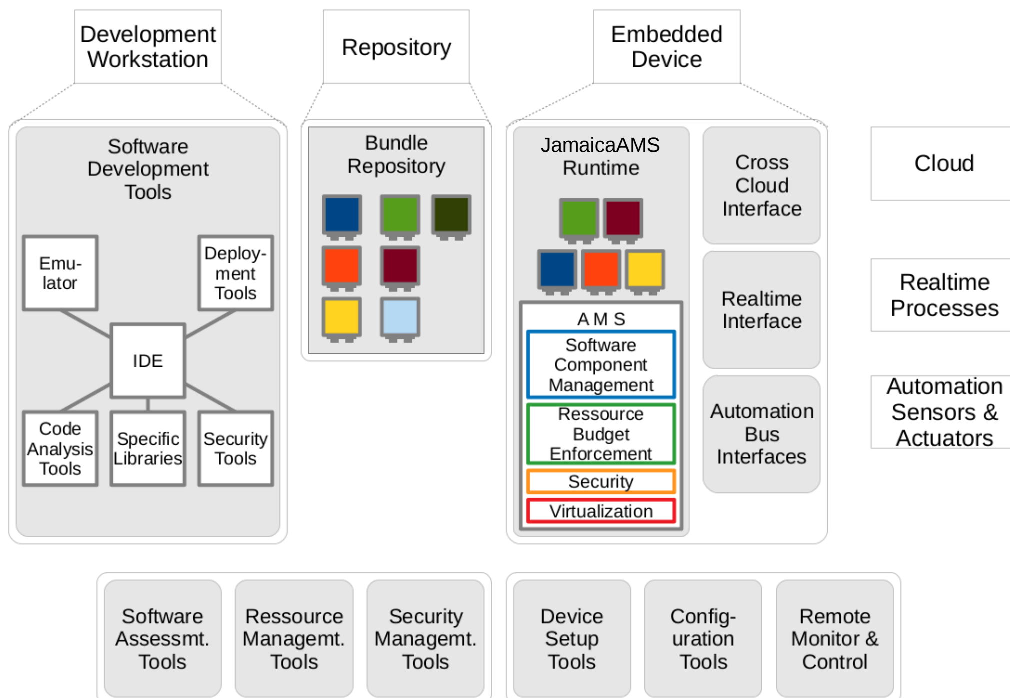

Figure 1.1: JamaicaAMS Framework and its components

Typographic Conventions

Throughout this manual the following typographic conventions were applied.

For items related to the file system, like paths and files:

path

filename

For references in code, like classes and methods:

class

method

Commands are given like:

“command”

Names of components, modules or bundles appear like:

“component”

“module”

“bundle”

Properties, Parameters, Environment Variables, and other items that require definition are written like:

property

VARIABLE

Output in terminal sessions is reproduced in monospaced. User inputs are designated by a prompt:

```
> input
```

Please note that the placeholder “< >” is to be replaced by information pertinent to the particular deployment in question, e.g., <path to JamaicaAMS> should be replaced with the current path to the JamaicaAMS directory in the user’s specific filesystem.

Chapter 2

Getting Started

JamaicaAMS comprises the interaction of three parts: a target runtime, that runs on the edge devices; the development components, which include, apart from the host runtime and libraries, the IDE of choice and tools; and the components that take care of connecting with cloud services.

This section offers information for a straight start, on how to install and execute JamaicaAMS on a target platform.

The current file structure of the JamaicaAMS distribution is depicted below. Please note that `<target>` and `<host>`, under `<setup>`, designate the folders containing binaries for (target) devices running JamaicaAMS and for a (host) platform to be used to develop bundles.

```
jamaica-ams
|-- ApacheLicense2.txt
|-- doc
|   |-- build.info
|   |-- documentation.pdf
|   |-- KNOWN_ISSUES
|   `-- Release_Notes
|-- example
|   |-- primes-<version>.jar
|   |-- primes-<version>-sources.zip
|   |-- primes-with-budget-<version>.jar
|   `-- primes-with-budget-<version>-sources.zip
|-- NOTICE.txt
`-- setup
    |-- bundle.1
    |   |-- configuration-admin-<version>.jar
    |   |-- org.apache.felix.cm.json-<version>.jar
    |   |-- org.apache.felix.configurator-<version>.jar
    |   |-- org.apache.felix.converter-<version>.jar
    |   |-- org.apache.felix.log-<version>.jar
    |   |-- org.apache.sling.commons.johnzon-<version>.jar
    |   |-- org.osgi.util.function-<version>.jar
```

```
|   |-- osgi-log-writer-<version>.jar
|   `-- security-<version>.jar
|-- bundle.2
|   `-- policy-file-reader-<version>.jar
|-- bundle.3
|   |-- org.apache.felix.gogo.command-<version>.jar
|   |-- org.apache.felix.gogo.runtime-<version>.jar
|   `-- org.apache.felix.gogo.shell-<version>.jar
|-- conf
|   |-- all.policy
|   |-- config.properties
|   |-- logging.properties
|   |-- osgi.all.policy
|   `-- system.properties
|-- <target>
|   `-- bin
|       |-- jams
|       |-- jamsdi
|       |-- jamsi
|       `-- jamsp
`-- <host>
    `-- bin
        |-- emulator
        `-- emulatordi

9 directories, 34 files
```

2.1 System Requirements

The development and execution environment of JamaicaAMS described in this chapter can be divided into two parts, namely the host system and the target system. The following are the necessary system requirements in order to practice the instructions which will be given in the rest of this document.

2.1.1 Host System

To develop applications with the JamaicaAMS framework with the tools provided, we recommend the following configuration:

- **Hardware:** x86-64 processor, with at least 2 GHz CPU clock and 4GB RAM
- **Operating System:** CentOS 7 (64-bit)

2.1.2 Target System

JamaicaAMS is capable of running on various combinations of hardware architectures and operating systems. The following example describes one of the typical configurations for running JamaicaAMS on an embedded target device.

- **Hardware:** ARM Cortex A53, 1.2 GHz CPU clock, 1GB SDRAM
- **Operating System:** Linux Kernel 4.14

> Note that...
>
> Running on a target with much less computation power is possible or even more common in a typical JamaicaAMS use case. For more detailed information about the supported targets, please contact `support@aicas.com`.

2.2 Installation

JamaicaAMS is distributed in the form of a ZIP file. In order to install JamaicaAMS simply unzip the file and copy the created directory structure to the target platform. For example on a Linux target:

```
> unzip JamaicaAMS-<version>-linux-armv7-le-raspberrypi-full.zip
```

The JamaicaAMS directory structure is as depicted below. Please note that the placeholder between "< >" needs to be replaced by information pertinent to the particular deployment in question, e.g., in Figure 2.1, the current path to the JamaicaAMS directory.

Directory	Contents
`<path to JamaicaAMS>/doc`	documentation
`<path to JamaicaAMS>/example`	examples
`<path to JamaicaAMS>/setup/bundle.`i [1]	auto-deploy OSGi bundles
`<path to JamaicaAMS>/setup/conf`	configuration files
`<path to JamaicaAMS>/setup/<host>/bin`	executable binaries
`<path to JamaicaAMS>/setup/<target>/bin`	executable binaries

Figure 2.1: JamaicaAMS directory structure.

[1] Here, the OSGi bundles from `bundle.`i are assigned start level i.

2.3 Execution

The JamaicaAMS distribution contains four executable binaries in the `bin` directory: `jamsi` (interpreted), `jams` (default), `jamsdi` (debug interface) and `jamsp` (profiling).

The `jamsi` executable should be used when debugging problems or issues, because it provides more detailed output in error cases; the `jams` executable contains ahead-of-time compiled classes which may lead to better performance. The `jamsdi` executable provides remote debugging functionality and the `jamsp` executable should be used to collect information on the amount of runtime spent for the execution of individual methods.

2.3.1 Launching

The binaries can be launched as follows, where `jams` is used for concreteness and `jamsi` can be launched the same way:

```
> cd <path-to-root-directory-of-JamaicaAMS>/setup

  ./<target>/bin/jams [-c <config-properties-file>] \
                      [-s <system-properties-file>] \
                      [-b <bundle-cache-dir>]
```

Here, the parameters have the following meaning:

-c <config-properties-file>

points to a framework configuration file. This should typically be set to `<path to JamaicaAMS>/conf/config.properties`. That file contains a description of the most important framework configuration properties and can be adapted if needed. If not given, it defaults to `./conf/config.properties`.

-s <system-properties-file>

points to a system configuration file. This makes it possible to set system properties when running JamaicaAMS. The JamaicaAMS distribution contains a sample system configuration file that shows how to enable security. If not given, it defaults to `./conf/system.properties`.

-b <bundle-cache-dir>

specifies the cache directory for the OSGi bundles. If not given, it defaults to `./jamaica-ams-cache`.

Apart from those, the following parameters are also available:

-version prints the version of JamaicaAMS and exits.

-help prints usage information of JamaicaAMS and exits.

2.3.2 Environment Variables

The following environment variables are recognized by JamaicaAMS:

- **JAMAICA_AMS_HEAP_SIZE** specifies the Java heap size to be used by JamaicaAMS. The Java heap is allocated at startup and stays at the fixed size specified by this environment variable. Default: 64M
- **JAMAICA_AMS_JAVA_STACK_SIZE** specifies the Java stack size to be used by Java threads. Values that are too small can cause a Java `StackOverflowError` to be thrown. Default: 16K
- **JAMAICA_AMS_NATIVE_STACK_SIZE** specifies the C stack size to be used by Java threads. Values that are too small can cause a Java `StackOverflowError` to be thrown or the system to crash. Default: OS-dependent (128K on Linux, 64K on QNX, 80K on Windows)
- **JAMAICA_AMS_NUM_THREADS** specifies the number of Java threads that JamaicaAMS should create on startup. No further Java threads will be created once this number of active threads has been reached. Default: 80

  Please note that JamaicaAMS inherits from JamaicaVM the limitation of 511 as maximum number of Java threads.

2.3.3 Interaction

To interact with JamaicaAMS, Apache Felix Gogo shell is supplied as auto-deploy bundle. After launching JamaicaAMS, type "`help`" into the shell to see the list of the available commands and "`help <command-name>`", to get specific additional information. Figure 2.2 shows a list of the commands.

felix:bundlelevel	gogo:cat
felix:cd	gogo:each
felix:frameworklevel	gogo:echo
felix:headers	gogo:format
felix:help	gogo:getopt
felix:inspect	gogo:gosh
felix:install	gogo:grep
felix:lb	gogo:history
felix:log	gogo:not
felix:ls	gogo:set
felix:refresh	gogo:sh
felix:resolve	gogo:source
felix:start	gogo:tac
felix:stop	gogo:telnetd
felix:uninstall	gogo:type
felix:update	gogo:until
felix:which	

Figure 2.2: Apache Felix Gogo shell commands

For further details, please refer to the Apache Felix Gogo Shell documentation [1].

2.3.4 Exit Codes

Figure 2.3 shows the exit codes that are currently defined and used by JamaicaAMS, followed by a complete list of exit codes which are specific to the Jamaica Virtual Machine.

