OSGi workshop

Martin Toshev
@martin_fmi

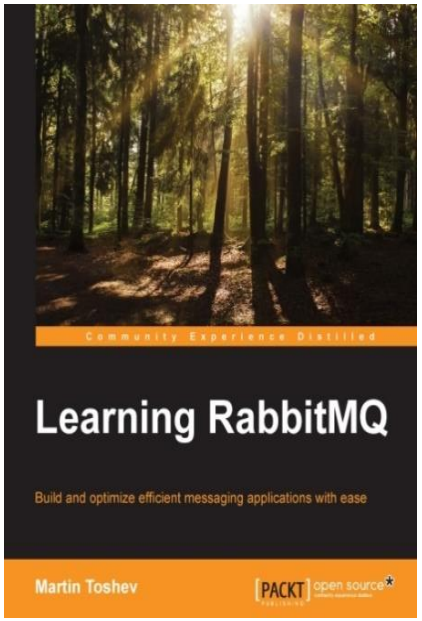
Who am I

Software consultant (CoffeeCupConsulting)

BG JUG board member (http://jug.bg)

(BG JUG is a 2018 Oracle Duke's choice award winner)

OpenJDK and Oracle RDBMS enthusiast



OSGi experience

Eclipse & Lotus notes plugin development

QIVICON IoT Platform development

OSGi-based modular tax calculation platform

Conference sessions:

- **Eclipse plug-in development**
- **Modularity of the Java platform**

Workshop agenda

Part 1 modules:

- **Modularity and OSGi fundamentals**
(exercise: developing and deploying a basic OSGi bundle)

- **Bundle communication**
(exercise: exporting & importing packages,
registering and consuming services, service listeners)

- **Service tracking and declarative services**
(exercise: adding service tracking and declarative services)

Workshop agenda

Part 2 modules:

- **OSGi security**
(exercise: securing an OSGi bundle)

- **Testing and debugging OSGi bundles**
(exercise: unit testing OSGi with PAX Exam, debugging bundles)

- **Building and deploying OSGi bundles**
(exercise: building and deploying OSGi bundles with Maven)

- **OSGi compendum features and runtime internals**
(exercise: implementing web and API bundles, clustering)

Modularity and OSGi fundamentals

- Modularity 101
- Modularity on top of the platform: OSGi
- Exercises

Modularity 101

Modularity 101

- Standard Java libraries are modules - Hibernate, log4j and any library you can basically think of …
- Build systems like Maven provide transparent management of modules

Modularity 101

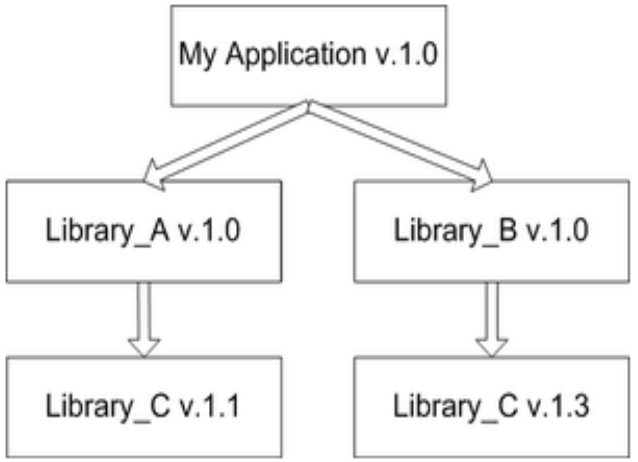
- Benefits of modularization:
  - smaller modules are typically tested easier than a monolithic application
  - allows for easier evolution of the system - modules evolve independently

Modularity 101

- Benefits of modularization:
  - development of the system can be split easier between teams/developers
  - increased maintainability of separate modules

Modularity 101

- The dependency mechanism used by the JDK introduces a number of problems that modular systems aim to solve:
  - The "JAR hell" problem caused by shortcomings of the classloading process



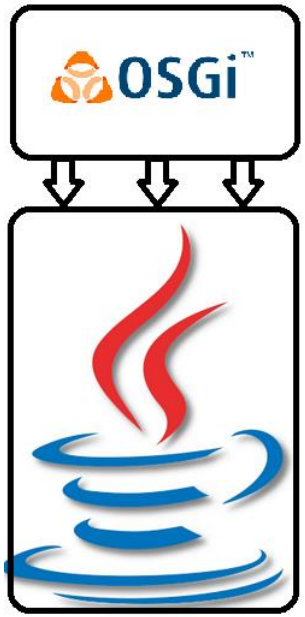
Modularity 101

- The dependency mechanism used by the JDK introduces a number of problems that modular systems aim to solve:
  - The lack of dynamicity in managing dependent modules
  - The lack of loose coupling between modules

Modularity 101

- Module systems aim to solve the mentioned problems and typically provide:
  - module management
  - module deployment
  - versioning
  - dependency management
  - module repositories
  - configuration management

Modularity on top of the platform: OSGi



Modularity on top of the platform: OSGi

- OSGi (Open Service Gateway initiative) provides a specification for module systems implemented in Java
- It is introduced as JSR 8 and JSR 291 to the Java platform

Modularity on top of the platform: OSGi

Q: So what is an OSGi runtime ?

