1. Concepts

Virgo Web Server is a Java application server composed of a collection of modules which supports applications which are also composed of a collection of modules. These may be traditional Java web applications packaged as Web ARchive (`.war`) files as well as other modular applications. Modules may be shared between applications and multiple versions of modules can co-exist.

This chapter introduces concepts necessary for developing Web Server applications. These concepts will become clearer as the GreenPages application is explored in later chapters.

1.1 OSGi concepts

Modules in Virgo are represented using a standard Java module system known as ***OSGi***. Modules consist of programs and resources organised by Java package together with metadata which declares imported and exported packages. A module *exports* a package to make the corresponding programs and resources available for use by other modules. A module *imports* a package to use the corresponding programs and resources of another module.

Representing a program as a collection of modules makes it easier for the programmer to manage it and modify it and for teams of programmers to divide responsibilities between themselves. A module is similar to a Java class in this respect. Design principles similar to those for organising data and programs into classes can be applied to organising applications into modules.

An industry consortium known as the ***OSGi*** *Alliance* (see the appendix Projects) develops **OSGi** specifications, reference implementations, and compliance tests. Virgo Web Server is built on the Equinox **OSGi** framework which is also the reference implementation for the **OSGi** framework specification.

Bundles

Modules in **OSGi** are known as *bundles*. Each bundle is stored in a file which conforms to the JAR file format and can contain Java classes, a **manifest** (in `META-INF/MANIFEST.MF`), and further resource files.

The **OSGi** framework enables bundles to be installed and run.

OSGi identifies bundles “by name” or “by identifier” (id).

The *symbolic name* and ***version*** of a bundle is an attribute of the bundle itself and uniquely identifies that bundle (by name) in an **OSGi** framework. A bundle usually declares its *symbolic name* and ***version*** in its **manifest** (a file called `MANIFEST.MF`) like this:

```
Bundle-SymbolicName: org.foo.bundle
Bundle-Version: 1.2.3.BUILD-2009-06-04
```

Additionally, the **OSGi** framework assigns a distinct number, known as a *bundle id*, to each bundle as it is installed. Bundles may be referred to “by identifier” using this number. The **OSGi** framework itself resides in a bundle with bundle id `0`.

The dependencies between bundles are expressed statically in terms of packages and dynamically in terms of services. A package is familiar to Java programmers. For example, a Java program may depend on a class `org.foo.X`, from package `org.foo`, and a bundle containing that program would either need to contain `org.foo.X` or depend on the package `org.foo`. Package dependencies are specified in the bundle **manifest**, for example:

```
Import-Package: org.foo
```

A bundle which provides a package for use by other bundles *must* export the package in its **manifest**. For example:

```
Export-Package: org.foo
```

The **OSGi** framework ensures that a given bundle’s package dependencies can be *satisfied* before the bundle runs. This process is known as *resolution*.

After a bundle is resolved, its classes and resources are available for loading. In **OSGi**, bundles and their packages do not appear on the application classpath. Instead, each bundle has a class loader which loads its own classes and loads classes belonging to each of its imported packages by deferring to the bundle class loader that exports the package.

Life cycle

The **OSGi** framework manages the *life cycle* of each bundle. A bundle is first of all *install*ed and will be in the INSTALLED state. If a request is made to *start* the bundle, the **OSGi** framework *resolve*s the bundle and, if resolution was successful, will subsequently move the bundle to the ACTIVE state. If a request is made to *stop* the bundle, the **OSGi** framework will move the bundle back to the RESOLVED state. A request may then be made to *uninstall* the bundle.

While the bundle is INSTALLED, ACTIVE or RESOLVED, it may be *updated* to pick up some changes. These changes are not detected by bundles which were depending on the bundle before it was updated. A “refresh packages” operation may be performed to ripple the changes out to those bundles. (See Services concepts.)

The life cycle of a bundle can be summarised by a state transition diagram. This diagram shows some more of the intermediate states of a bundle not described in the overview above:

Figure 1.1. Bundle life cycle

Bundle life cycle

Services

Bundles may publish Java objects, known as *services*, to a registry managed by the **OSGi** framework. Other bundles running in the same **OSGi** framework can then find and use those services. Services are typically instances of some shared Java interface. A bundle which provides a service need not then export the package containing the *implementation* class of the service.

For example, a bundle could export a package containing the interface `org.bar.SomeInterface`, thus:

```
Export-Package: org.bar
```

…implement the interface with a class `SomeImpl`:

```
package org.bar.impl;

class SomeImpl implements SomeInterface {
        …
}
```

…create an instance of `SomeImpl` and then publish this instance (as an instance of

the interface `SomeInterface`).

An **OSGi** framework publishes a number of standard services. For example, the *Package Admin* service provides the “refresh packages” life cycle operation mentioned above.

