DZone Refcardz

Brought to you by... eclipseCON 2009

CONTENTS INCLUDE:

Getting Started with Equinox & OSGi

By Jeff McAffer

WHAT IS EQUINOX

Equinox is a highly modular, dynamic Java runtime environment based on the OSGi framework specifications. It is small, performant and highly customizable. Equinox forms the basis of all Eclipse systems from embedded airline check-in kiosks and ski lift gates to rich client applications to IDEs to high-performance application servers such as WebSphere and the Spring dm server.

This reference card gives you a quick tour of the technology, how it works and how to use it. We touch on modularity basics, key metadata markup and some best-practices for creating modules. We then look at runtime elements of Equinox and OSGi – lifecycle, classloading, key APIs and strategies for inter-bundle collaboration (e.g., services and extensions).

GETTING STARTED

As part of the Eclipse ecosystem, the Equinox project also produces a number of downloads. Of course, much of Equinox is included in the regular Eclipse SDK and RCP downloads, so if you have Eclipse, you can start right away. That is what we will do here.

Hot Tip To get all the Equinox bundles, go to the Equinox download site – http://download.eclipse.org/equinox and choose a build. Choose a "Release" or "Stable" build for best results.

Hot Tip Equinox is 99% pure Java and runs on JREs as low as J2ME Foundation 1.1. So it runs on just about anything you have that runs Java. The Equinox launchers (e.g., eclipse.exe) depend on OS and chip architecture so look for platform-specific downloads.

Ok, fire up Eclipse and choose a new, empty workspace location and let's create your first bundle.

DEVELOPING YOUR FIRST BUNDLE

OSGi defines modules as *bundles*. We'll go into details in a minute but first, let's create and run one. Start by creating a new *plug-in project* for the bundle. **Select File > New > Project...** From the resulting dialog, select **Plug-in Project** and click

Developing your first bundle, continued

Next. For the project name, choose one you like (see the tip on naming). For the remainder of the settings, match them to the wizard shown in Figure 1 and click **Next**. On the next page ensure that the **Generate an activator** option in the **Plug-in options** section is checked.

Note **Bundle vs Plug-in.** Bundles and plug-ins are the same thing. *Bundle* is the traditional OSGi term whereas *plug-in* is the original Eclipse term. In Eclipse 3.0 when the Equinox project started and OSGi was adopted, these terms became synonyms.

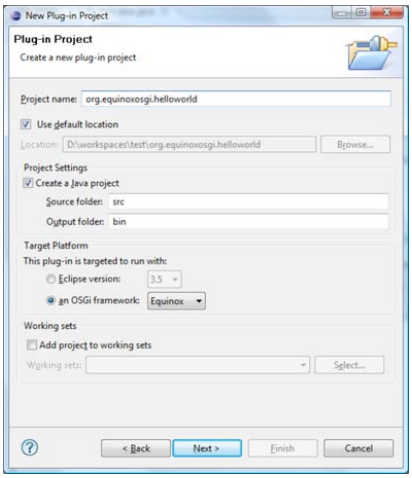


Figure 1

Developing your first bundle, continued

Hot Tip With Equinox it is most common to use the reverse domain name convention (i.e., Java package naming) for bundle names. Bundles are likely to end up grouped together so they need to have unique names. Since every bundle is developed in a separate project, it is convenient to match a project's name with that of the bundle it contains. In the wizard screenshot we used org.equinoxosgi.helloworld. Of course, you should ensure that you own the rights to the related domain (e.g., equinoxosgi.org in this case).

Now that your bundle project is created the bundle's manifest editor will be opened. On the **Overview** tab, click the **Activator** link and check out the Activator class that was created. It should look like this.

```
public class Activator implements BundleActivator {
   public void start(BundleContext context) throws
    Exception {
   }
   public void stop(BundleContext context) throws
    Exception {
   }
```

The Activator is the entry point for your bundle's code—sort of like the standard main() method but specific to a bundle. Change the start() and stop() methods to print a message (e.g., System.out.println("Hello/Goodbye World");)

Hot Tip To add these lines select in the body of a method and type 'sysout' then Ctrl-Space. The Java editor will auto-complete that to System.out.println(); and position the cursor inside the parentheses.