JamaicaAMS Exit Codes	
0	Normal termination
2	Error while parsing arguments
3	Framework implementation was not found
4	Error while initializing the framework
5	Evaluation timeout has expired
JamaicaVM Standard Exit Codes	
0	Normal termination
1	Exception or error in Java program
2..63	Application specific exit code from `System.exit()`
JamaicaVM Error Codes	
64	JamaicaVM failure
65	VM not initialized
66	Insufficient memory
67	Stack overflow
68	Initialization error
69	Setup failure
70	Clean-up failure
71	Invalid command line arguments
72	No main class
73	`Exec()` failure
74	Lock memory failed
JamaicaVM Internal Errors	
100	Serious error: `HALT` called
101	Internal error
102	Internal test error
103	Function or feature not implemented
104	Exit by signal
105	Unreachable code executed
130	POSIX signal SigInt
143	POSIX signal SigTerm
255	Unexpected termination

Figure 2.3: Summary of the exit codes.

2.3.5 Example Bundles

The JamaicaAMS distribution contains two examples bundles that can be used immediately: “primes”, “primes-with-budget”. The first example, “primes”, will simply calculate prime numbers and print a report in the console each time 1000 prime numbers are calculated, showing how long the calculation took (see Figure 2.4).

```
g! start 12
Computed 1000 primes in 626ms!
Computed 1000 primes in 903ms!
Computed 1000 primes in 1194ms!
Computed 1000 primes in 1508ms!
Computed 1000 primes in 1796ms!
Computed 1000 primes in 2121ms!
Computed 1000 primes in 2423ms!
Computed 1000 primes in 2725ms!
Computed 1000 primes in 3044ms!
stop 12
Stopping the worker thread
g!
```

Figure 2.4: Example of a bundle that calculates prime numbers

The bundle “primes-with-budget” does the exact same calculation, but with a CPU Budget set to 5%. For further information about Budgets in JamaicaAMS, please refer to Section 8.2.

2.3.5.1 How to install the Examples

In order to use the bundles provided as examples, those have to be installed using the GoGo Shell “`install`” command.

```
> install <path to JAR>
```

After they are successfully installed, they can be run with the GoGo Shell “`start #`” command, being “#” a placeholder for the Bundle ID. E.g.:

```
> start 12
```

2.3.5.2 Importing the Examples into Eclipse

The example bundles contained in the JamaicaAMS distribution are Maven Projects. To import these correctly into Eclipse, the following steps have to be taken:

- Extract the example bundle source (zip file)

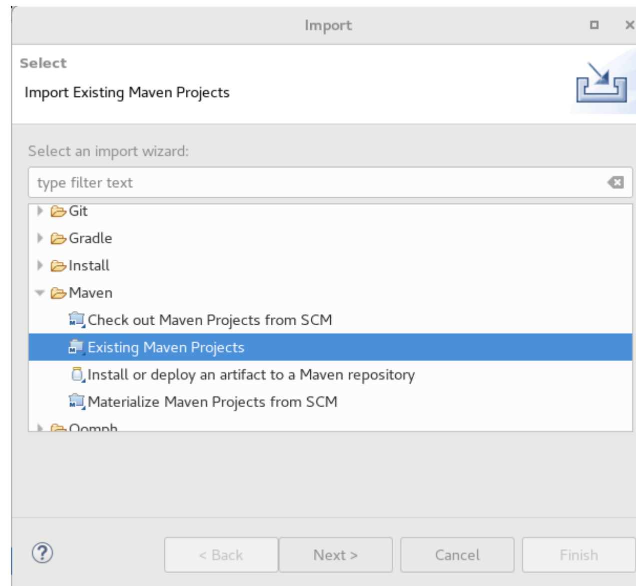

Figure 2.5: Importing the examples in Eclipse

- In Eclipse, proceed with **Select** → **File** → **Import** → **Maven** → **Existing Maven Projects** (see Figure 2.5).
- In the following Window, select **Browse** and navigate to the extracted source folder, clicking the [**OK**] button. Please note that the `pom.xml` will be automatically selected.
- Click the [**Finish**] button and the example bundle will be imported as a Maven Project into Eclipse

After the successful import, the project can be modified.

2.3.5.3 How to build the Examples

The sources for the examples are included in JamaicaAMS. To be able to build the examples, for instance after some changes are made to the source, Maven has to be installed on the system. The required Maven version is 3.2.0. It is also required to have at least Java 6 and an internet connection.

To build the examples, run:

```
> mvn clean package
```

The bundles can then be found in the `target` directory created by Maven.

Chapter 3

Tools and Components

In order to interact with JamaicaAMS, and to profit from its functional extensions, it is our recommendation that users employ tools such as the ones listed in this section. Either provided by third parties, part of aicas' portfolio or contained in the JDK, those tools are vital to fully optimize the development and couple the framework to the Industrial IoT (IIoT) environment.

3.1 Development

- **Integrated Development Environment** - The IDE of choice to write the bundles is Eclipse. For a practical "step-by-step" example, please refer to Section 6.
- **Emulator** - Developing Apps for the JamaicaAMS is done through the creation of bundles. Besides the flexibility it provides, this style of modular programming will extend your possibilities to create robust and reliable software components, also applying the security mechanisms. A central tool in this bundle development and debugging process is the JamaicaAMS Emulator (see Section 3.1.1).
- **JVMTI Interface** - JamaicaVM implements the Java Virtual Machine Tool Interface to support introspection and debugging of a local or remote Java Virtual Machine. This interface can also be used from inside common Java IDEs.
- **Code Analyzers** - aicas recommends as tools for static source analysis PMD and SpotBugs (see Section 3.1.2).
- **Bndtools** - Open source software, that uses bytecode analysis to accurately calculate the dependencies of OSGi bundles. It features a repository model for bundles, that may be referenced at build-time and can be used to satisfy runtime dependencies. For an introduction to the BND set of tools, please refer to the Bndtools website [2].

3.1.1 Emulator

The Emulator facilitates the testing and debugging of bundles greatly. By using the Emulator, developers can bypass the need to install a written bundle on an external device in order to find errors and bugs in the code.

In principle, the Emulator is the JamaicaAMS that is compiled for the host system, where it is executed in interactive command mode to detect bugs at an early stage of the bundle development. The Emulator can be launched at the host system the same way as in the target system. It can be optionally configured through the command-line parameters **-c**, **-s** and **-b**:

```
> cd <path-to-root-directory-of-JamaicaAMS>/setup

> ./<host>/bin/emulator [-c <config-properties-file>] \
                        [-s <system-properties-file>] \
                        [-b <bundle-cache-dir>]
```

3.1.2 PMD and SpotBugs

These open source tools are two different static source analyzers that are used for the same purpose: to analyze the code that is produced by a developer.

They detect problematic issues in the code, like missing synchronization, empty catch blocks or possible null pointer dereferences, classifying major or minor bugs according to potential severity. Note that code analyzers do not actively change anything in the code.

PMD as well as SpotBugs follow predefined rulesets, that are built in and compared with the code in a project. These rulesets contemplate aspects like bad practices and rules concerning for instance performance, security and correctness.

A brief introduction on how to use those tools follows.

SpotBugs

SpotBugs can easily be installed through the Eclipse Marketplace. It already contains several hundred rules that can also be extended.

After installation, SpotBugs can be configured to fit a specific category, e.g. performance- or bad practice oriented. Severity levels can be set from *20* (least) to *1* (most) severe, with a further separation on how a bug should be marked (error, info or warning). Single rules can be enabled or disabled for a further, more focused, analysis (see Figure 3.1).

This can be configured under **Window** → **Preferences** → **Java** → **SpotBugs**.

The rulesets in SpotBugs can also be extended under the **Plugins and misc. Setting** tab. These extensions are available online.

To use SpotBugs, right click on the preferred project and select **SpotBugs** → **Find bugs**.

For further details on SpotBugs, please refer to its documentation [7].

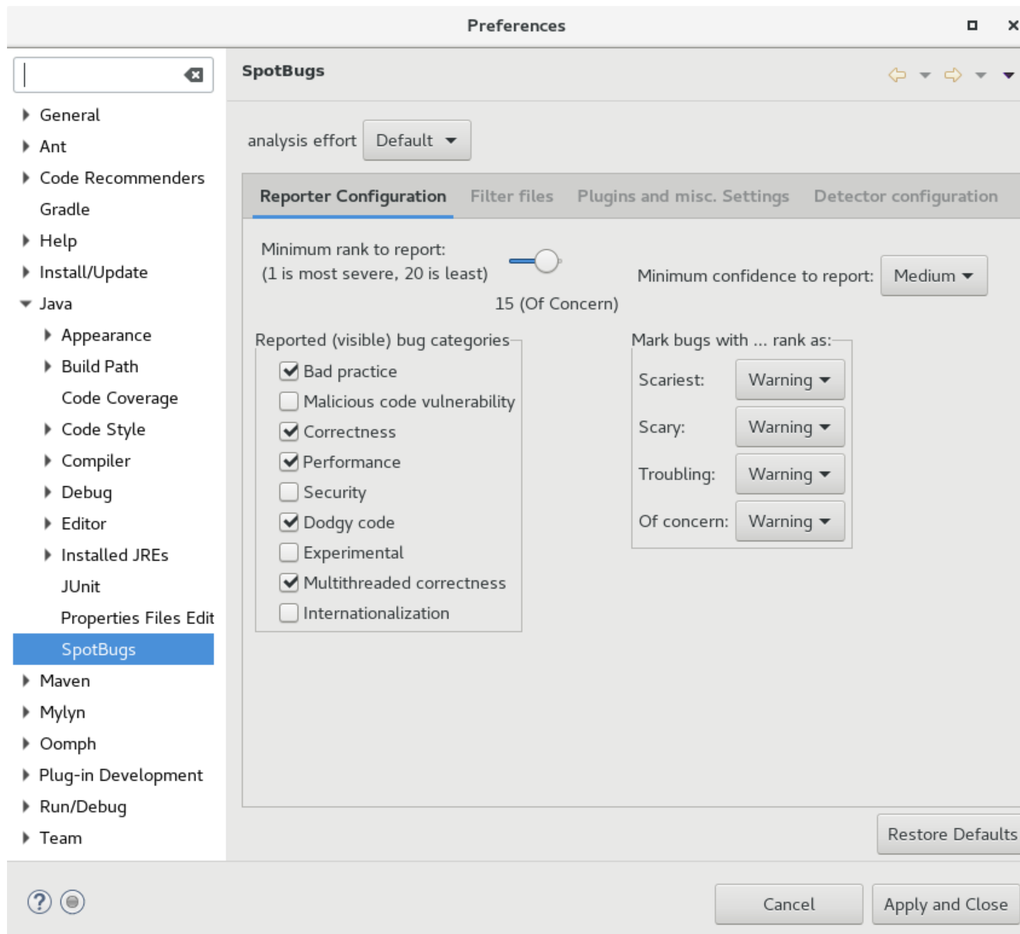

Figure 3.1: SpotBugs can be easily configured in Eclipse

PMD

PMD is as user-friendly as SpotBugs. The plugin we recommend can be downloaded from `https://sourceforge.net/projects/pmd/`. It can also be directly downloaded in Eclipse, from `https://dl.bintray.com/pmd/pmd-eclipse-plugin/updates/`, through **Window** → **Install new Software** → **add**. The tool can be configured in a similar way: **Window** → **Preferences** → **PMD** (see Figure 3.2).