OSGi provides an *API* which can be used to publish and find services, but it is much simpler to use Spring DM to accomplish this. (See Spring DM concepts.)

Versioning

OSGi allows different versions of bundles, packages, and several other entities, to co-exist in the same framework and provides some mechanisms for managing these versions.

Version numbers

An **OSGi *version*** *number* consists of up to three numeric components, or exactly three numeric components followed by a string component. These components are separated by a period (“.”) and are called the *major*, *minor*, *micro*, and *qualifier* components, respectively.

For example, the **version** `2.4.1.ga` has major component `2`, minor component `4`, micro component `1`, and a qualifier component `ga`. (There are restrictions on the characters that can appear in a qualifier. For example: letters, digits, underscores and hyphens are allowed; periods and commas are not.)

Trailing components may be omitted along with their period (.). So, for example, the **version** numbers `2`, `2.0`, and `2.0.0` all denote the same **version**. This example demonstrates that `0` is assumed if a numeric component is omitted, and the empty string is assumed for an omitted qualifier.

Version ranges

Dependencies on bundles and packages have an associated ***version*** *range* which is specified using an interval notation: a square bracket “`[`” or “`]`” denotes an *inclusive* end of the range and a round bracket “`(`” or “`)`” denotes an *exclusive* end of the range. Where one end of the range is to be included and the other excluded, it is permitted to pair a round bracket with a square bracket. The examples below make this clear.

If a single **version** number is used where a **version** *range* is required this does *not* indicate a single **version**, but the range *starting* from that **version** and including all higher versions.

There are three common cases:

- A “strict” **version** range, such as `[1.2,1.2]`, which denotes that **version** and only that **version**.
- A “half-open” range, such as `[1.2,2)`, which has an inclusive lower limit and an exclusive upper limit, denoting **version** `1.2.0` and any **version** after this, up to, *but not including*, **version** `2.0.0`.
- An “unbounded” **version** range, such as `1.2`, which denotes **version** `1.2` and *all* later versions.

Versioning policies

A *versioning policy* is a way of using **version** numbers to indicate compatible and incompatible changes. **OSGi** does not mandate a particular versioning policy. Instead, a specific versioning policy may be implemented using **version ranges**.

Strict and half-open **version ranges** are most useful in representing versioning policies. Unbounded **version ranges** can lead to problems as they (unrealistically) assume that compatibility will be preserved indefinitely.

For example, a conservative versioning policy might assume that any change, other than in the qualifier component of a **version**, implies an incompatible change to the object. Such a policy would employ **version ranges** such as `[1.2.1.beta,1.2.2)` which accept any **version** from `1.2.1.beta` (inclusive) up to but not including `1.2.2` (exclusive).

Alternatively, a relaxed versioning policy might assume that only changes in the major component of a **version** denote an incompatible change. Such a policy would employ **version ranges** such as `[1.2,2)` to capture this.

Bundle version

Each bundle has a **version**. The bundle’s **version** may be specified in the **manifest** using a `Bundle-Version` header:

```
Bundle-Version: 1.4.3.BUILD-20090302
```

If not specified the bundle **version** is assumed to be `0`.

Package version

Each exported package has a **version**. The exported package’s **version** may be

specified on the Export-Package **manifest** header. For example

```
Export-Package: org.foo;version="2.9",org.bar;version="1"
```

exports two packages: `org.foo`, at **version** `2.9.0` and `org.bar`, at **version** `1.0.0`.

If the **version** attribute is omitted the **version** is assumed to be `0`.

Each package *import* has a **version** *range*. The package import **version** range may be specified on the `Import-Package` **manifest** header. If interval notation is used, the **version** range must be enclosed in double quotes, for example:

```
Import-Package: org.foo;version="[2,3)",org.bar;version="[1,1]"
```

seeks to import a package `org.foo` in the range `[2.0.0,3.0.0)` and a package `org.bar` with the (exact) **version** `1.0.0`.

If a **version** range is not specified on an import, the range `0` is assumed, meaning that any **version** of this package would satisfy the import.

Bundle manifest version

Bundle manifests have a **version** which is `1` by default, indicating **OSGi** Release 3 semantics. Web Server is based on **OSGi** Release 4 and therefore expects bundle manifests to be at **version** `2`, indicating **OSGi** Release 4 semantics. (See the appendix Projects.) The bundle manifest's **version** should be specified on the Bundle-ManifestVersion **manifest** header, exactly as follows:

```
Bundle-ManifestVersion: 2
```

Manifest version

Manifests themselves also have a **version** which *must* be specified as `1.0`. This is not an **OSGi** definition but part of the JAR file specification (http://java.sun.com/javase/6/docs/technotes/guides/jar/jar.html).

```
Manifest-Version: 1.0
```