LAUNCHING AN EQUINOX SYSTEM

Now that you have a bundle, lets run it. Open the launch configuration dialog using **Run > Run Configurations...** menu entry. Double click on **OSGi Framework**. On the **Bundles** tab, uncheck the **Target Platform** box in the list of bundles and then click **Add Required Bundles**. You should now have two bundles selected, yours and org.eclipse.osgi. Click **Run**. The OSGi console will appear and your println message should appear.

```
osgi> Hello, World
```

To get a sense of what's happening and the kind of dynamic behavior that is inherent in Equinox, type ss in the console. This shows a "short status" of the system. Each bundle is listed along with its numeric id and current state. Notice that your bundle is ACTIVE. That means its start() method has been called.

```
osgi> ss
Framework is launched.
id  State   Bundle
0   ACTIVE  org.eclipse.osgi_3.5.0.v20081027-1700
1   ACTIVE  org.equinoxosgi.helloworld_1.0.0
```

Stop your bundle by typing **stop 1**. The bundle stopped and the message from stop() is printed. You can restart the bundle using **start 1**. A bundle can be started and stopped as many times as

Launching an Equinox system, continued

you like. Each time it gets the proper lifecycle events.

```
osgi> stop 1
Goodbye, World
```

Hot Tip The console is started by adding the -console command line argument when starting Equinox. The console has many useful commands. For example, you can use the "diag" command to show the missing prerequisites for bundles that are not resolved.

BUNDLES

Having created and run a bundle, let's take a look inside and see what's going on. A bundle is basically a JAR file or directory with some extra headers in the MANIFEST.MF. Looking at a typical bundle from an Eclipse install (Figure 2) you can see that there is a description of the bundle in the manifest, various class files in a standard configuration and additional support files such as legal information, translations and extension contributions.

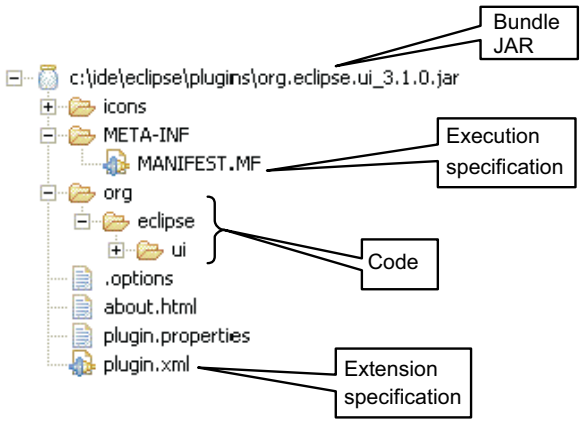


Figure 2

The manifest for a bundle specifies identity, lifecycle and dependency information.

```
Bundle-RequiredExecutionEnvironment: J2SE-1.5
Bundle-SymbolicName: org.equinoxosgi.helloworld
Bundle-Version: 1.0.0
Bundle-Name: Hello World
Bundle-ManifestVersion: 2
Bundle-ClassPath: .
Bundle-Activator: org.equinoxosgi.helloworld.Activator
Export-Package: org.equinoxosgi.helloworld
Import-Package: org.osgi.framework; version="1.3.0"
```

Important headers

Header	Description/Use
Bundle-SymbolicName	([a..zA..Z] | [0..9] | '_' | '-' | '.')* sequence that distinguishes this bundle from other bundles. Typically Java package naming conventions are used.
Bundle-Version	Four part numeric version number where the fourth segment is alphanumeric. The combination of Bundle-SymbolicName and Bundle-Version uniquely identify a set of bundle content.
Bundle-Name	Human-readable name for this bundle.
Bundle-ManifestVersion	What version of OSGi markup is being used. Typical value is 2.

Bundles, continued

Important headers

Header	Description/Use
Bundle-ClassPath	Comma-separated list of JAR entries (directories or JAR files) in the bundle in which to find classes and resources. '.' (dot) is the default and signifies the bundle's root directory (i.e., the bundle itself.)
Bundle-RequiredExecutionEnvironment	Comma-separated list of execution environments in which this bundle can run. For example, CDC-1.0/Foundation-1.0,J2SE-1.3.
Bundle-Activator	The class used to manage the lifecycle (e.g., start and stop) of this bundle.
Export-Package	Comma-separated list of Java packages made available to others by this bundle. Each package may be individually version numbered.
Import-Package	Comma-separated list of Java packages this bundle requires. Each package can be qualified with a version range.
Require-Bundle	Comma-separated list of bundles that this bundle requires. Each bundle can be qualified with a version range.
Bundle-NativeCode	Description of the native code libraries contained in this bundle.
Bundle-ActivationPolicy	"lazy" to indicate that this bundle should be activated when its code is first referenced.