PMD rules are also separated by categories, e.g. “performance”, and severity, e.g. “blocker”. To use PMD, right click on the preferred project and **PMD** → **Check code**.

For further details on PMD, please refer to the online documentation [6].

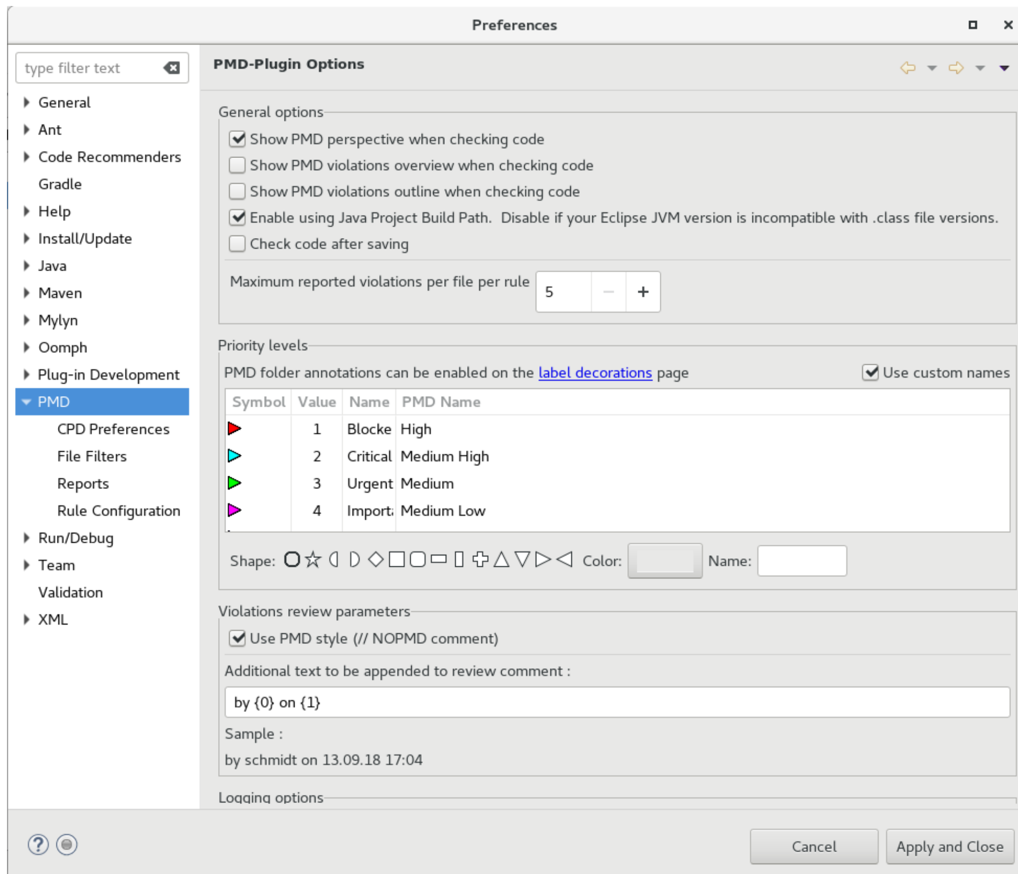

Figure 3.2: PMD is configured in a similar manner to SpotBugs

Expected Results

After running either of these tools, a list of bugs will be shown in Eclipse. Each bug can be selected, to see more detailed information. Both tools also offer an option to export a bug report as a `.txt` or `.xml` file.

In case both tools are deployed on the same project, the results might overlap: However, they will also be complementary.

3.2 Deployment

- **jarsigner** - Signing and verification tool, that uses key and certificate information from a keystore to generate digital signatures for JAR files (see Section 3.2.1).

3.2.1 jarsigner

JamaicaAMS's target runtime permits the execution of a bundle JAR file that is signed with a private key such that the corresponding certificate can be verified by the public key that was put on the target device (see also 4.1.1). To achieve this, the bundle `<bundlename>.jar` must be signed using the following command:

```
> jarsigner -keystore <keystore> <bundlename>.jar <alias>
```

This command uses a key pair generated as in the example below:

Creating a Public/Private Key Pair

The `keytool` that is part of OpenJDK is required to generate a key pair using the following command line:

```
> keytool -genkey -keystore <keystore> -alias <alias> -validity
<validity>
Enter keystore password: <password>
Re-enter new password: <password>
What is your first and last name? John Q. Public
What is the name of your organizational unit? SecurityServices
What is the name of your organization? OEM
What is the name of your City or Locality? Karlsruhe
What is the name of your State or Province?
What is the two-letter country code for this unit? DE
Is CN=John Q. Public, OU=Unknown, O=OEM, L=Karlsruhe, ST=Unknown,
C=DE correct? [no]: yes
Enter key password for <<alias>>
        (RETURN if same as keystore password): <RETURN>
```

Here, `<keystore>`, `<alias>`, and `<validity>` can be chosen freely.

It is extremely important that the generated file, `<keystore>`, remains private. It is recommended to install it only on a machine that is not connected to a network and to limit access to a minimum number of people. Also, the keystore should be protected by a strong password.

3.3 Configuration

- **Profiling** - Profiler is JamaicaVM's function to improve the performance of an application. *Profiling* means to identify the parts of the code that contribute most to the overall runtime. These parts are identified in a *profile run* of the application. For specific information on how to profile in JamaicaAMS, please see Section 3.3.1.
- **Bundle Configuration** - The configuration of multiple bundles is made easier through the implementation of the OSGi Configuration Admin service and the Configurator (see Section 3.3.2).

3.3.1 Profiling and JamaicaAMS

JamaicaAMS is offered with an additional profiling binary, called the `jamsp` file, that could be chosen by the user to start the framework. This profiling binary is a particular version of JamaicaAMS that allows the user to collect information related to the runtime execution.

JamaicaAMS profiling binary

The JamaicaAMS profiling binary collects information on the amount of runtime spent for the execution of different methods. This information is dumped to a file (in `/tmp/jamaica-ams.prof`) after a test run of the application has been performed. This collection of profile information is cumulative; that is, when this file exists, profiling information is appended.

The XPROF Environment Variable

Another way to optimize performance is to start JamaicaAMS with the Environment Variable **JAMAICA_AMS_XPROF** enabled. This variable enables collecting simple profiling information using periodic sampling. For detailed information on this subject, please refer to the JamaicaVM User Manual[3] (cf. Section 12.1.2, option **-Xprof**).

To execute JamaicaAMS with the periodic sampling enabled, an Environment Variable needs to be set before JamaicaAMS starts. To export the Environment Variable do as follows:

```
> export JAMAICA_AMS_XPROF=<value from 0 to 1000>
```

Example:

```
> export JAMAICA_AMS_XPROF=100
```

The value to specify is the number of profiling samples to be taken per second, e.g. in the example showed here, 100 samples per second.

This profile is used to provide an estimate of the methods which use the most CPU time during the execution of an application. During each sample, the currently executing method is determined

and its sample count is incremented, independent of whether the method is currently executing or is blocked waiting for some other event.

The total number of samples found for each method are printed when the application terminates.

3.3.2 Bundle Configuration

Bundles may load configuration data from the jar, from a local file or from a remote location; and they often require additional configuration, depending on the deployment environment. Furthermore, when multiple bundles require, each of them, particular configuration files, the result is increased complexity: duplicated files and exception handling code, incoherent file locations and different fetching procedures.

As a solution, OSGi offers specification for the Configuration Admin service and the Configurator bundle. JamaicaAMS packs the Apache Felix implementation of these bundles in `setup/bundle.1` of its distribution tree. This section explains how to use these bundles to access, parse and store configurations in a uniform way.

Note that...

the Apache Felix Configuration Admin and Configurator bundle implement chapters 104 (`https://osgi.org/specification/osgi.cmpn/7.0.0/service.cm.html`) and 150 (`https://osgi.org/specification/osgi.cmpn/7.0.0/service.configurator.html`) of the OSGi compendium specification, being this document the absolute reference.

3.3.2.1 Configurator

Instead of loading the needed configuration data from scattered files, the bundles load the information from the Configuration Admin service. This service acts as a database, simply storing and providing the configurations which are delivered by the Configurator. The role of the Configurator is therefore to fetch and parse configuration files, and then to add the information to the Configuration Admin service.

The Configurator can load files from the local file system, from the bundle jar or from a remote location, supporting the HTTP and FTP protocols. It requires a URL to the file that it should get, and this must contain the protocol to be used (e.g., file:.conf/config1.json, http://www.yourserver.com/standardConfig.json).

Note that...

a list of the URLs for the files to be loaded must be present in the configurator.initial property, which can be modified in the conf/config.properties file.

Configuration File Format

A configuration file may contain multiple configurations, which are identified using a persistent identifier (PID). PIDs are used by the bundles to request a certain configuration from the configuration admin service. Those PIDs that start with the ":configurator:" prefix contain information or instructions that are relevant to the Configurator.

An example of PID and configuration can be seen in the listing below:

```
{
  // Resource Format Version
  ":configurator:resource-version": 1,

  // First Configuration
  "pid.a":
    {
      "key": "val",
      "some_number": 123
    },

  // Second Configuration
  "pid.b":
    {
      "a_boolean": true
    }
}
```

Configuration files must be written in the JSON format. When those files are not loaded from the bundle jar, it is mandatory to provide a symbolic-name and a version for the configuration (for an example, see table 150.1 in section 150.3.1 in `https://osgi.org/specification/osgi.cmpn/7.0.0/service.configurator.html`).

3.3.2.2 Configuration Admin Service

As already mentioned, the Configuration Admin acts as a database for bundle configurations, storing them persistently (in the JamaicaAMS cache) and distributing them to concerned bundles whenever they are updated. The configurations are handled as java.lang.Dictionary objects and stored in the .properties format.

Retrieving Configurations

There are two ways of interacting with the Configuration Admin: Either synchronously, by fetching the current available configuration through the ConfigurationAdmin interface, or asynchronously, by registering a ManagedService that gets notified whenever its corresponding configuration is updated.

Both methods of accessing the bundle configuration are described in the specification, as they are part of the Configuration Admin service, and are not subject to changes in the different implementations. Thus a bundle that uses the code discussed in the following subsections will work with any implementation of the Configuration Admin.

Synchronous Access

Assuming that an implementation of the Configuration Admin service is installed and a corresponding configuration dictionary is available, a bundle can retrieve and update this configuration using the

code presented in te following listing. If no configuration exists, the same code will result in a new configuration.

```
// retrieve the service interface
ServiceReference<ConfigurationAdmin>serviceReference=
context.getServiceReference(ConfigurationAdmin.class);
ConfigurationAdmin configurationAdmin=null;
if (serviceReference!=null)
  {
    configurationAdmin=(ConfigurationAdmin)
    context.getService(serviceReference);

    // fetch the current configuration
    Configuration configuration=
    configurationAdmin.getConfiguration("ConfigUserTest");
    Dictionary props=configuration.getProperties();
    // if null,the configuration is new
    if (props==null)
      {
        props=newHashtable();
      }
    // set some properties
    props.put (..., ...);
    // update the configuration
    config.update (props);
  }
```

Asynchronous Access

A different approach is that bundles can be reconfigured by a configuration provider while running, thus achieving highly configurable applications. A configuration listener must implement the ManagedService interface, which provides the notification method updated(Dictionary). The listener is then to be published as a service, in association with the configuration name it wants to be updated with.

