17

Working with Geronimo Plugins

With the release of Geronimo 1.1, the ability to easily add incremental system and application features to a Geronimo server becomes a reality. Geronimo plugins enable J2EE applications and resources to be preconfigured and then installed into any compatible Geronimo server with a single command or click. Plugins also work on a system level, and enable system modules to be installed into a minimally configured server (such as the Little G minimal server distribution) with equal one-click ease.

Before the arrival of the plugin mechanism, users had to configure and deploy J2EE applications and resources manually on each Geronimo server. This can be a tedious process that involves complex roll-out planning when a large number of Geronimo servers are involved.

Before plugins, Geronimo administrators were also forced to select one of the preconfigured Geronimo assemblies: either a full-fledged J2EE 1.4 server, or a minimal Web tier only server. The full-fledged J2EE assembly may be an overkill for many applications, while the Little G assembly may be inadequate for an application. Plugins allows a Geronimo instance to be custom configured to a user's requirements, without being forced to select one of the one-size-fits-all canned assemblies, and without the need to learn how to compile and re-build the entire Geronimo server from source code. And therein lies the real power behind plugins.

This chapter examines Geronimo plugins, including the following:

* The Geronimo plugin architecture, and how it fits into the larger Geronimo architecture
* The composition of a plugin
* Plugin dependencies and prerequisites
* Plugin repositories
* XML plans associated with plugins
* Searching and installing plugins
* Installing standalone plugins

* Setting up a plugins repository
* Using plugins to clone Geronimo configuration across server instances

By the time you finish the chapter, the architecture and concepts behind Geronimo plugins will become familiar to you. You will be able to explore all the plugins and additional features available to you in public plugin repositories. You will also be able to create your own plugins and manage your own plugin repository.

Geronimo Plugin Architecture

While the concept of plugins is not new, the ability to install preconfigured modules as plugins, together with the ability to add system modules using one-click installation, are certainly advanced concepts for a J2EE application server. For most J2EE servers, however, having a component-based building block construction doesn't make much sense. Especially for commercial servers, there is little need to fabricate the server itself in terms of components. This is because commercial J2EE servers are designed to do one and only one single thing well — act as a J2EE compatible application server at all times.

Component Based Architecture

In this respect, Geronimo is markedly different from the competition. Geronimo is designed from the start to be a server construction kit — aiming to make the creation of robust, manageable, and high-performance servers (network servers of almost any kind) possible using the Java programming language. The fact that users typically use Geronimo as a J2EE 1.4 server is simply one example of the infinite possibilities that Geronimo provides. In fact, the J2EE distribution of the Geronimo server also includes two independent servers:

* A relational database server: Derby
* A message queue server: the ActiveMQ Broker

These two independent servers take advantage of the modular architecture of Geronimo. It is a matter of time before you will see other servers taking advantage of the Geronimo foundation. Chapter 18 provides a more extensive view of how the Geronimo server is constructed out of preconfigured assemblies of modules.

This modular-design is pervasive throughout the construction of the J2EE personality of the Geronimo server, and passes consistently to the application level, where components grouped by modules can be configured and deployed to the server — essentially enhancing the overall capabilities of the server. From a J2EE perspective, deploying applications to Geronimo involves the deployment of a EAR or WAR module to the server. From a generic Geronimo perspective, this is no different than the configuration and deployment of new modules to add new features to a base server.

The modularity of Geronimo is completely different from other J2EE servers, and one where it makes sense to have a facility to incrementally add functionality to the server stack. This facility to incrementally add features to a Geronimo server is the plugins mechanism built into Geronimo 1.1 and beyond.

More than Deployable Modules

When you first hear about Geronimo plugins, the first question that comes into mind may be, "If Geronimo can already handle run-time deployable binary modules (such as WAR and EAR), why is there a need for a plugin at all?"

The simple answer is that plugins are much more than just deployable modules.

Before you can successfully deploy application modules to Geronimo, you will need to perform the following configuration steps:

* Determine if all the prerequisites of the module are satisfied.
* Determine if all the dependencies of the module can be satisfied; if not, download and install them.
* Prepare any Geronimo specific deployment plans that may be necessary (such as `geronimo-web.xml`, `geronimo-application.xml`).

During module deployment, the Geronimo deployer invokes a target-specific builder to parse the deployment plan. The builder configures the GBeans in a module according to the values specified in the deployment plan, providing the GBeans with an initial starting state. For example, there is an EJB builder that knows how to configure and load EJBs. There is a Tomcat builder that knows how to configure and load JSPs and servlets into Tomcat. And there is a Jetty builder that configures JSPs and servlets for Jetty. The output of these builders is a binary module configuration (in serialized format) that is placed into the repository. A successful deployment is always a combination of this binary module configuration with the actual executable binaries of the module, as shown in Figure 17-1.