Modularity on top of the platform: OSGi

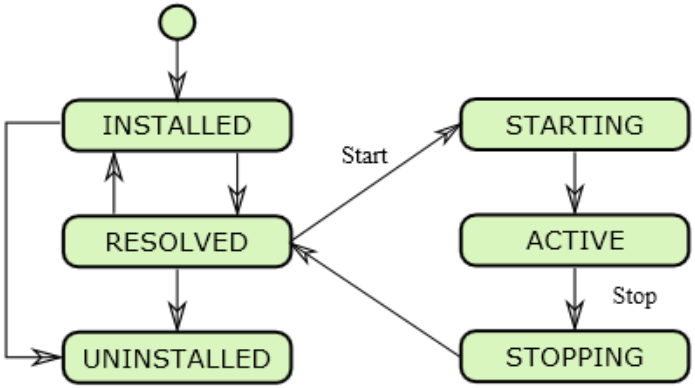
Q: So what is an OSGi runtime ?

A: An OSGi runtime (module system) makes use of the Java classloading mechanism in order to implement a container for modular units (bundles) and is based on the OSGi spec - a series of standards by the OSGi Alliance. Many application servers are implemented using OSGi as a basis - it is also used in systems from a diversity of areas

Modularity on top of the platform: OSGi

Q: So what is an OSGi runtime ?

A: An OSGi bundle is a just a JAR file that contains source code, bundle metadata and resources. A bundle may provide various services and components to the OSGi runtime. An OSGi runtime allows for bundles to be installed, started, stopped, updated and uninstalled without requiring a reboot



Modularity on top of the platform: OSGi

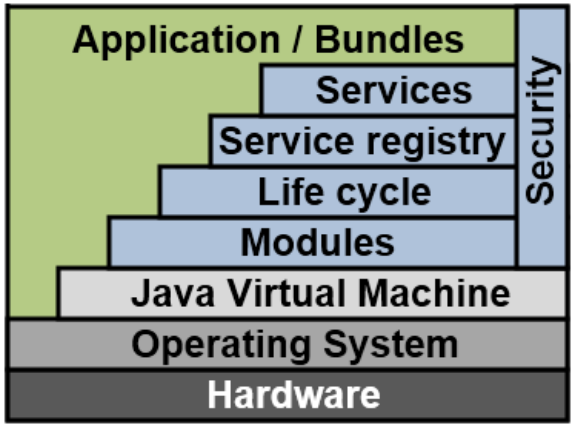
Q: So what is an OSGi runtime ?

A: The OSGi Core spec defines a layered architecture that determines what is supported by the runtime – each layer defines a particular functionality supported by the runtime and the bundles

OSGi logical units:

- bundles
- services
- services registry
- life-cycle
- modules
- security
- execution environment

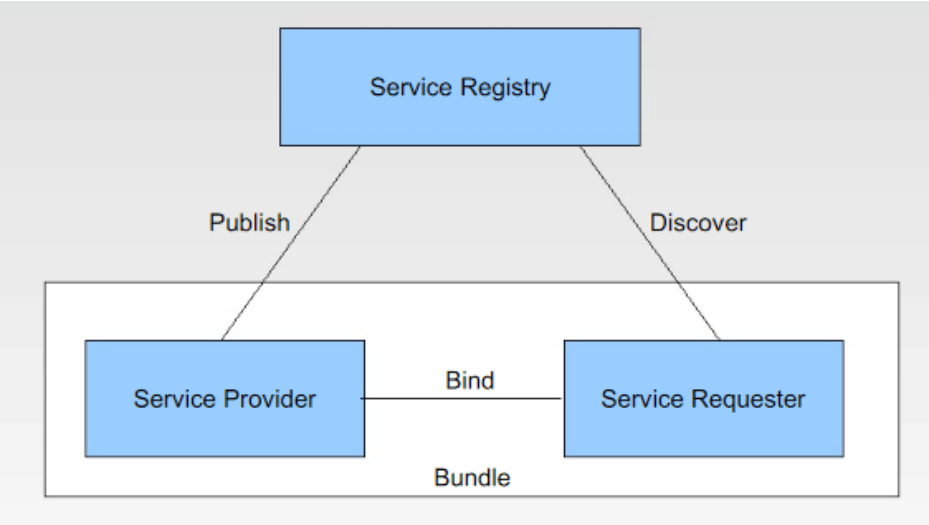
OSGi logical layers:



Modularity on top of the platform: OSGi

Q: So what is an OSGi runtime ?

A: Bundles may export packages for use by other bundles or import packages exported by other bundles - this dependency mechanism is referred to as **wire protocol** and is provided by the Module layer of OSGi. Bundles may publish services to the runtime and use already published services from the runtime – this dependency mechanism is provided by the Service layer of OSGi.



Modularity on top of the platform: OSGi

Q: So what is an OSGi runtime ?

A: The MANIFEST.MF file of the bundle's JAR file describes the metadata of the bundle