This approach is also associated with a greater adoption effort, since it imposes a substantial change to the design of the involved bundles. The possibility of updating a bundle with new configurations imposes new logical states on the bundle (which could be called “unconfigured”, “configured” and “running”), while it is in the “active” state. Furthermore, the configuration update occurs in a thread that belongs to the Configuration Admin (thus the name “asynchronous approach”), which may require further handling depending on the code to be executed. For example, if no configuration is present initially, the bundle start method must register a ManagedService and use the update thread to actually initiate the bundle.

The following listing exemplifies a configuration listener published as an OSGi service, which will get notified whenever the associated configuration is changed.

```
private ServiceRegistration configListenerRegistration;

public void start(BundleContext context)
{
```

```
    Dictionary props=new Hashtable();
    props.put("service.pid","configurationNameOrPersistanceID");
    configListenerRegistration=
    context.registerService(ManagedService.class.getName(),
    new ConfigurationListener(),props);
  }

  public void stop(BundleContext context)
  {
  if (configListenerRegistration!=null)
    {
      configListenerRegistration.unregister();
      configListenerRegistration=null;
    }
  }
```

Persistence Managers

In cases where the storage logic needs to be overwritten completely (e.g., when the configuration needs to be stored remotely or through a database manager), a persistence manager can be used.

The persistence manager created by the user must provide basic functionality (loading, storing and deleting dictionaries) and must be published as an OSGi service. It needs to implement the interface PersistenceManager present in the package `com.aicas.jamaica.ams.org.apache.felix.cm`, which can be found in the file `<path to JamaicaAMS>/setup/bundle.1/configuration-admin-<version>.jar`.

The Configuration Admin can be configured to use a certain persistence manager by providing its name in the `felix.cm.pm` property.

Note that...

the persistence manager mechanism is specific to the Apache Felix implementation, not being part of the OSGi specification of the Configuration Admin service. The PersistenceManager interface present in the JamaicaAMS distribution is the same one created by Apache Felix and documented in `http://felix.apache.org/apidocs/configadmin/1.6.0/org/apache/felix/cm/PersistenceManager.html`.

The developer must however consider the different package name, which is, as mentioned above:

`com.aicas.jamaica.ams.org.apache.felix.cm`.

Chapter 4

Security

The OSGi Security Layer is based on the Java security architecture. It provides the infrastructure to deploy and manage applications that must run in controlled and resource constrained environments.

For JamaicaAMS, the primary requirement for security is to ensure that only code that has been approved can run on the framework; and that code can only use those facilities which are approved. This means that JamaicaAMS must know how to determine whether the source which distributes the code is an approved one, and what facilities that code may use. JamaicaAMS uses X.509 certificates and the Java security manager provides a sound security mechanism.

As part of its security mechanisms, JamaicaAMS enables the system owner to determine who may write applications (bundles) for its application management system and to provide fine-grained restrictions on what each application may access in the system. Each application provider is identified via one or more certificates with associated permissions.

4.1 Terms and Definitions

The following are some common terms that are useful to describe the JamaicaAMS security model.

4.1.1 Public/Private Key Pair

Public key cryptography is the base for Digital Signing. It matches a pair of mathematically related keys used by an asymmetric key algorithm: a public (PuK) and a private (PrK) key. The *Public Key* may be distributed freely, while the corresponding *Private Key* should only be known to the user. Messages signed with the private key can only be verified correctly with the public key. In the "Java World" keys are often stored in `KeyStores` (`*.jks` files).

Messages signed with the private key can only be verified correctly with the public key. This can be used to authenticate the signer of a message (assuming the public key is trusted).

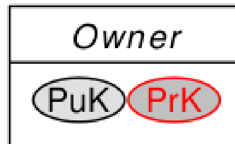

Figure 4.1: A Public/Private Key Pair

4.1.2 Certificates and Chains

The JamaicaAMS framework uses a common standard for certificates: X.509. The information commonly stored in a X.509 certificate is shown in Figure 4.2.

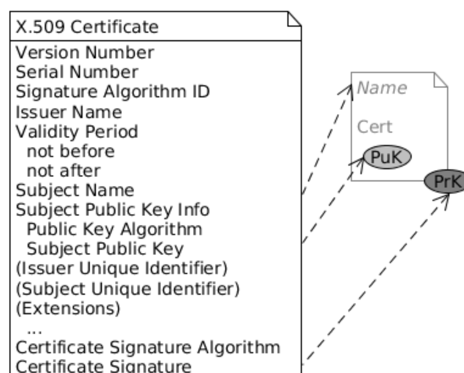

Figure 4.2: X.509 Certificate

Certificates can be arranged in a chain, forming a so called *Chain of Trust*. For this the following conditions must be met:

- The issuer of each certificate (except the last one, the *Root of Trust*) matches the subject of the superordinate certificate.
- Each certificate (except the last one) is signed by the *Private Key* of the superordinate certificate. Hence, its signature can be verified using the *Public Key* of the superordinate certificate.

An example of *Chain of Trust* is depicted in Figure 4.3.

Note that...

Even though called "chain" according to the view from the bottommost certificate towards the root, certificate chains can span a tree with multiple instances at any level (except the root). This way, in case a certificate issued at a parent level becomes compromised, all its children can be tracked with their certificates revoked. This breaks the chain back to the root and makes the signer invalid.

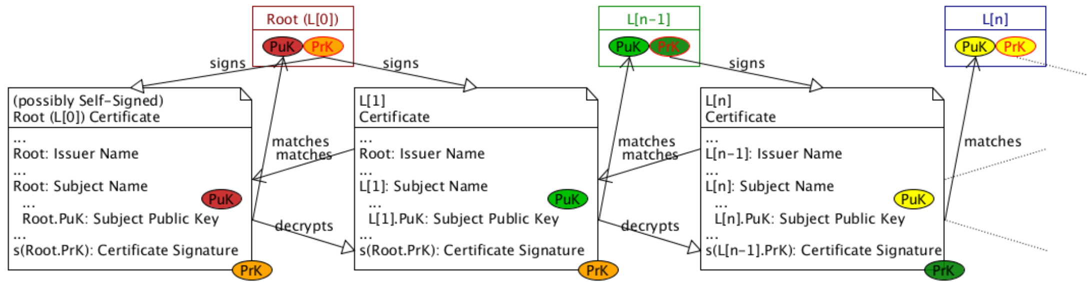

Figure 4.3: A Certificate Chain

4.2 Signed JAR File

A fundamental way for an OSGi framework to authenticate code is to have the Java code signed: this makes it possible to put JAR files as leaves (final instances) into a *Chain of Trust*. Depending on the implementation, specific services that manage the permissions associated to the authenticated code are defined. Figure 4.4 shows the security-related content of a JAR file.

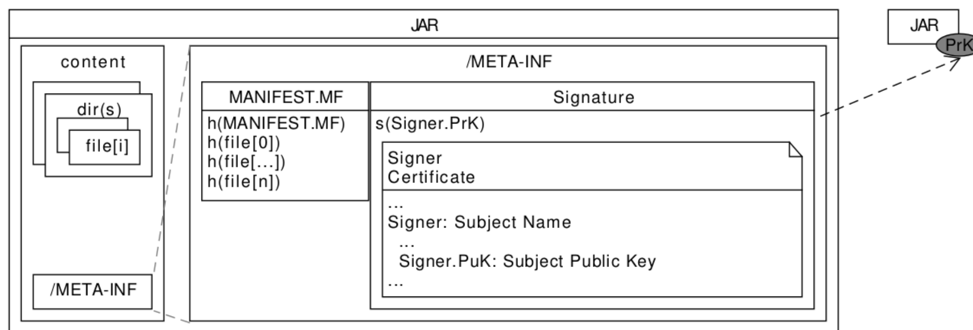

Figure 4.4: A Signed JAR file

However, before a JAR is signed, the creation of a public/private key pair and the corresponding certificate is needed. Furthermore, it must be ensured that the public key that verifies the certificate is present on the target device. The involved certificate must be included in the trust store specified in Section 4.4

For more information on the process of signing a JAR file, please refer to Section 3.2.1.

Note: In its Core document for the Release 8, the OSGi Alliance informs:

JAR Structure and Manifest

OSGi JARs must be signed by one or more signers that sign all resources except the ones in the META-INF directory; the default behavior of the jarsigner tool. This is a restriction with respect to standard Java JAR signing; there is no partial signing for an OSGi JAR.

The OSGi specification only supports fully signed bundles. The reason for this restriction is because partially signing can break the protection of private packages. It also simplifies the security API because all code of a bundle is using the same protection domain.

Signature files in nested JAR files (For example JARs on the Bundle-ClassPath) must be ignored. These nested JAR files must share the same protection domain as their containing bundle. They must be treated as if their resources were stored directly in the outer JAR.

Each signature is based on two resources. The first file is the signature instruction file; this file must have a file name with an extension .SF. A signature file has the same syntax as the manifest, except that it starts with Signature-Version: 1.0 instead of Manifest-Version: 1.0.

The only relevant part of the signature resource is the digest of the Manifest resource. The name of the header must be the name algorithm (e.g. SHA1), followed by -Digest-Manifest. For example:

Signature-Version: 1.0

SHA1-Digest-Manifest: RJpDp+igoJ1kxs8CSFeDtMbMq78=

MD5-Digest-Manifest: IIsI6HranRNHMY27SK8M5qMunR4=

The signature resource can contain name sections as well. However, these name sections should be ignored.

If there are multiple signers, then their signature instruction resources can be identical if they use the same digest algorithms. However, each signer must still have its own signature instruction file. That is, it is not allowed to share the signature resource between signers.

Source: `https://docs.osgi.org/download/r8/osgi.core-8.0.0.pdf`.

4.3 Authentication and Permissions

In order to ensure that a tampered bundle will be detected by JamaicaAMS's security mechanism, before a bundle can be distributed for use on the target device, it needs to be signed by the platform provider or a trusted entity using a public/private key signature. All files in the JAR need to be included in the signatures, including resources and not only class files.

On the target device, JamaicaAMS verifies the signature against a public key that is pre-installed on the device. Permissions of all accesses to resources must be checked and resource budgets must be enforced during execution.

Permissions

The OSGi Framework uses `Java 2` permissions for securing bundles. Each bundle is associated with a set of permissions. During runtime, the permissions are queried when a permission is requested through the Security Manager which is a software component in charge of permission validation for the bundles.

The management of the bundle's permissions is handled through "Conditional Permission Admin", "Permission Admin", or another security agent. The Apache Felix implementation includes "PermissionAdmin" and "ConditionalPermissionAdmin", provided by a "framework.security" extension bundle. JamaicaAMS incorporates the "org.apache.felix.framework.security" as "bundle.1" into the framework.

