Activate your FREE membership today | Log-in

- Java
- SOA
- EBIZQ
- Test/QA
- .NET
- AJAX
- TSSJS

- Today On TSS
- Discussions
- Topics
- White Papers
- Multimedia
- RSS

- Design/Architecture
    - SOA and service integration
    - REST Web services
    - ESB products
    - Software development
    - Application Integration middleware
    - Application security
    - Design for test
    - Data-oriented design
    - Java architecture for cloud
- EJB
    - EJB specification
    - EJB containers
    - EJB troubleshooting
    - EJB programming
    - Java application servers
    - EJB products
- Web Services
    - Web services architecture
    - WS* specifications
- Web Applications
    - Web app frameworks
    - Spring framework
    - Hibernate framework
    - Java servlets
    - Java server pages
    - Ajax Web development
    - Rich Internet applications
    - Java server faces
    - Java Web portals
    - Other UI
- Development
    - Java programming language
    - Software programming languages
    - JVM languages
    - DSLs
    - Java application deployment
- Performance/Scalability
    - Software performance tools

    - Application performance measurement
    - Cloud/grid/memory systems
    - JVM
    - Application scalability
- Tools
    - Eclipse development and tools
    - Open source tools
    - IDEs
    - Java EE
    - Java software testing
    - Development hardware

SEARCH this site and the web Search

Site Index

- Home
- >Topics
- >Application and Web services development tools
- >Eclipse development and tools
- >*Eclipse, Equinox, and OSGi*

Print
Email This

## Java Development News:

-

# Eclipse, Equinox, and OSGi

By Jeff McAffer and Simon Kaegi

01 Jan 2007 | TheServerSide.com

- ContentSyndication

- Digg This
- Stumble
- Delicious
- Google Fusion

Eclipse has been enormously popular as a tooling platform. With the use of Eclipse as a Rich Client Platform (RCP), Eclipse made a step towards being a runtime platform. Now, with the emergence of Eclipse on the server, Eclipse clearly has leapt into the runtime world. So what makes Eclipse capable of adapting to these different environments – what makes Eclipse tick?

At its core, Eclipse has a modular Java runtime called Equinox ( http://eclipse.org/equinox). Equinox in turn is an implementation of the OSGi Alliance’s Service Platform R4 specification ( http://osgi.org). In fact, it is the reference implementation for the framework portion of that specification. At the heart of the OSGi specification is the notion of *bundles*.

Bundles are used to capture modularity for Java. A bundle is just a standard Java JAR whose manifest contains extra markup identifying the bundle and laying out its dependencies. As such, each bundle is fully self-describing. Below is an example.

```
Bundle-SymbolicName: org.eclipse.equinox.registry
Bundle-Version: 3.2.100.v20060918
Bundle-Name: Eclipse Extension Registry
Bundle-Vendor: Eclipse.org

Bundle-ClassPath: .

Bundle-Activator: org.eclipse.core.internal.registry.osgi.Activator

Export-Package: org.eclipse.equinox.registry

Import-Package: javax.xml.parsers,
org.xml.sax,
org.osgi.framework;version=1.3
Require-Bundle: org.eclipse.equinox.common;bundle-version="[3.2.0,4.0.0)"
Bundle-RequiredExecutionEnvironment: CDC-1.0/Foundation-1.0,J2SE-1.3
```

The first section identifies the bundle giving it a symbolic name, version and various human-readable descriptive values. The symbolic name and version go together to uniquely identify the bundle.

The classpath entry details where, inside the bundle itself, the bundle's code can be found. Specifying '.' is the default and indicates that the bundle's code is at the root of the bundle JAR itself.

The OSGi architecture is highly dynamic. Bundles can come and go at any time in a session. The bundle activator allows bundles to hook the start() and stop() lifecycle events and thus perform initialization and cleanup as needed.

Bundles that provide function to the rest of the system must expose the API for that function by exporting the API package. Exporting a package allows other bundles to reference the classes in the package. Notice that bundles can hide packages simply by not exporting them.