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Sample
Bundle-SymbolicName: com.sample
Bundle-Version: 1.0.0.qualifier
Bundle-Activator: sample.Activator
Bundle-Vendor: QIVICON
Require-Bundle: org.eclipse.smarthome.core,
 com.qivicon.extensions
Bundle-RequiredExecutionEnvironment: JavaSE-1.7
Service-Component: OSGI-INF/service.xml
Import-Package: com.qivicon.services.hdm;version="3.0.0"
Export-Package: com.sample.utils
```

Modularity on top of the platform: OSGi

Q: So what is an OSGi runtime ?

A: The runtime may implement extensions based on the OSGi Compendium spec that extends the OSGi Core spec. These could be:

- remote services
- log service
- HTTP service
- device access service
- configuration admin
- metatype service
- preferences service
- user admin
- wire admin
- DMT admin service
- IO connector service
- provisioning service
- UPnP device service
- configuration admin
- declarative services
- event admin service
- deployment admin
- XML parser service
- monitoring service
- others

Modularity on top of the platform: OSGi

Q: What are the most notable OSGi runtimes ?

A: Notable implementations are:

- Equinox (used for Eclipse plug-in development)
- Apache Felix
- Apache Karaf
- Knopflerfish
- Concierge OSGi
- ProSyst (Bosch IoT Gateway Software)

Modularity on top of the platform: OSGi

Q: Who is using OSGi ?

A: OSGi is used in a variety of application domains:

- JavaEE app servers (Glassfish, JBoss, WebSphere, Weblogic)
- web portals (Liferay)
- ESBs (ServiceMix, Fuse ESB)
- home automation/IoT (Eclipse SmartHome, OpenHab)
- plug-in systems (Eclipse IDE, IntelliJ, Confluence, JIRA)
- others

Modularity on top of the platform: OSGi

Q: What about Maven support for OSGi bundles ?

Modularity on top of the platform: OSGi

Q: What about Maven support for OSGi bundles ?

A: Such a support is provided by the Tycho Maven plug-ins that provides support for packaging types, target platform definitions, interoperability with the Maven dependency mechanism and so on … There is also a maven-bundle-plugin which provides ‘bundle’ packaging type for Maven artifacts

Exercise

(developing and deploying a basic OSGi bundle)

Bundle communication

- Module layer (package import/export)
- Service layer (publishing/consuming services)
- Exercises

Module Layer (package import/export)

Package Import and Export

- By default the packages of an OSGi bundle are not visible to the outside world
- This improves module encapsulation and provides an additional layer of security
- The OSGi module layer defines a mechanism to expose and require packages explicitly within a bundle
- In addition every package from the bundle might have a version along with the export of the package

Package Import and Export

- Packages visible to the outside world are defined with the **Export-Package** attribute:

```
Export-Package: org.osgi.workshop.docgen
```

- Packages required from other bundles in the OSGi runtime are specified with the the **Import-Package** attribute:

```
Import-Package: org.osgi.framework
```

Package Import and Export

- Packages can be exported with a version:

```
Export-Package: org.osgi.workshop.docgen;version=1.0.1
```

- Imported packages can be specified with a version or a version range:

```
Import-Package: org.osgi.framework;version=1.8.0
Import-Package: org.osgi.framework;version="(1.7.0,1.8.0]"
```

- In case multiple packages match the version range, the one with newest version is imported

Package Import and Export

- If no OSGi module is providing an imported package specified by the module we develop then our module won't be resolved

- There are however cases when an imported package becomes available at a later point in time (another bundle that provides it is installed)

- In that case we can use the so called dynamically imported packages in the declaration of our module metadata:

```
DynamicImport-Package: com.example.bundle
```

Package Import and Export

- An alternative option is to specify resolution:=optional on the imported package:

```
Import-Package: com.example.bundle;
                resolution:=optional
```

Package Import and Export

- If you need to use a third-party library in your OSGi bundle you can use an “OSGified” version of that library that has the proper MANIFEST.MF bundle metadata for an OSGi bundle

- If there is no such metadata for the library or one cannot be supplied then the library can be embedded within the bundle and specified on its classpath:

```
Bundle-ClassPath: lib/thirdparty.jar
```

```
Bundle-NativeCode: /lib/lib.DLL; osname = Windows10
```

Bundle Import

- All exported packages from a bundle can be imported by using the Require-Bundle metadata attribute:

```
Require-Bundle: com.example.bundle
```

- Usage of Require-Bundle is however discouraged as Import-Package is a more preferable way to request package imports from the runtime

Services Layer
(publishing/consuming services)

OSGi services

- OSGi services are defined by the OSGi service layer
- Services are a more loosely coupled way for bundles to communicate with each other than package imports/exports
- Services are registered in a service registry that is typically implemented in a bundle provided by the OSGi runtime
- An org.osgi.framework.ServiceFramework instance provides access to the OSGi service

OSGi services

- Every OSGi bundle has a BundleContext method associated that is injected in the start()/stop() methods of the bundle activator or can be retrieved from the OSGi framework

- A service can be registered via any of the BundleContext.registerService(…) methods

- A service can be retrieved via the BundleContext.getService(ServiceReference)

OSGi services

- A service is registered by providing the name or class of the service interface
- In addition a service object instance is provided
- The framework validates that the service object instance indeed extends the provided service interface
- Additional properties describing the service are provided as key/value pairs in a map

OSGi services

- It is a good practice that the service interface is in a separate bundle than any concrete service implementations

- In practice that is not the case … In many real world scenarios a service interface has just one implementation and both are provided by the same bundle …

OSGi services

- Service listeners that receive events when a service has been registered, modified or unregistered from the OSGi registry can be defined in a bundle

- This can be done with the BundleContext.addServiceListener(ServiceListener) method

Exercise

(exporting & importing packages,
registering and consuming services,
service listeners)

Service tracking and declarative services

- Service tracking
- Declarative services
- Exercises

Service Tracking

Service Tracker overview

- The OSGi service tracker is a utility that provides an alternative way to keep track of service registration/modification/unregistration from the service registry

- It is defined in the Tracker specification from the OSGi core specification along with another utility for tracking of bundles (bundle tracker)

- The implementation of the service tracker is provided by the **org.osgi.util.tracker.ServiceTracker** class

Service Tracker overview

- ServiceTracker achieves the same purpose service listener as service listeners …

- However with service listeners we cannot track the existing state of a service when the listener is registered and that is solved by using a service tracker instead

Declarative Services

Declarative services overview

- Declarative services are a mechanism that can be used to create OSGi components in a more expressive manner
- Declarative services are defined as part of the OSGi Compendum Specification
- You can think of declarative services as the OSGi equivalent of Spring or CDI beans

Declarative services overview

- Declarative services are specified with the **Service-Component** manifest attribute that defines one or more XML files (comma-separated) within the bundle that specify components and their dependencies to other components or services:

```
Service-Component: OSGI-INF/services.xml
```

- The components are managed thought an OSGi runtime component called the Service Component Registry (SCR) that keeps track of changes to components

Declarative services annotations

- Declarative services also provide annotations that are used to specify the components and their dependencies and are build-time only (XML definitions are generated out of them):

  @Component – defines and OSGi component (that could expose a service)

  @Reference – Injects an OSGi service

  @Activate – annotates a method called when starting the bundle

  @Deactivate – annotates a method called when stopping the bundle

Declarative services mechanism

- Declarative services eliminate the need of using a service activator to register or unregister services
- Also they eliminate the need of using a service listener or a service tracker
- Declarative services are a dependency injection mechanism in OSGi that keeps track of changes in services provided by the service registry

Declarative service injection types

- Declarative services provide three types of injection targets:
  - Constructor parameters
  - Methods
  - Fields

Exercise

(exercise: adding service tracking and declarative services)

Building and deploying OSGi bundles

- Building OSGi bundles using Maven
- Deploying OSGi bundles
- Exercises

Building OSGi bundles with Maven

Maven plug-ins

- To build OSGi bundles from Maven you need proper Maven plug-ins that provide OSGi metadata and dependency resolution capabilities:
  - maven-bundle-plugin
  - bnd-maven-plugin
  - karaf-maven-plugin
  - tycho-maven-plugin

Bundle repositories

- In order to resolve bundle dependencies the various plug-ins support different types of bundle repositories such as:
  - Local file system
  - OBR (OSGi bundle repository)
  - Maven Nexus
  - JFrog artifactory
  - p2 / target platform
  - Karaf Cave

Maven bundle plug-in

- A wrapper around the BND tool
- Provides a **bundle** packaging type
- Provides the possibility to add OSGi metadata without changing the existing packaging type
- Provides the possibility to deploy the bundle to an OBR

Maven bundle plug-in

- Provides the possibility to embed resources and dependencies to the bundle’s classpath
- Provides the possibility to amend an existing bundle manifest file
- Provides the possibility to validate bundle’s metadata entries

Maven bundle plug-in

```
<plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <extensions>true</extensions>
        <configuration>
        <instructions>
                <Export-Package>.../Export-Package>
                <Import-Package>...</Import-Package>
                <Bundle-Activator>...</Bundle-Activator>
                ...
        </instructions>
        </configuration>
</plugin>
```

BND Maven plug-in

- Provides possibility to generate bundle manifest file
- Provides possibility to generate declarative service descriptor files
- BND instructions are specified and referenced directly from a **bnd** file instead of the pom file (as with maven-bundle-plugin)
- It is also possible (but not recommended) to include bnd instructions directly in the pom file

BND Maven plug-in

```
<plugin>
        <groupId>biz.aQute.bnd</groupId>
        <artifactId>bnd-maven-plugin</artifactId>
        <configuration>
                <bndfile>bnd/docgen.bnd</bndfile>
        </configuration>
</plugin>
```

BND Maven plug-in

```
<plugin>
        <groupId>biz.aQute.bnd</groupId>
        <artifactId>bnd-maven-plugin</artifactId>
        <configuration>
                <bnd><![CDATA[
                -exportcontents:org.osgi.workshop.docgen.api
                ]]></bnd>
        </configuration>
</plugin>
```

BND Maven plug-in

- Provides possibility to generate bundle manifest file
- Provides possibility to generate declarative service descriptor files
- BND instructions are specified and referenced directly from a **bnd** file instead of the pom file (as with maven-bundle-plugin)
- It is also possible (but not recommended) to include bnd instructions directly in the pom file

Karaf Maven plug-in

- The karaf-maven-plugin is specific to the Apache Karaf framework (provides the **feature**, **kar** and **karaf-assembly** packaging types)

- Provides the possibility to:

  - validate Karaf feature descriptor files
  - add Karaf features (with descriptors and bundles) to a local Maven repository
  - create a KAR (Karaf ARChive) file for a feature
  - create Karaf distribution archives
  - deploy a Karaf application to a remote Karaf instance

Karaf Maven plug-in

```
<packaging>feature</packaging>
<plugin>
        <groupId>org.apache.karaf.tooling</groupId>
        <artifactId>karaf-maven-plugin</artifactId>
        <version>4.2.1</version>
        <extensions>true</extensions>
        <configuration>
                <enableGeneration>true</enableGeneration>
        </configuration>
         <executions>
        <execution>
                <id>generate-features-file</id>
                <phase>generate-resources</phase>
                <goals>
                        <goal>features-generate-descriptor</goal>
                </goals> </execution> </executions> </plugin>
```

By default traverses transitive Maven dependencies and looks for OSGi bundles to include to the generated Karaf feature descriptor

Karaf Maven plug-in