4.4 Activating Security in JamaicaAMS

JamaicaAMS contains implementations of the OSGi security layer and a policy file reader in the form of auto-deploy bundles. In order to activate security, create a system configuration file with the following content:

```
org.osgi.framework.security=osgi
java.security.policy=<path to a policy file>
jamaica-ams.security.policy=<path to an OSGi  policy file>
org.osgi.framework.trust.repositories=<path to a trust store>
```

It is the policy file (`all.policy`) that defines the general permissions; it should contain:

```
grant {
    permission java.security.AllPermission;
};
```

A policy that provides no permissions at all would be empty, as seen below.

```
grant
{
};
```

More specifically, the OSGi policy file (`osgi.all.policy`) defines the permissions for the bundles. The simple example below shows how any bundle could be granted permission to call all restricted APIs.

```
ALLOW {
    (java.security.AllPermission "*" "*")
} "Give AllPermission to all Bundles"
```

Finally, the trust store which is specified by the property **org.osgi.framework.trust.repositories** should contain the trust anchors to be used by the framework. Only OSGi bundles signed with a private key and whose certificate has a *Chain of Trust* to one of these trust anchors are considered as properly signed bundles. Then the policy file can be used to specify that only the properly signed bundles can be installed and started.

4.5 An Example of Security Configuration

In this section an example will be used to summarize the information provided in the previous sections. The instructions are performed on a Linux machine and the following files are involved:

- `primes-<version>.jar`: The example bundle.
- `TestKey.jks`: A keystore for signing bundles (alias: test1, password: password)
- `conf/osgi.policy`: An OSGi policy file that restricts which bundles may be installed.
- `conf/system.properties`: System properties to enforce that `osgi.policy` is used.
- `conf/trust.jks`: A truststore containing the certificate of the example bundle signer.

The process of creating the keystore file `TestKey.jks` and the truststore file `conf/trust.jks` is divided in three steps.

1. Create the keystore for signing bundles using `keytool`. In the root directory of the JamaicaAMS distribution, type the following command to create `TestKey.jks`. For more information about `keytool` refer to Section 3.2.1.

   ```
   > keytool -genkey -keyalg RSA
   -alias test1 -keystore TestKey.jks
   -storepass password -keypass password
   -validity 365 -keysize 2048
   -dname "cn=Demo, ou=JamaicaAMS, o=aicas GmbH, l=Karlsruhe,
   c=DE"
   ```

2. Export the certificate of the keystore created in the previous step. In this case, a certificate file `host.crt` is to be created with the following command. Note that the command requires the password that has been set for the keystore.

```
> keytool -export -alias test1 -keystore TestKey.jks
-file host.crt
```

3. Import the certificate to the truststore file `conf/trust.jks`. The file will be generated in the folder `conf` if it does not exist.

```
> keytool -import -v -trustcacerts
-alias test1 -file host.crt
-keystore conf/trust.jks
```

Set a password for the truststore after the following request:

```
Enter keystore password:
Re-enter new password:
```

Type “`yes`” in the terminal, when asked whether the imported certificate can be trusted:

```
Trust this certificate? [no]:  yes
Certificate was added to keystore
[Storing conf/trust.jks]
```

4.5.1 Sign the Example Bundle

It is important to notice that the bundle is provided unsigned. In order to sign the bundle, do as follows:

1. Open a terminal and go to the JamaicaAMS root directory, where the file `TestKey.jks` is.
2. Enter in the terminal:

```
> jarsigner -keystore TestKey.jks example/primes-<version>.jar
test1
```

3. Enter the password for the keystore (PW = password). Now the JAR file is signed.

4.5.2 Configure the System Properties

Adapt the `system.properties` to select the policy file and the trust repositories. In particular, the property `org.osgi.framework.security` shall be set to `osgi` to enable the security. The property `jamaica-ams.security.policy` shall point to the security policy file which is in this case the `osgi.policy` file. The property `org.osgi.framework.trust.repositories` shall point to the truststore `trust.jks`.

```
# The policy where the policy rules are defined
jamaica-ams.security.policy=./conf/osgi.policy

# Trust anchors for signature checks
org.osgi.framework.trust.repositories=./conf/trust.jks

# Enable security
org.osgi.framework.security=osgi
```

4.5.3 Configure the Policy

The `osgi.policy` should contain the following rules. The signer information needs to be matching the information that is present in the truststore. This is the certificate matching the key that was used for signing the bundle.

```
ALLOW {
  (org.osgi.framework.AdminPermission
  "(signer=CN=Demo, OU=JamaicaAMS, O=aicas GmbH, L=Karlsruhe, C=DE;-)"
  "lifecycle")
} "Allow installation of properly signed Bundles"
DENY {
  (org.osgi.framework.AdminPermission "*" "lifecycle")
} "Deny installation of all other Bundles"
ALLOW {
  (java.security.AllPermission "*" "*")
} "Give AllPermission to all Bundles"
```

4.5.4 Set-Up and Operation

1. As a next step, the signed bundle will be installed and demonstrated. The `emulator` is used for the purpose of this demonstration. To do so:
   - Open a terminal, go to the setup directory of the JamaicaAMS distribution and start JamaicaAMS by entering "`./<host>/bin/emulator`"
   - Enter "`install ../example/primes-<version>.jar`"
   - Enter "`start`" ID and "`stop`" ID to run and stop the bundle
2. After running the signed versions of the bundle "primes", try to install the unsigned bundle `example/primes-with-budget-<version>.jar` to get the security check and the *"Access denied"* message displayed.
3. To see the IDs, you can enter "`lb`" (for "list bundles")

Chapter 5

OSGi Framework and Bundles

5.1 Framework Layers

OSGi offers a software layer, running on top of a Java platform, that supports the design and implementation of modular systems. It does that by specifying a secure infrastructure that enables the distribution, interoperability and remote management of application- and service components, called bundles and sometimes also referred as packages.

The central part of the specification is the framework, that defines a model to manage the lifecycle of the bundles, also including a registry, that exposes functionalities made available as services to be imported and exported by the bundles, and an execution environment.

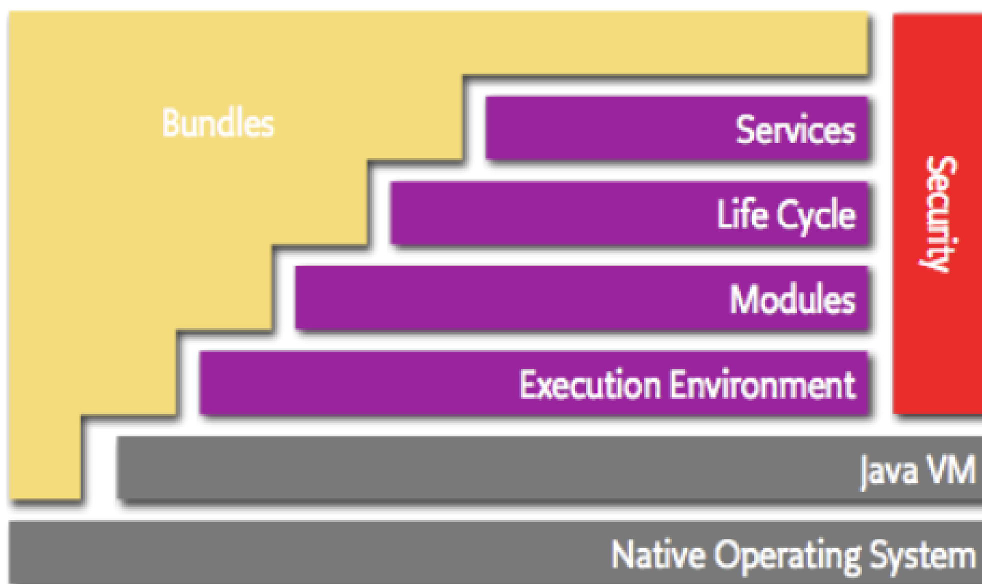

Figure 5.1: OSGi Framework: Layers Overview

Source: `https://www.osgi.org/developer/architecture/layering-osgi/`

5.2 Bundle Lifecycle

The Lifecycle Layer provides an API for controlling the different phases of the bundle operations.

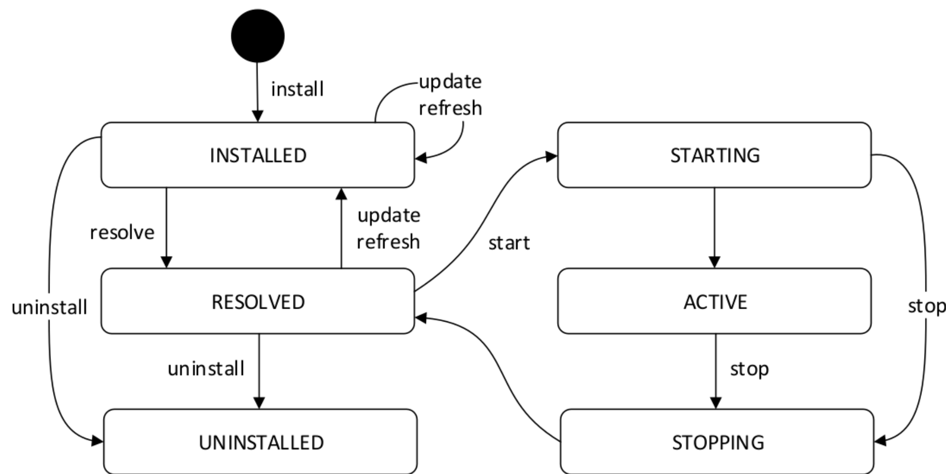

Figure 5.2: Bundle's lifecycle

Figure 5.2 shows the states that a bundle can experience in a lifecycle, as well as the possible transitions it could make to other states. Those can be summarized as follows:

- **Installed:** The bundle was successfully installed.
- **Resolved:** This state evaluates if the bundle is ready (complete), by checking things like the Java version, if the imported packages are available etc.
- **Starting:** This is a transition state in the lifecycle; in this state, the method that activates and starts the bundle is called.
- **Active:** In this state, the bundle has been successfully validated and is running.
- **Stopping:** Another transition state; the method that stops the bundle is being executed and the bundle is being stopped.
- **Uninstalled:** The bundle is uninstalled and cannot be induced into another state.

5.3 Service Orientation

The service layer of the Framework rules the way the bundles dynamically connect and collaborate, through a process known as "publish-find-bind".

The concept of service in the framework's context involves the following aspects:

- **Specification:** An interface that specifies the public methods.
- **Implementation:** A Java class that implements the methods specified in the interface.
- **Registry:** Where the services are included and made available to other bundles.
- **Customers:** The bundles that make use of a given service.

5.4 Controlling the Bundles

The control of the bundles' lifecycle as well as other functionalities can be done through administration consoles or via command line, depending on the implementation. This way, any bundle can be dynamically installed or uninstalled without the application having to be stopped or restarted. Also information about active services, the listing of imports and exports of each bundle and the metadata contained in the headers are shown this way.