DEPENDENCIES

In OSGi, all bundles explicitly declare the packages they expose to others and the packages they require from others – their dependencies. This yields two main benefits:

- Creating valid configurations is easier. For example, there must be an export for every import or the importing bundle does not resolve.
- A resolved bundle dependency graph tells the system exactly where to look for any given package and greatly improves classloading performance.

Export-Package

A bundle must export every package that it wants other bundles to be able to use. If a package is not exported, the types simply cannot be referenced from outside. This is a key benefit as it forces bundle developers to define their API. Exported packages can, and should, be qualified with a version number that changes whenever the relevant aspects of its API change. Again, this allows others to specify their dependencies accurately and makes the API contract clear.

Dependencies

There are two mechanisms for specifying dependencies in OSGi—Import-Package and Require-Bundle. As the names imply one specifies a dependency on a particular package, the other on a whole bundle. Both can be qualified with a version range and an optional flag indicating that the prerequisite is not absolutely required.

Import-Package

Advantages	Disadvantages
• Loose coupling – implementation independence • Arbitrary attributes allow sophisticated export matching • No issues with package splitting or shadowing – whole package	• More metadata to be created and maintained – each imported package must be declared • Only useful for code (and resource) dependencies • Cannot be used for packages split over bundles

Dependencies, continued

Require-Bundle

Advantages	Disadvantages
• Can be used for non-code dependencies: e.g. Help doc contributions • Convenient for depending on all exports from a bundle • Joins packages split over bundles • Useful when refactoring bundle code or introducing OSGi	• Tight coupling – can be brittle since it requires the presence of a specific bundle • Split packages – Completeness, ordering, performance • Allows one bundle to shadow/override packages from another • Can result in unexpected signature changes

Hot Tip Historically, Eclipse has used Require-Bundle to specify prerequisites, as that was the mechanism first put into place. Since the introduction of Equinox, however, Import-Package has been the recommended way of specifying dependencies.

Hot Tip Eclipse includes comprehensive tooling for Equinox and OSGi. The Plug-in Development Environment (PDE) includes tools for defining, navigating and launching bundles. For example, the manifest editor we saw earlier is part of PDE. It includes a Dependencies tab that supports the analysis of the code in a bundle and the automatic addition of dependencies found.

PROGRAMMING MODEL

The OSGi specification identifies a number of roles and objects that help define and manage bundles at runtime.

Bundle = Identity to others
Other bundles can ask the system for a Bundle object, query its state (e.g., started or stopped), look up files using getEntries(), and control it using start() and stop(). Developers do not implement Bundles—the OSGi framework supplies and manages Bundle objects for you.

Do not confuse Bundle with the old Plugin class. Plugin was an amalgam of several OSGi concepts and is largely obsolete.

You can access the complete set of installed Bundles using various methods on BundleContext.

BundleContext = Identity to the system
At various points in time, bundles need to ask the system to do something for them, for example, install another bundle or register a service. Typically, the system needs to know the identity of the requesting bundle, for example, to confirm permissions or attribute services. The BundleContext fills this role.

BundleContexts are created and managed by the system as an opaque token. You simply pass it back or ask it questions when needed. This is much like ServletContext and other container architectures.

BundleContexts are given to bundles when they are started, that is, when the BundleActivator method start(BundleContext) is called. This is the sole means of discovering the context. If the bundle code needs the context, its activator must cache the value.

Programming model, continued

BundleActivator = Lifecycle handler

Some bundles need to initialize data structures or register listeners when they are started. Similarly, they need to clean up when they are stopped. Implementing a BundleActivator allows you to hook these start and stop events and do the required work.

COLLABORATION

The goal of modular systems is loose coupling. However, even in the most loosely coupled systems, modules need to interact and collaborate. In Equinox you have two major mechanisms to support this collaboration: services and extensions.

Services

Services are the traditional OSGi collaboration mechanism. When active, bundles can add services to the service registry. Other active bundles can then discover those services and invoke them. Service providers don't know about their users and the users don't know about the providers – the coupling is done dynamically.