```
<packaging>kar</packaging>
<build>
        <plugins>
                <plugin>
                        <groupId>org.apache.karaf.tooling</groupId>
                        <artifactId>karaf-maven-plugin</artifactId>
                        <version>4.0.0</version>
                        <extensions>true</extensions>
                </plugin>
        </plugins>
</build>
```

Tycho Maven plug-in

- The tycho-maven-plugin is used specifically to build Eclipse plug-ins (provides the **eclipse-plugin, eclipse-test-plugin, eclipse-feature, eclipse-repository** and **eclipse-target-definition** packaging types)

- Dependencies might be resolved from:

  - remote p2 repository
  - Eclipse target definition
  - Local mirror of a remote Eclipse update site

Deploying OSGi bundles

Deployment packages

- For ease of deployment different OSGi implementations provide way for define OSGi applications as a set of bundles deployed and managed together.

- Equinox bundles and in particular Eclipse PDE define features as a grouping of bundles and a way to package multiple features as Eclipse update sites

- Apache Karaf defines a different format for features used to represent a set of OSGi bundles and a way to package multiple features in a Karaf archive (KAR)

Repository deployment

- Depending on the type of repository you have different options to automate repository deployment of OSGi bundles (or applications):
  - Maven repository: the deploy:deploy-file goal of the Maven deploy plug-in

```
Mvn deploy:deploy-file -Dfile=bundle.jar.jar -DpomFile=pom.xml
-DrepositoryId=internal -Durl=http://server/nexus
```

  - OBR repository: bundle:deploy-file file goal of the Maven bundle plug-in
  - P2 repository: tycho-p2-extras:publish-features-and-bundles goal of the Tycho p2 Extras plug-in

Manual container deployment

- Easiest way is to do a manual install using the **install** command from the OSGi shell on a single bundle or use specific commands to install the particular feature or application as provided by the OSGi framework

- You can use different location such as the file system, URL from a Nexus repository etc.

- Not convenient for automatic deployment and management

Automated container deployment

- Typical options to achieve automatic deployment to a running OSGi container are:
  - Hot deployment
  - Installation from a local OSGi shell connected to a remote OSGi runtime

Automated container deployment

- Some containers like Karaf provide a “hot deployment” options whereby you put artifacts in a folder and they are automatically deployed

- For Karaf you can put bundles, features and KAR files in the **deploy** folder

- You can use utilities such as **SCP** to copy artifacts to the hot deploy directory of a remote OSGi runtime

Automated container deployment

- The Karaf client can connects (using SSH) to a remote Karaf instance (Karaf runs an SSHd server) and execute a command remotely

- As bundles and Karaf features can be stored in a remote repository (i.e. Nexus) we can deploy a bundle/feature on the remote instance using the client:

```
bin/client -a 8101 -h hostname -u karaf -p karaf features:install docgen
```

- You can also use an SSH client (such as OpenSSH) to connect to the remote Karaf instance

Updating package versions

- OSGi by default does not change the package version upon which a target bundle depends if a newer version of that packages is installed that satisfies the wire protocol

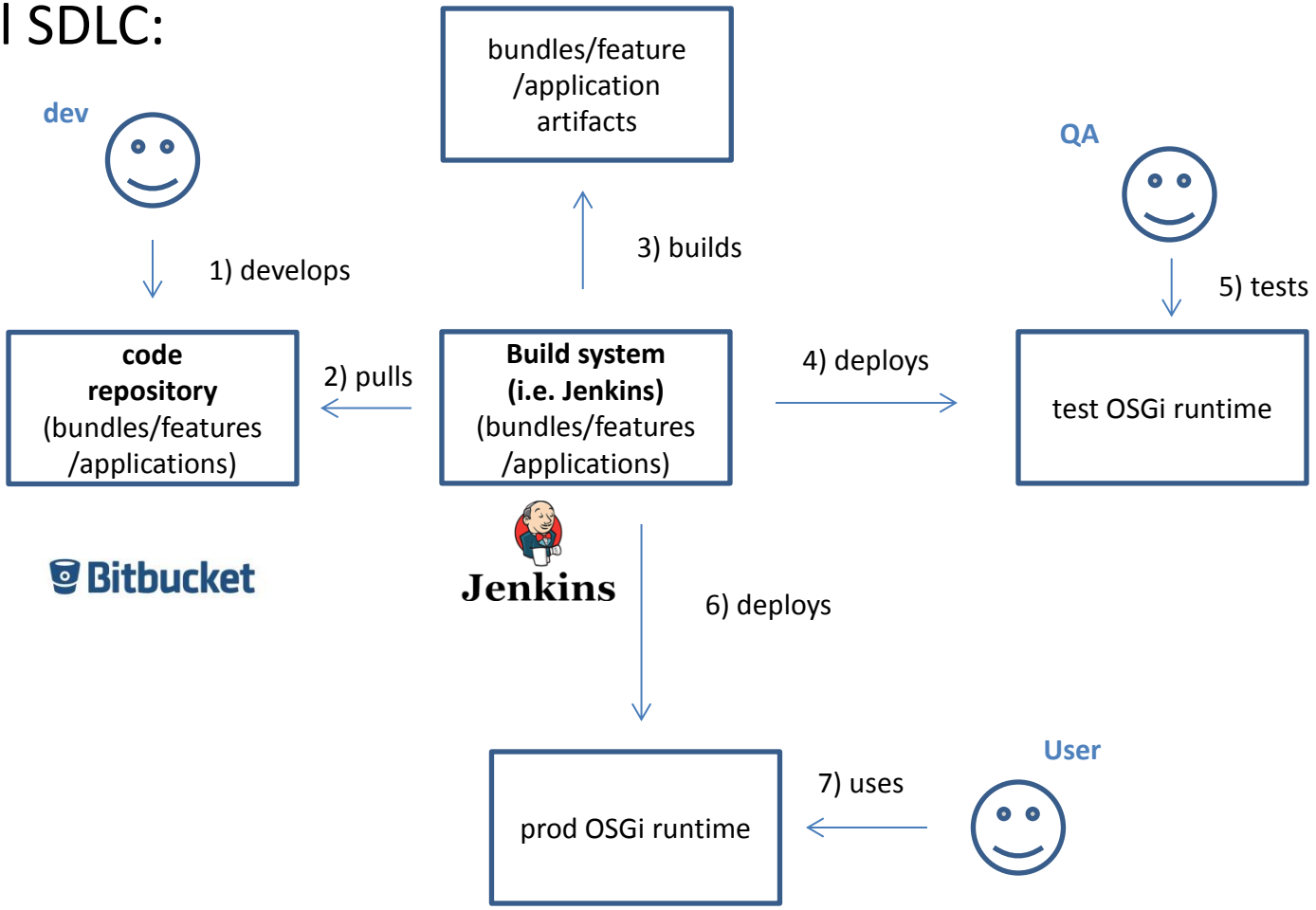
- To “re-wire” the dependencies of installed bundles you need to explicitly call the refresh command to refresh package dependencies