As mentioned in Section 2.3.3, JamaicaAMS provides the Apache Felix Gogo shell as an auto-deploy bundle. The command "`lb`" lists the bundles and their present states, as seen below.

```
g! lb
START LEVEL 3
   ID|State      |Level|Name
    0|Active     |    0|System Bundle (0.0.0)|0.0.0
    1|Active     |    1|JamaicaAMS Configuration Admin (0.1.0.SNAPSHOT)|0.1.0.SNAPSHOT
    2|Active     |    1|Apache Felix Configuration Json (1.0.6)|1.0.6
    3|Active     |    1|Apache Felix Configurator Service (1.0.14)|1.0.14
    4|Active     |    1|Apache Felix Converter (1.0.18)|1.0.18
    5|Active     |    1|Apache Felix Log Service (1.2.4)|1.2.4
    6|Active     |    1|Apache Sling Commons Johnzon Wrapper Library (1.2.6)|1.2.6
    7|Active     |    1|org.osgi:org.osgi.util.function (1.1.0.201802012106)|1.1.0.201802012106
    8|Active     |    1|JamaicaAMS OSGi Log Writer (0.1.0.SNAPSHOT)|0.1.0.SNAPSHOT
    9|Resolved   |    3|JamaicaAMS Security Provider (0.1.0.SNAPSHOT)|0.1.0.SNAPSHOT
   10|Active     |    2|JamaicaAMS Policy File Reader (0.1.0.SNAPSHOT)|0.1.0.SNAPSHOT
   11|Active     |    3|Apache Felix Gogo Command (1.1.2)|1.1.2
   12|Active     |    3|Apache Felix Gogo Runtime (1.1.4)|1.1.4
   13|Active     |    3|Apache Felix Gogo Shell (1.1.4)|1.1.4
g!
```

Chapter 6

How to write a Bundle with Eclipse

This section describes how to write an OSGi bundle for JamaicaAMS with the Eclipse IDE.

6.1 Prerequisites

The steps described in this section assume the following knowledge:

- General experience with a Java Runtime Environment
- General experience in writing applications in the Java programming language
- Basic understanding of the OSGi framework, especially of the bundle's interface and lifecycle
- General experience with the Eclipse IDE [9]

The steps described in this section require the following setup to be present:

- An up-to-date version of the Eclipse IDE from [8]. The required minimum version is 3.6, but the latest version is recommended.
- The installed Plug-In Development Environment (PDE) [10]. If the PDE plug-in is not contained in your Eclipse distribution, it can be obtained using the Eclipse Update manager, by choosing **Menu** → **Help** → **Install New Software**

6.2 Bundle Tutorial

6.2.1 Create a new Plug-In Project

1. Create a new Plug-in project by navigating to **File** → **New** → **Project**. In the **New Project** wizard choose **Plug-In Development** → **Plug-in Project**

2. In the **New Plug-in Project** wizard on the **Plug-in Project** page enter `HelloWorld` into the **Project Name** textfield.

   Leave the **Project Settings** options unchanged.

   Set the **Target Platform** options to match “This plug-in is targeted to run with: an OSGi framework: standard”.

   Proceed by clicking the [**Next >**] button.

3. On the **Content** page leave the **Properties** options unchanged. Make sure that in the **Options** section, the **Generate an activator, a Java class that controls the plug-in’s life cycle** checkbox is checked.

   Proceed by clicking the [**Next >**] button.

4. On the **Templates** page you can see a selection of project templates.

   Choose **Hello OSGi Bundle** for now.

   Confirm by clicking the [**Next >**] button.

5. The template **Basic OSGi Bundle** will let you choose a message to be displayed on start and one on stop respectively. Freely choose any or keep with the defaults.

6. End this procedure by clicking the [**Finish**] button. At this point, Eclipse may suggest switching to the **Plug-in Development** *Perspective* because this is the default for Plug-in Projects. Please do so.

6.2.2 Make Yourself Familiar with the UI

Verify that you have successfully finished the first stage by finding the `HelloWorld` project with `src` and a *Manifest* as `META-INF/MANIFEST.MF` in the *Workbench*’s **Packages / Package Explorer** *View*.

If you never have used the **Plug-in Development** *Perspective* before, it might be a good moment to make yourself familiar with it by, e.g., having a look how `META-INF/MANIFEST.MF` is represented.

6.2.3 Implement the Functionality

There is literally nothing that needs to be done. See how `src/helloworld/Activator.java` implements `BundleActivator` with the input you chose in the wizard.

6.2.4 Run the Bundle on the Integrated Framework

As a next step you might want to run the newly created bundle. You can run it in the Equinox OSGi framework, which is a fundamental part of every Eclipse installation.

1. Select the `HelloWorld` project that contains your OSGi bundle.
2. Choose **Run** → **Run Configurations...**. This opens the **Run Configurations** dialog.
3. Create a new **OSGi Framework** configuration.
4. On the **Bundles** tab, please ensure that Equinox is set as runtime framework and select the following bundles:

   For Eclipse 3.6 and 3.7 users:

   - `HelloWorld`
   - `org.eclipse.osgi`

   For Eclipse 4.x users:

   - `HelloWorld`
   - `org.eclipse.osgi`
   - `org.eclipse.equinox.console`
   - `org.apache.felix.gogo.command`
   - `org.apache.felix.gogo.runtime`
   - `org.apache.felix.gogo.shell`
5. On the **Settings** tab, enable the **Clear the configuration area before launching** checkbox.
6. Now click the [**Apply**] button to save the configuration and then press **Run** to launch an Equinox instance with the selected bundles. The **Console** *View* will show you the expected output of the `Activator` class and the OSGi shell.

Since you have started a regular OSGi framework, you can start and stop your bundle by using the commands `start <bundle-id>` and `stop <bundle-id>`. Use the `help` command to get a list of the most important OSGi commands.

6.2.5 Deployment

As a next step you might want to deploy your bundle to a target system. To obtain a `Java Archive (JAR)` file for your bundle, in order to install it into any standard OSGi framework, take the following steps:

1. Select the `HelloWorld` project in the **Package Explorer** *View*.
2. Go to **File** → **Export...**.
3. In the **Export** wizard, on the **Select** page choose **Plug-in Development** → **Deployable plug-ins and fragments**.

4. Proceed by clicking the [**Next >**] button

5. On the **Deployable plug-ins and fragments** page, in the **Destination** tab select the **Directory** option and determine the place where to put the bundle *JAR* by editing the textfield.

   Shortcut Hint:

   - By choosing the *Bundle Directory* of an OSGi framework here, the framework will use the bundle on next start.
   - If the framework has a file install mechanism activated, you can use this option or the **Install into host** option to install the bundle on-the-fly.

6. Click the [**Finish**] button to create the bundle *JAR* file.

Chapter 7

Debugging Bundles with Eclipse

This section describes how to debug an OSGi bundle with the Eclipse IDE. Besides the traditional debug facilities in Eclipse for debugging on the local host machine, here the focus is especially set on remote debugging. The latter is particularly required when a bundle is executed and debugged directly on a target platform running JamaicaAMS.

7.1 Prerequisites

The steps described in this section assume the following knowledge:

- General experience with a Java Runtime Environment
- General experience in debugging applications in the Java Programming Language
- Basic understanding of the OSGi framework, especially the Bundle interface and lifecycle
- General experience with the Eclipse IDE, especially how to debug applications using Eclipse IDE

The steps described in this document require the following setup:

- The minimum required Eclipse IDE is version 3.6, but the latest version is recommended.
- Java Virtual Machine (JVM) V5.0 or later must be used, such as JamaicaVM or OpenJDK 8.
- A JamaicaAMS release

7.2 Background

Debugging an OSGi bundle can be accomplished in several different ways for different scenarios, e.g.:

- a bundle is deployed and debugged locally in the OSGi environment hosted by the Eclipse IDE;
- a bundle is running in a custom target OSGi environment apart from Eclipse (e.g. JamaicaAMS) and is debugged remotely;
- the target OSGi framework may run on the local or on a remote host

However, in either case the Eclipse IDE serves as an interface for debugging, which may connect to an integrated or a separate OSGi platform. In the case of JamaicaAMS, the Debuggee, i.e. the bundle to be debugged, is always executed within a dedicated JamaicaAMS framework running on a local or a remote host. In both cases, Eclipse IDE connects remotely to the target JamaicaAMS framework.

In order to facilitate debugging activities, the JamaicaAMS distribution provides two binary applications that have been built by Jamaica Builder with JVMTI agent enabled (JVMTI [1]). These applications are located in:

- `<JamaicaAMS>/setup/<target>/bin/jamsdi`
- `<JamaicaAMS>/setup/<host>/bin/emulatordi`

Both `jamsdi` and `emulatordi` are JamaicaAMS components that are compiled for the target system and host system, respectively.

The following example shows how to remotely debug a JamaicaAMS built-in bundle (`<JamaicaAMS>/example/primes-<version>.jar`, the debuggee) using the `emulatordi` and Eclipse IDE.

7.3 Setup of the Debugging Environment

To debug the “primes” bundle, setup the following environments:

- Create a Plug-in Project from the extracted Bundle JAR that is located at `<path to JamaicaAMS>/example/primes-<version>-sources.jar` and set a breakpoint (e.g., somewhere in the `PrimesActivator.start()` method). Figure 7.1 shows an example of a breakpoint that stops the execution before checking whether a number is prime or not.
- Copy the `<path to JamaicaAMS>/example/primes-<version>.jar` into the folder `<path to JamaicaAMS>/setup/bundle.3`, in order to auto start the bundle. You can also manually export the “primes” project as a bundle and install the bundle when JamaicaAMS runs.

[1] https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html

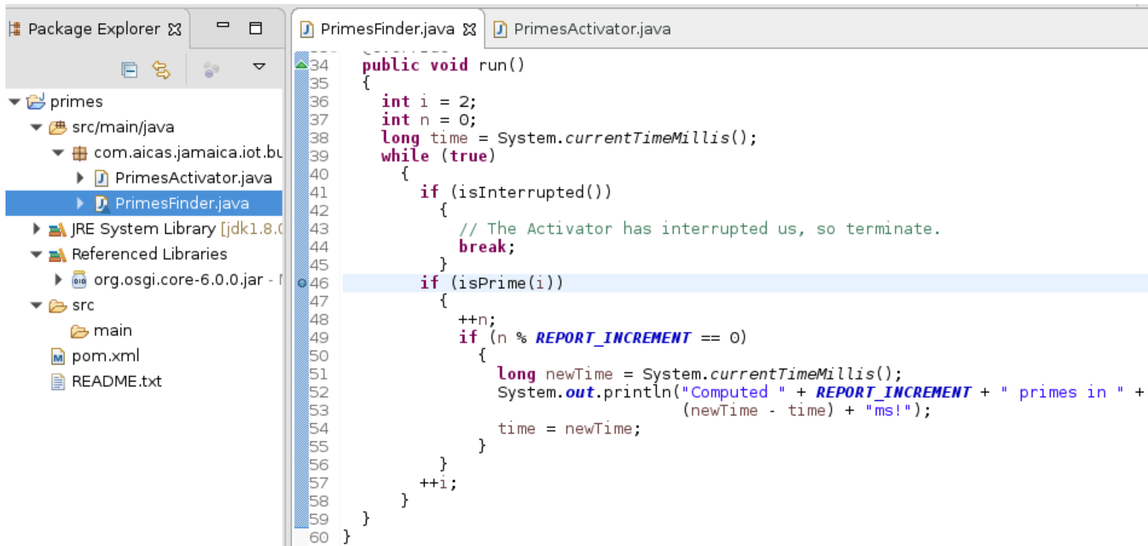

Figure 7.1: Setup the Java project for Computing Prime numbers

7.3.1 Run the Debug Server

Supposing the bundle “primes” is in the folder `<JamaicaAMS>/setup/bundle.3`, type the following command to start the `emulatordi` and you will see the message below it.

```
> ./<host>/bin/emulatordi
```

```
Listening for transport dt_socket at address: 4000
```

This shows that the debug server (`emulatordi`) listens for a socket connection on the pre-defined port 4000 and will be suspended until the debugger connects.