Services are knit together using one of two patterns: Registration and the Whiteboard. In the Registration pattern some component provides a service, for example, the UI Shell in Figure 3. Other components discover the shell and register with the service to participate. In this example, the screens discover the shell and explicitly add themselves to the UI.

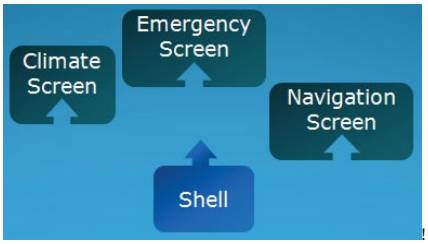


Figure 3

In the Whiteboard approach, the roles are reversed – the various participants register as services and one or more coordinators discover all services and call them. A sort of "don't call us, we'll call you" approach. In Figure 4 we see the screens registering services and the shell consuming these services to render the UI.

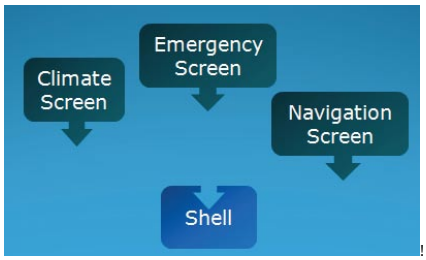


Figure 4

	Pro	Con
Registration	simple, pure POJOs, same programming model everywhere	requires code to run to register
Whiteboard	can be declarative, enables lazy class loading	very difficult to do with pure POJOs

Services, continued

Regardless of which pattern you use, it is best to use the Declarative Services mechanism. This infrastructure allows you to declare in a file, typically component.xml, which services you need and which you provide. The runtime then coordinates services and manages the service lifecycle. This takes the place of complicated and error-prone coding patterns.

For example, the markup for a screen in the registration pattern example is very straight forward. Here the Emergency screen bundle declares that it references an ICrustShell service and that the Component class should receive the service when discovered.

org.equinoxosgi.toast.swt.emergency/component.xml

```
<component name="org.equinoxosgi.toast.swt.emergency">
   <implementation class="org.equinoxosgi.toast.swt.
emergency.internal.bundle.Component"/>
   <reference
      name="shell"
      interface="org.equinoxosgi.crust.shell.ICrustShell"/>
</component>
```

Correspondingly, the shell declares that it provides the ICrustShell service using its Component class.

org.equinoxosgi.crust.shell/component.xml

```
<component
    name="org.equinoxosgi.crust.shell"
    immediate="true">
   <implementation
      class="org.equinoxosgi.crust.Component"/>
   <service>
      <provide interface="org.equinoxosgi.crust.
       ICrustShell"/>
   </service>
</component>
```

Services are used modestly in the Eclipse community but should likely be used more. As of the Galileo release (June 2009) PDE includes tooling for the declarative services markup files.

Extensions

Extensions and extension points are the traditional Eclipse collaboration mechanism. The *extension registry* is a declarative means for one bundle to hook into another in a well-defined way. For example, the UI bundle might expose a menu *extension point* to allow the addition of menu entries. Bundles then contribute extensions detailing their menu entries. The UI bundle then presents the menus using the information given declaratively. When the menu is selected, the class contributed in the extension is instantiated and run. Figure 5 captures this example behavior between the UI and some bundle called Hyperbola.

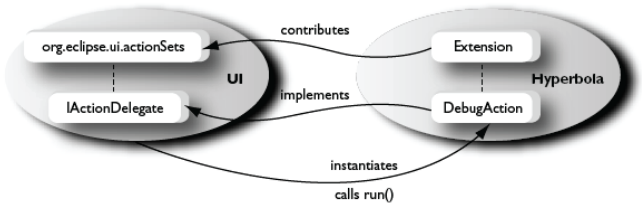


Figure 5

Extension Registry	Manages the declarative relationships between bundles
Extension Point	Bundles open themselves for configuration/extension
Extension	Bundle extends another by contributing an extension

Extensions, continued

Extensions and extension points are defined declaratively in the plugin.xml file in a bundle. An extension point is simply an id, a human readable name and a schema that defines the form of any contributed extensions.

org.eclipse.equionx.http.registry/plugin.xml

```
<plugin>
   <extension-point
    id="servlets"
    name="%servletsName"
    schema="schema/servlets.exsd"/>
</plugin>
```

An extension identifies the extension point it is contributing to and all the information mandated by the schema. In the example here the servlet extension point requires an alias and the name of a class that implements HttpServlet.

org.equinoxosgi.helloworld/plugin.xml

```
<plugin>
   <extension
    point="org.eclipse.equinox.http.registry.servlets">
      <servlet
       alias="/hello"
       class="org.equinoxosgi.helloworld.Servlet">
      </servlet>
   </extension>
</plugin>
```

Eclipse contains literally hundreds of extension points and thousands of extensions and PDE include sophisticated tooling for defining and maintaining extensions and extension points.