```
refresh
```

- If you want to create “snapshots” based on a particular set of bundles along with their versions it is good to have separate feature files for each of those snapshots

SDLC

Typical SDLC:



Exercise

(exercise: building and deploying OSGi bundles with Maven)

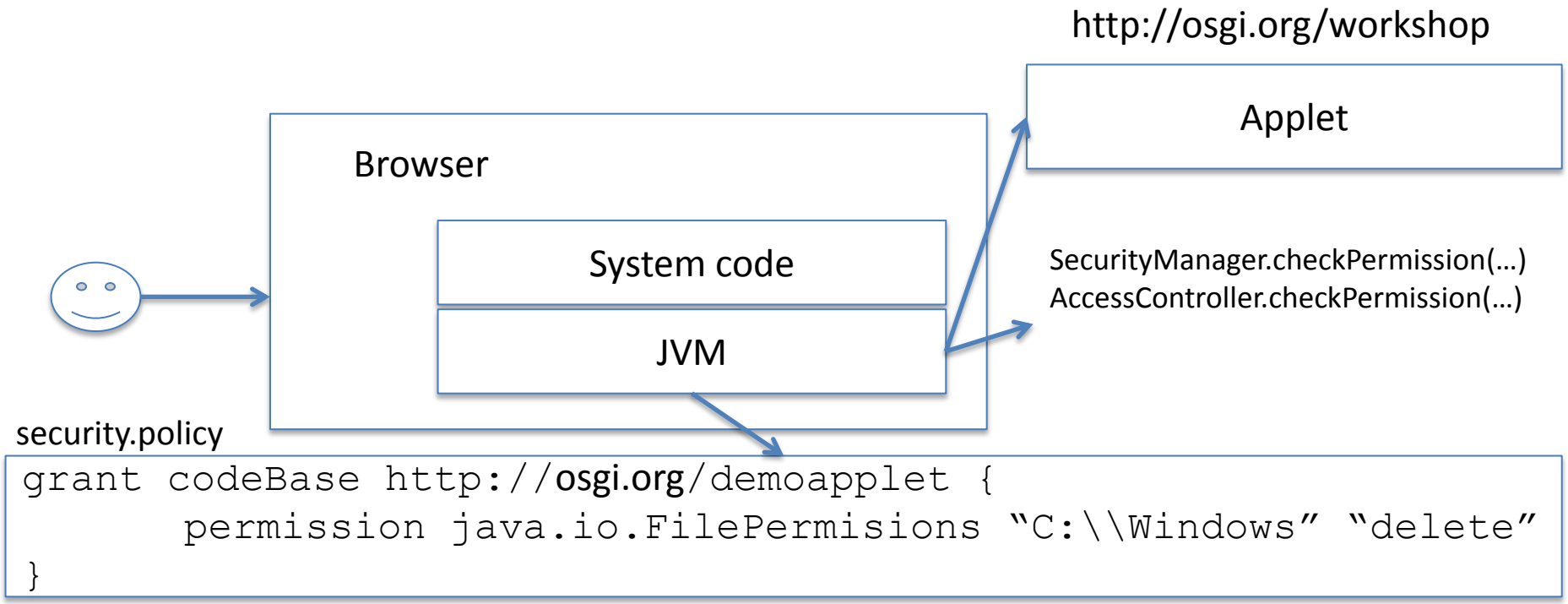
OSGi security

- JDK security primer
- OSGi security overview
- Karaf security
- Exercises

JDK security primer

JDK security sandbox model

- The JDK security sandbox model is introduced in JDK 1.2:



Protection Domain

- The notion of **protection domain** introduced – determined by the security policy
- Two types of protection domains – system and application
- The protection domain is set during classloading and contains the code source and the list of permissions for the class

```
applet.getClass().getProtectionDomain();
```

Permission properties

- One permission can imply another permission:

```
java.io.FilePermissions “C:\\Windows” “delete”
implies
java.io.FilePermissions “C:\\Windows\\system32” “delete”
```

- One code source can imply another code source:

```
codeBase http://dev.bg/
implies
codeBase http://dev.bg/demoapplet
```

Protection Domain

- Since an execution thread may pass through classes loaded by different classloaders (and hence – have different protection domains) the following rule of thumb applies:

  The permission set of an execution thread is considered to be the intersection of the permissions of all protection domains traversed by the execution thread

Protection Domain

- The following is a typical flow for permission checking:
  - upon system startup a security policy is set and a security manager is installed