7.3.2 Run the Debug Client

In Eclipse, right click the Class `PrimesFinder.java` that contains a breakpoint and then select the **Debug as** option, followed by the **Debug Configurations...** option. In the [**Debug Configurations**] dialog on the **Connect** tab, choose the following options (depicted in Figure 7.2).

- ConnectionType: `Standard (Socket Attach)`
- Host: `localhost`
- Port: `4000`
- Allow termination of remote VM

Click the [**Debug**] button and you can debug the “primes” bundle as usual.

Running the debug server `jamsdi` is similar to using `emulatordi`, except for changing the **Host** option on the **Connect** tab of the [**Debug Configurations**] dialog to the IP address of the target system that runs the bundle.

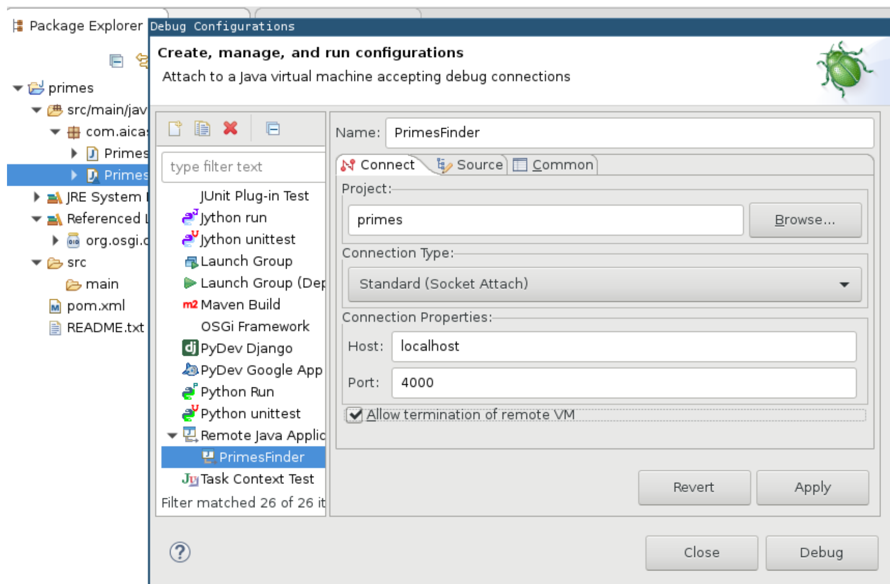

Figure 7.2: Debug Configurations for debugging with `emulatordi`

Chapter 8

JamaicaAMS Runtime Reference

8.1 JamaicaAMS Properties

In the JamaicaAMS `setup/conf` directory there are three configuration files: `config.properties`, `system.properties` and `logging.properties`. The `config.properties` is typically used to configure the behavior of the framework and the bundles; the `system.properties` is purely a convenience, to avoid having to set complicated system properties via command line. JamaicaAMS only directly loads the configuration properties.

When JamaicaAMS starts, its launcher is activated and configures the system by passing the properties loaded from `system.properties` to the framework. Though it is allowed to specify duplicated properties in both configuration files, the configuration properties will override the system properties for duplicated cases.

Note that...

A complete description of the OSGi launching properties implemented by JamaicaAMS can be found in section 4.2.2 of the OSGi Core Release 8 specification [5].

8.1.1 Config Properties

This section presents the properties listed in the `config.properties` file.

- **org.osgi.framework.system.packages**

  To override the packages the framework exports by default from the classpath, this variable must be set.

- **org.osgi.framework.system.packages.extra**

  To append packages to the default set of exported system packages, this value must be set.

- **org.osgi.framework.bootdelegation**

  This property makes specified packages from the classpath available to all bundles and should be avoided. However, if such a configuration is to be made, an example of values listed could be *sun.*,com.sun.*,jdk.**.

- **jamaica-ams.bootdelegation.implicit**

  According to the boolean value of this property, JamaicaAMS tries to infer when to implicitly boot delegate to ease integration with external code. This feature is set to *true* by default.

- **org.osgi.framework.storage**

  This property specifies the location of the bundle cache, which defaults to `jamaica-ams-cache` in the current working directory `$user.dir`. If this value is not absolute, then the `jamaica-ams.cache.rootdir` will control how the absolute location is calculated as in

  org.osgi.framework.storage=jamaica-ams.cache.rootdir/jamaica-ams-cache.

- **jamaica-ams.cache.rootdir**

  This property is used to convert a relative bundle cache location into an absolute one, by specifying the root to prepend to the relative cache path. The default for this property is the current working directory `$user.dir`.

- **org.osgi.framework.storage.clean**

  This property controls whether the bundle cache is flushed the first time the framework is initialized. Possible values are “none” and “onFirstInit”. The default value is “none”.

- **jamaica-ams.cache.locking**

  This boolean property is used to enable/disable bundle cache locking. On JVMs that do not support file channel locking, you may want to disable this feature. The default is enabled.

- **jamaica-ams.cache.filelimit**

  The integer value of this property limits how many open files the bundle cache is allowed to use. The default value is *0*, which is unlimited.

- **jamaica-ams.auto.deploy.action.1=install,start**

- **jamaica-ams.auto.deploy.action.2=install,start**

- **jamaica-ams.auto.deploy.action.3=install,start**

  The properties above determine which actions are performed when processing the auto-deploy directory. It is a comma-delimited list of the values “install”, “start”, “update”, and “uninstall”.

  An undefined or blank value is equivalent to disabling auto-deploy processing. The numerical ending component is the targeted start level. Any number of these properties may be specified for different start levels.

- **jamaica-ams.auto.deploy.dir.3**

  This property specifies the directory to use as the bundle auto-deploy directory; the default is `bundle.n` in the working directory. The numerical ending component is the target start level. Any number of such properties may be specified for different start levels.

- **jamaica-ams.auto.install.3**

  This property is a space-delimited list of bundle URLs to install when the framework starts. The numerical ending component is the target start level. Any number of such properties may be specified for different start levels.

- **jamaica-ams.auto.start.3**

  This property is a space-delimited list of bundle URLs to install and start when the framework starts. The ending numerical component is the target start level. Any number of such properties may be specified for different start levels.

- **jamaica-ams.log.uncaught.exceptions.bundle.threads**

  This property suppresses logging of uncaught exceptions in threads belonging to bundles. The default is *false*.

- **org.osgi.framework.startlevel.beginning=3**

  This property sets the initial start level of the framework upon startup.

- **jamaica-ams.startlevel.bundle=3**

  This property sets the start level of newly installed bundles.

- **jamaica-ams.service.urlhandlers**

  JamaicaAMS installs a stream and content handler factories by default. The value of this property should be actively set to *false*, in order not to install them.

- **jamaica-ams.shutdown.hook**

  By default, JamaicaAMS's launcher registers a shutdown hook to cleanly stop the framework. The value of this property should be actively set to *false*, in order to disable this hook.

8.1.2 System Properties

The following are built-in system properties of the JamaicaAMS, listed in `<path to JamaicaAMS>/setup/conf/system.properties`.

- **java.security.policy=./conf/all.policy**

  This property sets the path to the Java security policy files and should be adapted as needed.

- **jamaica-ams.security.policy=./conf/osgi.all.policy**

  This property sets the path to the JamaicaAMS security policy files and should be adapted as needed.

- **org.osgi.framework.security=osgi**

  This property enables security.

- **org.apache.felix.log.maxSize=100**

  This property sets the maximum size of entries in the log history. A value of *-1* means the log has no maximum size; a value of *0* means that no historical information is maintained.

- **org.apache.felix.log.storeDebug=false**

  This property determines whether or not debug messages will be stored in the history.

- **jamaica-ams.log.level**

  This property is used to set the log levels. The JamaicaAMS logging levels match those specified in the OSGi Log Service (i.e., 1 = error, 2 = warning, 3 = information, and 4 = debug).

8.1.3 Logger Properties

Note that...

The Log Services specification defines the methods and semantics of interfaces which bundle developers can use to log entries and to retrieve log entries. Bundles can use the Log Service to log information for the Operator. Other bundles, oriented toward management of the environment, can use the Log Reader Service to retrieve Log Entry objects that were recorded recently or to receive Log Entry objects as they are logged by other bundles.

Source: The OSGi Alliance - OSGi Compendium

In JamaicaAMS, Log Service is used to expose event messages. There are two JamaicaAMS loggers, with distinguished identifiers:

- **jamaica-ams.log**
- **jamaica-ams.BundleLogger**

They are used for the modules "framework" (i.e. the system bundle) and "bundle" (i.e., the bundles except the system bundle), respectively.

Generally, log levels can be set to control the amount of logging performed, where a higher number results in more logging. A log level of zero turns off the logging functionality completely.

The loggers are implemented using **java.util.logging**. It can be configured using the file specified in **java.util.logging.config.file**.

8.2 Budgets

Note that...

One distinguishing feature of JamaicaAMS in comparison to other OSGi frameworks is that resource budgets may be imposed on the separate bundles.

The possibility to define budgets is a very important aspect, since all bundles are executed within the same process and it becomes necessary to limit the resources used by each one of them. Otherwise, a single misbehaving bundle could impact the performance of the system as a whole.

In order to impose budgets on a bundle, the bundle's JAR file needs to contain an entry named **bundle.properties**. This entry needs to be in Java property file format (see [4]).

Currently JamaicaAMS supports CPU, memory, and thread budgets. Budgeting the CPU time ensures that all bundles get a fair share of execution time. Budgeting the number of threads and the memory usage ensures that no single bundle monopolizes these shared resources. In JamaicaAMS, budgets are implemented through the following properties:

- **budget.cpu** It defines the CPU budget as a percentage of each period of 100 milliseconds. Values for the **budget.cpu** property have the format *number%*, where number is an integer between *0* and *100*.

  If a bundle exceeds its budget in a given period, the priority of all threads started by the bundle is lowered to 0 for the remainder of the period. Priority 0 is a special priority that is lower than all Java priorities. At the beginning of the next period, the original priorities are restored for all threads of the bundle.

  The effect of a priority being dropping to the lowest possible value is that the bundle's threads will only be scheduled to run if no other thread in the system is ready to be executed.