Finally, bundles can express their dependence on code supplied by other bundles. This is done either indirectly using Import-Package statements or directly using the Require-Bundle header. Importing packages allows bundles to be bound (or *wired* as we say) to arbitrary providers of the needed packages. Conversely, directly requiring bundles allows for implementation-specific dependencies. In both cases the prerequisite can be qualified by version ranges and other markup to refine the dependency. Note also that bundles can specify dependencies on the underlying JRE used to run them. The bundle in this example can run on J2ME Foundation 1.0 or higher or J2SE 1.3 or higher.

Given a set of bundles, Equinox *resolves* the dependencies and wires together the bundles. Bundles missing prerequisites are not resolved and are not eligible to be run. In the end each bundle has a dynamically computed and unique classpath that tells it exactly where to get its prerequisite code. Typically the standard Java application classpath is ignored and everything is computed from the expressed dependencies. Gone are the days of fighting with the global classpath.

## How do I program and run an Equinox based system?

Developing bundles with Eclipse is the same as developing an Eclipse plug-in. In fact, there is no difference between bundles and plug-ins. While running the Eclipse SDK, simply use the **New > Project > Plug-in Project** wizard, enter a project name and select "OSGi framework" as the intended target platform (at the bottom of the wizard). Click **Next** until you get to the last page. Here you can choose from a set of templates. For your first bundle, choose the **Hello OSGi Bundle** template. Click **Finish** and you have a new bundle project.

To run the bundle, right click on the bundle project and choose **Run As… > Equinox**. You should see a console appear with an "osgi>" prompt and the message "Hello World!!". That message was printed by your new bundle. At the osgi> prompt you can type various commands. For example, you can stop your bundle by typing "stop <your bundle symbolic name>". Note that by default the project name is used as the bundle's symbolic name. You should see the "Goodbye World!!" message. You can of course restart the bundle using "start <bundle name>". Go take a look at the code the wizard generated and you will see an Activator class with start() and stop() methods that print the corresponding messages. There are many more details and capabilities but this should get you running.

## What did I just do?

An Equinox-based system consists of; Equinox itself, a set of bundles, and a *configuration*. The configuration is a list of bundles to run. This list is drawn from the pool of bundles available on the user's machine. In the example above, the tooling created a configuration that included your bundle and all the bundles the system knew about and launched it. Note that in the simple workflow used here there are many extra bundles added to the configuration. For this example, you really only need your bundle and the org.eclipse.osgi bundle in the configuration.

When Equinox is launched, it installs all of the bundles listed in the configuration, resolves them and starts any bundles listed as needed to be started. Since your bundle was installed and resolved and is listed (by default) as needing to be started, it was started and the Activator.start() method called and message printed.

## Equinox and OSGi in the larger context

Both Equinox and OSGi are seeing a surge in interest from a wide range of communities from embedded, the traditional domain of OSGi, to desktop tools and applications to mobile devices and servers. The server side work is of particular interest.

The Equinox project has a server side effort that seeks to integrate the dynamic module capabilities of OSGi into standard application server scenarios (see http://eclipse.org/equinox/server). There are two main approaches; embedding Equinox in the servlet container or embedding the servlet container in Equinox. To a large extent the choice made here does not impact your server application or functionality. The choice is more a function of the infrastructure needs of your environment. For example, are there other standard web applications running, are you using clustering or other infrastructure, …

The following three diagrams illustrate the traditional
web application server setup and contrasts it with the Equinox in Container approach and the Container in Equinox approach. Equinox is embedded in an existing servlet container as shown in Figure 2 by installing a very thin web application. This application contains a Bridge Servlet that launches an Equinox instance inside the application and then forwards all requests to the Equinox-based application (servlets, JSPs, …) via a very lightweight HTTP service.

In Figure 3, the approach is turned around and Equinox runs an application container (e.g., embedded Jetty) and provides some glue code that wires Equinox-based applications into the application container.