```
Policy.setPolicy(…)
System.setSecurityManager(…)
```

  - during classloading (e.g. of a remote applet) bytecode verification is done and the protection domain is set for the current classloader (along with the code source and the set of permissions)

Protection Domain

- The following is a typical flow for permission checking:
  - when system code is invoked from the remote code the SecurityManager (or AccessController) is used to check against the intersection of protection domains based on the chain of threads and their call stacks:

```
SocketPermission permission = new
        SocketPermission("dev.bg:8000-9000","connect,accept");
SecurityManager sm = System.getSecurityManager();
if (sm != null) sm.checkPermission(permission);
```

```
SocketPermission permission = new
        SocketPermission("dev.bg:8000-9000", "connect,accept");
AccessController.checkPermission(permission)
```

Protection Domain

- The following is a typical flow for permission checking:
  - application code can also do permission checking with all permissions of the calling domain or a particular JAAS subject:

```
AccessController.doPrivileged(…)
Subject.doAs(…)
Subject.doAsPrivileged(…)
```

OSGi security overview

OSGi security

- Security of the OSGi framework is defined by a separate section of the OSGi Core specification called the security layer
- As OSGi is a managed environment it must provide a mechanism to sandbox OSGi bundles in a secure manner
- The security layer is specified as optional in the OSGi Core specification but most frameworks implement it

OSGi security overview

- The OSGi security model is an extension of the Java security sandbox model
- The OSGi spec provides a set of custom permissions such as **PackagePermission, ServicePermission, BundlePermission** or **AdminPermission**
- **PackagePermission** specifies whether a bundle is allowed to export or import a package
- **ServicePermission** specifies whether a bundle is allowed to retrieved or register an OSGI service

OSGi security model

- The **PermissionAdmin** and **ConditionalPermissionAdmin** classes provide additional permission management on top of SecurityManager

- The **PermissionAdmin** and **ConditionalPermissionAdmin** are further defined in separate sections of the OSGi Core specification

- To support the extended security model each bundle is set a specific protection domain (for Apache Felix bundles that is an instance of **org.apache.felix.framework.BundleProtectionDomain**)

JDK security model extensions

- The permission admin classes provide dynamic permission management
- They also provide the ability to also DENY certain permissions for bundles (rather than only ALLOW then as provided by default in the JDK security model)
- Local permissions can be specified for each bundle in **OSGI-INF/permissions.perm** and are useful for bundle security auditing

PermissionAdmin

- Is the pre-OSGi 4.0 way to manage permissions
- Instance can be retrieved as an OSGi service
- Can be used to manage permissions per bundle
- Allows a management agent (such as the OSGi console) to manage bundle permissions

PermissionAdmin

- Java permissions managed by the permission admin are wrapped in **org.osgi.service.permissionadmin.PermissionInfo** instances

- Provides a way to specify default set of permissions for all bundles

ConditionalPermissionAdmin

- Supersedes the PermissionAdmin
- Instance can also be retrieved from the OSGi service registry
- Provides the capability to specify additional permission conditions used during permission checking

ConditionalPermissionAdmin

- Updating bundle permissions (example):

```
ConditionalPermissionAdmin conPermAdmin =
        (ConditionalPermissionAdmin) context.getService(serviceRef);
ConditionalPermissionUpdate update =
        conPermAdmin.newConditionalPermissionUpdate();
List<ConditionalPermissionInfo> perms =
        update.getConditionalPermissionInfos();
List<ConditionalPermissionInfo> oldPermissions =
        new ArrayList<ConditionalPermissionInfo>(perms);
update.getConditionalPermissionInfos().clear();
perms.add(...)
operation update.commit();
```

ConditionalPermissionAdmin

- Adding new bundle permissions with condition (example):

```
ConditionInfo condition = new
ConditionInfo(BundleLocationCondition.class.getName(), new String[]
{someBundle.getLocation()});
ConditionInfo conditions = new ConditionInfo[] {condition};
PermissionInfo perm1 = new
PermissionInfo(FilePermission.class.getName(), "/some/path", "read");
PermissionInfo perm2 = new
PermissionInfo(FilePermission.class.getName(), "/other/path",
"read,write");
PermissionInfo[] permissions = new PermissionInfo[] {perm1, perm2};
admin.newConditionalPermissionInfo(description, conditions,
permissions, ConditionalPermissionInfo.ALLOW);
```

OSGi permission checking flow

- The OSGi security model typically has the following flow during a security check:
  - If the requested permission is not listed as a local permission (if a permission.perm file is present) then a security exception is thrown
  - If the previous check passes the global permissions set from the ConditionalPermissionAdmin are checked to see if the requested permissions is denied

Signed OSGi bundles

- The OSGi framework supports permission checking based on the location of the OSGi bundle and also the signers of the bundle's JAR file (if the JAR file is signed)

- The JDK provides a utility called **jarsigner** used to sign and verify JAR files

Felix security

- Security Layer implementation provided by the **org.apache.felix.framework.security** bundle
- The following properties must be specified:
  - org.osgi.framework.security="osgi"
  - java.security.policy=all.policy
  - org.osgi.framework.trust.repositories=<list of keystores>
- The all.policy file should grant all permission

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

Felix security

- You can use the framework's default security manager:

```
java -Djava.security.policy=all.policy
-Dorg.osgi.framework.security="osgi" -jar bin/felix.jar
```

```
java -Djava.security.policy=all.policy
-Djava.security.manager  -jar bin/felix.jar
```

- You can also specify a different security manager using the fully-qualified Java class name as part of the **java.security.manager** or **org.osgi.framework.security** properties

Felix security

- Additional properties for the certificate keystore (example):