Hot Tip

Services and extensions are complementary technologies. The key differences are in the lifecycle and the scope of the collaboration. Extensions come and go as bundles are RESOLVED. Generally that is once when a bundle is installed. Services on the other hand come and go during each run of the system.

The service registry is a single global table where any bundle can discover and use any service. The extension registry uses a more tightly coupled model where extensions are contributed directly to those who will consume them. Both characteristics are useful or problematic in different scenarios.

Extensions	• Tightly coupled model – extensions bound to specific extension points • Contribute code and/or structured data • Lazy loading of extension class • Highly scalable • Life cycle scoped to RESOLVED state of bundle
Services	• Global public context • Loosely coupled model – Any bundle can bind to a service • Services are code-based • Service class eagerly loaded • Life cycle scoped to started ACTIVE of bundle

KEY EQUINOX EXECUTION OPTIONS

Equinox can be configured to run in many different ways. This is done using command line arguments and/or System property settings. Many of the command line arguments can be specified using System properties either on the command line using -D VM arguments, by specifying their values in a config.ini file or a <launcher>.ini file. Using the two latter techniques it is possible to customize your Eclipse without using command line arguments at all.

Key Equinox Execution Options, continued

Below are the key command line arguments and corresponding properties in {}. For more options and information, see the Eclipse Help page **Platform Plug-in Developer Guide > Reference > Other reference information > Runtime Options**.

-application <application id> {eclipse.application}	Gives the identifier of the application to run.
-clean {osgi.clean}	Any cached data used Equinox is flushed. This includes the caches used to store bundle dependency resolution and extension registry data.
-configuration <configuration area path> {osgi.configuration.area}	Sets the configuration location for this session. The configuration determines what plug-ins are run as well as various other system settings.
-console [port] {osgi.console}	Causes the Equinox console to be started. If the given value is a suitable integer, it is interpreted as the port on which the console listens and directs its output to the given port. The console is extremely handy for investigating the state of the system.
-consoleLog {eclipse.consoleLog}	Echoes any log output to Java's System.out (typically back to the command shell if any). Handy when combined with –debug.
-data < data area path> {osgi.instance.area}	Sets the instance data location for this session. Plug-ins use this location to store their data. For example, the Resources plug-in uses this as the default location for projects (aka the workspace).
-debug [options file path] {osgi.debug}	Puts Equinox into debug mode. If the value is a string it is interpreted as the location of the .options file. This file indicates what debug points are available for a plug-in and whether or not they are enabled. If a location is not specified, the platform searches for the .options file under the install directory.
-noExit {osgi.noShutdown}	Causes the Java VM to continue running after Equinox has finished execution. This is useful for examining the framework when the application exits unexpectedly.
-vm <path to java vm>	This option is used by the Equinox executable (e.g., eclipse.exe) to locate the Java VM to use to run Equinox. It should be the full file system path to an appropriate: Java jre/bin directory, Java Executable, Java shared library (jvm.dll or libjvm.so), or a Java VM Execution Environment description file. If not specified, the executable uses a search algorithm to locate a suitable VM. In any event, the executable then passes the path to the actual VM used to Java Main using the -vm argument. Java Main then stores this value in eclipse.vm.
-vmargs [vmargs*]	This option is used to customize the operation of the Java VM to use to run Equinox. If specified, this option must come at the end of the command line. Even if not specified on the executable command line, the executable will automatically add the relevant arguments (including the class being launched) to the command line passed into Java using the -vmargs argument. Java Main then stores this value in eclipse.vmargs.
eclipse.ignoreApp	Setting this property to "true" causes Equinox to simply startup and then exit rather than trying to start an application. This is useful in conjunction with –noExit to start a framework and leave it running.
osgi.bundles	The comma-separated list of bundles which are automatically installed and optionally started once the system is up and running. Each entry is of the form: `<URL | simple bundle location>[@ [<start-level>] [":start"]]` If the start-level (>0 integer) is omitted then the framework will use the default start level for the bundle. If the "start" tag is added then the bundle will be marked as started after being installed. Simple bundle locations are interpreted as relative to the framework's parent directory. The start-level indicates the OSGi start level at which the bundle should run. If this value is not set, the system computes an appropriate default.