**Figure 1: Traditional web application server structure**

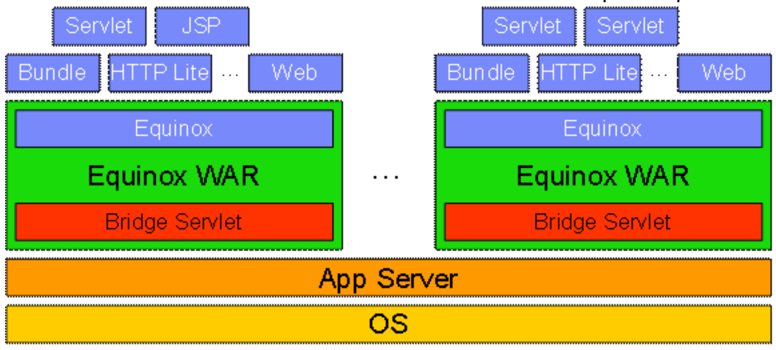

**Figure 2: Embedding Equinox in an existing application container**

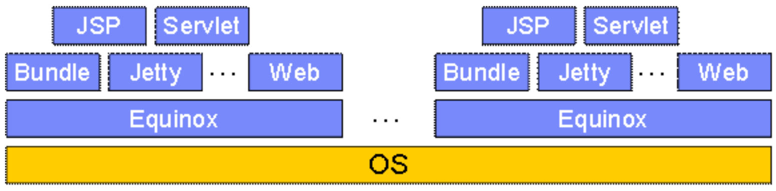

**Figure 3: Embedding an application container in Equinox**

In both Equinox-based scenarios, writing your application is the same. You create a bundle project using the Eclipse plug-in development environment (PDE) and add the static content, servlets and JSPs as needed. Then add traditional Eclipse extension markup to surface your content in the user's URL space. The snippet below shows an example.

```
<plugin> <extension point="org.eclipse.equinox.http.registry.resources"> <resource alias="/files" base-name="/web_files"/> </extensio
```

The first extension places the content of the "web_files" directory of the bundle at /files in the user's URL space (e.g., http://localhost/files/...). The second extension places MyServlet at /test in the URL space (e.g., http://localhost/test).

The easiest way of running you application is to launch directly from Eclipse using the Container in Equinox approach. This is the same process you used with the example bundle above and allows you to run without setting up an application server or deploying the code. As a result it is much faster and easier to test and debug. For complete instructions, see the Server-side QuickStart Guide at http://eclipse.org/equinox/documents/http_quickstart.php.

Since your function is defined in bundles and running in Equinox, everything is dynamic and extensible. You can add, remove and update dynamically without taking down the server, the URL space is updated as you modify the function available and your business logic can be reused on the server, the desktop and in mobile clients.

## What's next for Equinox and OSGi?

Equinox is a mature codebase with millions of users – after all, every Eclipse user is running Equinox. Nonetheless the Equinox team is continually refining the implementation and adding support for new use cases.

OSGi on the server and in the enterprise is becoming increasingly interesting. The next version of IBM's flagship WebSphere Application Server is actually based on Equinox under the covers. The OSGi Alliance is responding to the increased interest by creating an Enterprise Expert Group (EEP) to look at issues such as distributed computing, multi-language support and enterprise wide provisioning.

The general Java community has recognized that it too can benefit from the modularity that OSGi brings. While JSR 277 seeks to add a level of modularity in the Java 7 timeframe, JSRs 232 and 291 provide OSGi-based dynamic Java modularity to Java today. These two JSRs enable OSGi-based modularity at all levels of Java from J2ME CDC/Foundation 1.0 to J2SE 1.3 and up.

The explosion in interest in Equinox and OSGi should not have surprised us but it did. Many people in the Java community are struggling with system complexity and management issues. With Eclipse, Equinox and OSGi you too can enjoy the benefits of modular, componentized Java wherever you compute.

## Biographies

Jeff McAffer leads the Eclipse RCP and Runtime teams and is one of the Eclipse Platform's original architects and committers. Prior to his work at IBM's Ottawa Software Lab, he was a developer at Object Technology International focusing on areas such as distributed/parallel OO computing, expert systems, and meta-level architectures. Jeff holds a Ph.D. from the University of Tokyo.

Simon Kaegi works at Cognos as a developer looking at the run-time provisioning of enterprise software. A committer on the Eclipse Equinox project his current focus is on enabling the many use-cases for OSGi in server-side environments.

**Related Topics:** Eclipse development and tools, VIEW ALL TAGS

- Digg This
- Stumble
- Delicious
- Google Fusion