  Example:

  In the JamaicaAMS subdirectory `<example>`, the bundles "primes" and "primes-with-budgets" display a sequence of prime numbers. For the variant "primes-with-budgets" however, it is specified a cpu budget that the bundle is allowed to use. This is declared under `<resources/bundle.properties>`.

  Example:

  ```
  budget.cpu=5%
  ```

  By running both bundles, the user can perceive the impact of imposing such restrictions.

- **budget.memory** JamaicaAMS allocates a fixed-sized Java heap at startup (typically between *64 MB* and *256 MB*). All memory allocations take place in this heap, that is shared by all bundles.

  Memory budgets ensure that no single bundle uses too much of the shared heap. Otherwise, starting additional bundles might not be possible, or else already running bundles may cease to operate properly due to an out-of-memory situation.

  JamaicaAMS permits memory budgets to be defined in bytes. Values for the **budget.memory** property have the format *number*, where number is a non-negative integer denoting the memory budget in bytes.

  If a bundle exceeds its memory budget, any subsequent allocation via "`new`" will result in an `OutOfMemoryError`.

  Example:

```
budget.memory = 5242880
```

This budget permits a bundle to use up to five MB of Java heap.

- **budget.threads** It defines the thread budget as number of concurrently active threads. The format of this property is *bundle.maxNumberOfThreads* = <*n*>.

  A value larger than *0* will enforce the number of active threads to be limited by this value. Any value smaller than *0* is invalid.

  If a bundle reaches its thread budget, any subsequent creation of a new thread will result in an exception. For example:

  ```
  com.aicas.jamaica.lang.ThreadGroupMaxActiveCountExceededError:
  ThreadGroup is limited to 10 threads, contains 10, so add is not
  allowed
  ```

  A bundle may create an arbitrary number of instances of class `java.lang.Thread`, and there is no limit on the number of calls to `start()` on these threads. However, the number of threads that may be alive simultaneously is limited by this property. To be sure that a thread is no longer alive, a call to `Thread.join()` is required. Before a call to `Thread.join()`, the thread may have finished its Java execution, but it may still not have been released back to the framework's thread pool.

8.3 Thread Count

This section accounts for the threads that are created by JamaicaAMS. This information is useful for configuring the overall thread number and for configuring thread numbers for individual bundles.

In JamaicaAMS, the overall thread number can be configured through the environment variable **JAMAICA_AMS_NUM_THREADS**: this sets the maximum allowed threads in the runtime. Bundle thread numbers can be configured through the property **budget.threads**. There are in total 39 threads when JamaicaAMS runs with the default bundles.

Below is a list of JamaicaAMS threads. Note that starter threads are created when JamaicaAMS starts and do not terminate until it shuts down. To complete the information, this section also includes a list of the Jamaica Virtual Machine main threads.

JamaicaAMS Starter Threads

- Main Thread
  To invoke the construction and the initialization of JamaicaAMS and to start the initialized framework.

JamaicaAMS Framework Threads

- FrameworkWiring
  To perform asynchronous package refreshes.
- StartLevel
  To query and modify the start level information for the framework.
- FrameworkDispatchQueue
  To update all listeners (e.g., Framework, bundle, synchronous, and service listeners) associated with a specified bundle, and to dispatch the events of the bundle to the event listeners.
- FrameworkExecutorThread
  To perform special framework operations (e.g. bundle termination) asynchronously in an isolated and safe manner.

Administrative JamaicaVM Threads

- Finalizer
- MemReservation0 (idle time GC)
- Reference Handler
- PosixEventThread (heap)
- SignalPumpThread

JamaicaVM Realtime Threads

- AbstractAsyncEventHandlerThread

8.3.1 Bundle Threads

Those contain all user threads that are created by the bundle. Note that user threads often have a limited lifetime. The configurable thread numbers (**JAMAICA_AMS_NUM_THREADS** and **budget.threads**) impose bounds on the number of threads that exist *simultaneously*.

In addition to the user threads, there is an administrative thread for each bundle, which only exists while the bundle is in active state. This thread is associated to the installed bundle. There is a `BundleThreadGroup` per Bundle for managing its threads. There are in total 28 threads constructed for the default built-in bundles.

JamaicaAMS Bundle Threads

- **com.aicas.jamaica.ams.bundle.osgi-log-writer**
  Uses the **java.util.logging** to output the log entries recorded by the logging service. The **java.util.logging** runs on a seperated thread.
- **com.aicas.jamaica.ams.bundle.policy-file-reader**
  Loads the policy file for the OSGi security layer.
- **com.aicas.jamaica.ams.bundle.configuration-admin**
  Specifies a service, which allows for easy management of configuration data for configurable components. Threads **CM Event Dispatcher** and **CM Configuration Update** are part of this package.
- **org.apache.felix.gogo.shell**
  Provides a simple textual user interface to interact with the command processor. Internally it starts the shell on a separate thread, **Gogo Shell**. The Gogo Shell thread continues to create threads **job controller**, **pipe-gosh –login**, and a thread pool, which incrementally constructs 10 threads. Therefore, there are in total 13 threads constructed directly and indirectly from the **org.apache.felix.gogo.shell** thread.
- **org.apache.felix.gogo.runtime**
  Provides the core command processing functionality.
- **org.apache.felix.gogo.command**
  Provides a set of basic commands.
- **org.apache.felix.configurator**
  Provides an implementation of the OSGi Configurator Service Specification.
- **org.osgi.util.function**
  Provides a set of basic utilities.
- **org.apache.sling.commons.johnzon**
  Powered javax.json library.
- **org.apache.felix.log**
  Provides a set of log utilities. Thread **FelixLogListener** is part of this package.
- **org.apache.felix.converter**
  Provides a set of basic utilities.
- **org.apache.felix.cm.json**
  Provides support for OSGi configurations specified in JSON documents.

8.4 Usage of the Java Native Interface (JNI)

In general developers are discouraged from using the Java Native Interface (JNI). Nevertheless, the usage of JNI allows them to use native code and easily interact with Java objects (e.g. get and set field values, and invoke methods without many constraints).

On the one hand, with JNI it is possible to use the safety behavior of the Java language for the ability to accomplish the tasks listed earlier; on the other hand, JNI provides powerful low-level access to the machine resources (memory, I/O, and so on), but you need to be careful because you are working without the safety net usually provided to Java developers.

JNI's flexibility and power introduce the risk of programming practices that can lead to poor performance, bugs, consume system heap and fail on a heap allocation.

For these reasons, developers must be careful about the code they include in their software and use good practices to safeguard its integrity. Please note the additional information on best practices:

- A common JNI programming error is to call a JNI method and to proceed without checking for exceptions once the call is complete. This can lead to buggy code and crashes. For this reason it is important to return to the Java side in case of a pending exception. The pending exception will then be raised as a regular Java exception.
- The memory allocated on the native side of JNI is not accounted for in the memory limits.
- Many JNI methods have a return value that indicates whether the call succeeded or not. A common error, similar to not checking for exceptions, is to fail to check the return value and for the code to proceed on the assumption that the call was successful.
- Native methods can create global references so that objects are not garbage collected until they are no longer needed. Common errors are forgetting to delete global references that have been created or losing track of them completely. Not freeing global references is an issue not only because they keep the object itself alive but also because they keep all objects that can be reached through the object alive. In some cases this can add up to a significant memory leak.

Note: In its Core document for the Release 8, the OSGi Alliance advises developers about the use of the Java Native Interface.

Considerations Using Native Libraries

There are some restrictions on loading native libraries due to the nature of class loaders. In order to preserve namespace separation in class loaders, only one class loader can load a native library as specified by an absolute path. Loading of a native library file by multiple class loaders (from multiple bundles, for example) will result in a linkage error.

Care should be taken to use multiple libraries with the same file name but in a different directory in the JAR. For example, foo/http.dll and bar/http.dll. The Framework must only use the first library and ignore later defined libraries with the same name. In the example, only foo/http.dll will be visible.

A native library is unloaded only when the class loader that loaded it has been garbage collected. When a bundle is uninstalled or updated, any native libraries loaded by the bundle remain in memory until the bundle's class loader is garbage collected. The garbage collection will not happen until all references to objects in the bundle have been garbage collected, and all bundles importing packages from the updated or uninstalled bundle are refreshed. This implies that native libraries loaded from the system class loader always remain in memory because the system class loader is never garbage collected.

It is not uncommon that native code libraries have dependencies on other native code libraries. This specification does not support these dependencies, it is assumed that native libraries delivered in bundles should not rely on other native libraries.

Source: `https://docs.osgi.org/download/r8/osgi.core-8.0.0.pdf`.

Chapter 9

Information for Specific Targets

Generally all JamaicaVM target systems should be also feasible to run JamaicaAMS, and this chapter aims at documenting information referring to specific platforms.

9.1 Linux

JamaicaAMS can be run on variant Linux distributions for variant target, e.g., Intel X86 architecture (32-bit/64-bit) and ARM architecture (v7 and v8). The following section describes several configuration issues which might need to be taken care of to run the JamaicaAMS.

9.1.1 Shared Libraries

In order to run JamaicaAMS, the following shared libraries should exist in the library search paths.

- `libm.so`
- `libpthread.so`
- `libnsl.so`
- `libdl.so`
- `librt.so`
- `libstdc++.so`

Some Board Support Packages (BSPs) might not have `libstdc++.so` integrated. In this case, the `libstdc++.so` has to be copied to the target file system where the library loader can discover it. For example, if the `libstdc++.so` is copied in a folder `<path to JamaicaAMS>/setup/<target>/lib`, the environment variable `LD_LIBRARY_PATH` should be set with following command:

```
> export LD_LIBRARY_PATH=<path to JamaicaAMS>/setup/<target>/lib:$LD_LIBRARY_PATH
```

9.1.2 Random Number Generator

JamaicaAMS requires a random number generator been initialized on the target system. If the BSP does not initialize the random number generator at boot time, it has to be done prior to starting the JamaicaAMS. For example, on a Raspberry Pi, the application processor contains a Hardware Random Number Generator. The BSP has been configured with the device (`/dev/random` and `/dev/hwrng`) existing in the system. Therefore, no additional steps need to be done. For other distribution or targets, please consult with the document of the specific distributions or the targets.

Bibliography

[1] Apache Felix Gogo Shell. http://felix.apache.org/documentation/subprojects/apache-felix-gogo.html.

[2] Introducing the Bndtools. https://bnd.bndtools.org.

[3] JamaicaVM 8.5 User Manual. https://www.aicas.com/download/manuals/JamaicaVM-8.5-Manual.pdf.

[4] Java Properties File Format. http://www.aicas.com/jamaica/8.1/doc/jamaica_api/java/util/Properties.html#load-java.io.Reader-.

[5] OSGi Core Release 8 Specification. https://docs.osgi.org/download/r8/osgi.core-8.0.0.pdf.

[6] PMD Information. https://pmd.github.io/.

[7] SpotBugs Information. https://spotbugs.github.io/.

[8] The Eclipse Download page. https://www.eclipse.org/downloads/.

[9] The Eclipse project. https://www.eclipse.org/.

[10] The Plugin-in Development Environment (PDE) for Eclipse. https://www.eclipse.org/pde/.