LAUNCHER INI FILE

The eclipse.exe can read parameters from an associated ini file by the same name but with the .ini extension (e.g., eclipse.ini). You can specify any parameters in this file but it is recommend to only specify the vm location and the vm arguments in this file and use the config.ini file for others.

The <launcher>.ini must be named after the executable name (e.g., eclipse.exe reads eclipse.ini, whereas launcher.exe reads launcher.ini) and every parameter must be specified on a new line in the file. Here is an example of such a file specifying the vm location and some parameters:

```
-vm
c:/myVm/java.exe
-vmargs
-Dms40M
```

GET MORE INFORMATION

Equinox website and wiki • http://eclipse.org/equinox • http://wiki.eclipse.org/equinox	The home of the Equinox project. Find downloads, tutorials and getting started guides, project plans, contribute patches and bug reports.
OSGi Alliance • http://osgi.org	The home of OSGi. Get copies of the spec and find more information on the technology.
Equinox OSGi book • http://equinoxosgi.org	The definitive guide to Equinox. Get comprehensive code samples and connect to the book on Safari.
Apache Felix • http://felix.apache.org	The home of the Felix OSGi implementation. Get more bundles and find other OSGi-minded people

ABOUT THE AUTHOR

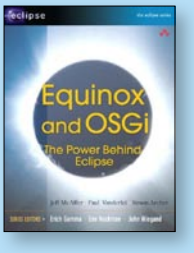
Jeff McAffer

Jeff McAffer leads the Eclipse Equinox OSGi, RCP and Orbit teams and is co-founder and CTO of EclipseSource. He is one of the architects of the Eclipse Platform and a co-author of *The Eclipse Rich Client Platform* and the upcoming book *Equinox and OSGi*. He co-leads the RT PMC and is a member of the Eclipse Project PMC, the Tools Project PMC and the Eclipse Foundation Board of Directors and the Eclipse Architecture Council. Jeff is currently interested in all aspects of Eclipse components from developing and building bundles to deploying, installing and ultimately running them.

Publications

- The Eclipse Rich Client Platform (Addison Wesley)
- Equinox and OSGi - The power behind Eclipse (Addison Wesley)

RECOMMENDED BOOK



Equinox and OSGi incrementally guides you through building an application. It focuses on OSGi and its application to building highly modular and dynamic systems. Equinox and Eclipse are used throughout, but the lessons and code should be applicable to anyone writing OSGi systems.

BUY NOW
books.dzone.com/books/equinox-osgi

Get More FREE Refcardz. Visit refcardz.com now!

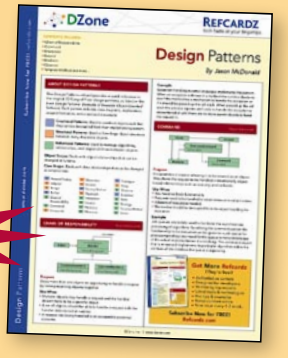
Upcoming Refcardz:

SOA Patterns
Essential EMF
Windows Presentation Foundation
HTML and XHTML
SOA Governance
Agile Methodologies

Available:

Getting Started with Equinox & OSGi
Core Mule
Core CSS: Part III
Using XML in Java
Essential JSP Expression Language
Getting Started with Hibernate Search
Core Seam
Essential Ruby
Essential MySQL
JUnit and EasyMock
Spring Annotations
Getting Started with MyEclipse
Core Java
Core CSS: Part II
PHP
Getting Started with JPA
JavaServer Faces
Core CSS: Part I
Struts2
Core .NET

FREE



Design Patterns
Published June 2008

Visit refcardz.com for a complete listing of available Refcardz.

DZone

DZone communities deliver over 4 million pages each month to more than 1.7 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheatsheets, blogs, feature articles, source code and more. "DZone is a developer's dream," says PC Magazine.

DZone, Inc.
1251 NW Maynard
Cary, NC 27513
888.678.0399
919.678.0300
Refcardz Feedback Welcome
refcardz@dzone.com
Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-934238-38-7
ISBN-10: 1-934238-38-4
50795
9 781934 238387
$7.95