```
felix.keystore=file:docgen.jks
felix.keystore.type=jks
felix.keystore.pass=pass
osgi.signedcontent.support=all
```

Equinox security

- For Equinox the setup is similar and you also need to grant AllPermission in the security.policy file

```
java -Djava.security.manager="" -Djava.security.policy=all.policy
-Dosgi.framework.keystore=file:keystore.ks
-Dosgi.signedcontent.support=true
-jar org.eclipse.equinox.launcher.jar -noExit
```

Karaf security

- Karaf adds yet another layer of security on top of the OSGi runtime
- It provides authentication and authorization through the JAAS framework

Exercise

(exercise: Securing an OSGi bundle)

Testing and debugging OSGi bundles

- Testing OSGi bundles
- Debugging OSGi bundles
- Exercises

Testing OSGi bundles

Testing OSGi bundles

- The approaches for testing of OSGi bundles follow standard practices of testing code in a managed environment:
  - **unit testing**: frameworks such as JUnit and TestNG can be used to test the code of an OSGi bundles with possibly the use of mocking frameworks in order to mock/stub utilities provided by other bundles or the OSGi framework itself
  - **Integration testing**: this are frameworks that provide the possibility to write container tests, i.e. running a test over an OSGi bundle that is first deployed along with its dependencies in an OSGi runtime

Pax Exam

- Pax Exam is one of the most popular and developed libraries for OSGi integration testing
- It provides the ability to execute a target OSGi container, deploy the bundles of your application (system under test) into the container and execute JUnit tests from within the container
- It also provides the ability to run a Karaf container and execute commands directly

Pax Exam

- Provides a **PaxExam** JUnit runner
- Supports major OSGi frameworks such as Equinox, Felix, Knopflerfish and Karaf
- Supports **Native** and **Forked** containers
- Native containers run in memory
- Forked containers run in a separate JVM

Pax Exam

```
@RunWith(PaxExam.class)
@ExamReactorStrategy(PerMethod.class)
public class ExampleTests {

   @Inject
   private SomeService someService;

   @Configuration
   public Option[] config() {

      return options(
         mavenBundle(“<groupId>", “<artifactId>", “<version>"),
         bundle(“<bundle_jar>"),
         junitBundles()
         );
   }

   @Test
   public void getHelloService() { … }
}
```

Debugging OSGi bundles

Debugging OSGi bundles

- A variety of options include that allow you to diagnose issues with OSGi bundles through the OSGi shell or a web console bundle provided by the OSGi container

- You can also debug directly OSGi bundles from an IDE of your choice

Debugging OSGi bundles

- For Equinox you can use directly the PDE debug support to create debug configurations for OSGi bundles/Eclipse plug-ins

- For Felix/Karaf or another runtime you need to attach remotely to the OSGi container in order to be able to debug bundles

Diagnostic commands

- Before actually debugging the bundle code you may also want to diagnose what is the state of the bundle in the OSGi runtime

- At the very basic this can be achieved with the **list** and **bundle** commands that allow you to see the state of bundles and what metadata do they have in the runtime

Karaf diagnostic commands

- The **dev:dump-create** command creates a snapshot of the state of the Karaf environment including information about bundles, features, heap, threads etc.

- The **bundle:diag** provides information on why bundles cannot be resolved

- The **system:framework** displays the current OSGi framework in use (by default that is Apache Felix)

Karaf diagnostic commands

- The **shell:stack-traces-print** prints full stack trace information in case a command throws an exception

- The **bundle:tree-show** command shows the depedency tree for a bundle based on the wiring information (the Karaf equivalent of **mvn dependency:tree**), for Equinox you can use the **Dependency Hierarchy** tab from the PDE

- The **bundle:watch** command enables watching bundles for updates – if a bundle is updated in the target Maven repository Karaf updates it in the OSGi runtime

```
bundle:watch *
```

Karaf diagnostic commands

- Additional bundle diagnostic commands:
  - **bundle:capabilities** displays OSGi capabilities of a bundle
  - **bundle:classes** displays a list of classes/resources contained in the bundle
  - **bundle:diag** displays diagnostic information for a bundle
  - **bundle:find-class** locates a specified class in any deployed bundle
  - **bundle:headers** prints bundle metadata

Karaf diagnostic commands

- You can also view diagnostic information for a running Karaf instance using one of the JMX MBeans provided by the framework

- You programmatically connect to the Karaf runtime using provided Java clients for the SSHd and JMX servers running on the Karaf instance

Exercise

(exercise: unit testing OSGi with PAX Exam, debugging bundles)

OSGi compendum and runtime internals

- OSGi compendum
- OSGi runtime internals
- Exercises

OSGi Compendum

OSGi Compendum Overview

- The OSGi compendum specification fulfils the OSGi core specification by providing a number of specification for the implementation of standard OSGi services such as logging, HTTP, device API-related services and so on.

- Each of those services are defined in separate section of the compendum specification which is typically implemented in a separate bundle starting with **org.osgi.service** (i.e. for the logging service that is the **org.osgi.service.log** bundle)

OSGi Runtime internals

Classloading

- Every OSGi bundle is loaded by a separate classloader which is defined by the OSGi core specification
- The classloader can load class from the bundle classpath, OSGi runtime classpath or additional bundle resources (like fragments)
- Fragments (or fragment bundles) are OSGi bundles that are used to enhance the functionality of existing OSGi bundles

OSGi clustering

- Комуникация между два бъндъла от различни контейнери (remote OSGi, others?)

- Достъп до OSGi услуги в контернер отвън, например при стартиране на UI тестове (remote services ?)

// **Karaf Cellar**

Exercise

(web and API bundles, clustering)

Q&A

Thank you