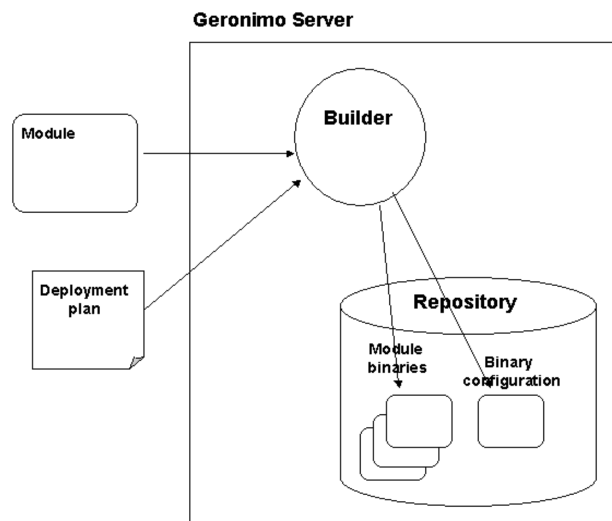

Figure 17-1: Deployment and creation of binary configuration

The serialized binary configuration allows the Geronimo server to load and start a configured module quickly, and without re-configuration. A module can be quickly loaded, its GBeans instantiated, initialized to a known initial state using the binary configuration, and started. This serialized binary configuration is a vital part of every plugin.

Composition of a Plugin

A Geronimo plugin is a preconfigured module. This preconfigured module is physically stored in a JAR archive, and it contains the following:

* The module binaries
* The serialized module configuration in binary form
* An XML file providing metadata regarding the plugin

The last item, the XML file, should be placed in the `META-INF` directory of the plugin archive, and be named `geronimo-plugin.xml`. A plugin archive is typically placed into a CAR file (Configuration ARchive). Figure 17-2 illustrates the composition of a Geronimo plugin.

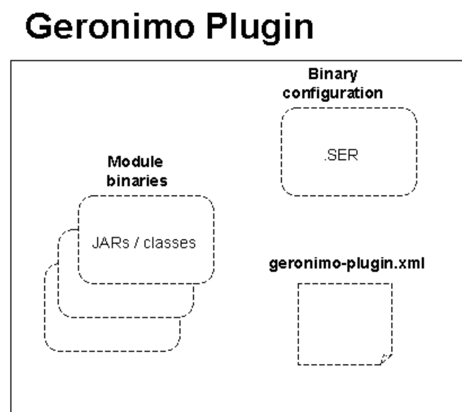

Figure 17-2: Composition of a Geronimo plugin

Plugins are more than deployable modules because all plugins, by definition, have already gone through the deployment process successfully. This is the secret of one-click installation. As long as the plugin is being installed into a similarly configured server, with all the prerequisites available and dependencies installed, the plugin is ready to run.

If you examine a plugin CAR file, you will see the following items:

* `META-INF/config.ser`
* `META-INF/config.info`
* `META-INF/config.ser.sha1`
* The actual class files of your the plugin module and included dependencies

Note that the filenames and extension under `META-INF` may change with future versions of Geronimo because of design changes, but the content is essentially the same.

From a Geronimo administrator point of view, plugins can be interesting for several reasons. You can download and install plugins to add features to your server, and you can also create your own plugins either to rapidly install features/applications to multiple Geronimo servers, or to share them with others (internal to your enterprise or over the Internet). In any case, you will need to have a place to put the plugins, and expose them in a way that other Geronimo users can find and install them. This is the job for a plugin repository.

About Plugin Repositories

A *plugin respository* is a location where Geronimo plugin installer can locate and install available plugins. You typically only need the URL to a plugin repository to download and install plugins.

At the plugin repository, the following must exist:

* A `geronimo-plugins.xml` file describing all the plugins available
* A Maven 2 repository from where the plugins can be downloaded

Although it is usually the case, the Maven 2 repository need not be located on the same physical host machine as the `geronimo-plugins.xml`. The `<default-repositories>` element in the `geronimo-plugins.xml` file can point the plugin installer to other location to find the Maven 2 repository. You will see the detailed layout of this `geornimo-plugins.xml` later in this chapter.

Because of concerns for impartiality, Geronimo, by default, does not come with any plugin URL. However, you can click an Update Repository List link on the Web console to get a list of available public repositories.

Maven Coordinates

A Geronimo plugin repository is a Maven 2 repository. In earlier chapters, you have been shown that Geronimo borrows many concepts from the design of Maven — and in particular, that the respository at the heart of a Geronimo server was inspired by the design of the Maven repository. In fact, it is possible to set up a Geronimo server to act as a Maven repository for the purpose of plugins download and installation. This is shown later in this chapter in the section, “Cloning System Configurations.”

One very important Maven concept is that of a *Maven coordinate*. Basically, a Maven coordinate is a 4-tuple that specifies a unique artifact. You can think of an artifact as a Geronimo module. The coordinate consists of the following:

* Group or organization
* Artifact ID
* Version
* Type

For example, a set of Maven coordinate may be (wrox, authorlist, 1.1, car). This can also be written as `wrox/authorlist/1.1/car`. If you think this looks like a Geronimo module ID, you are right on the money. The format of the module ID is directly borrowed from Maven. The same Maven coordinate expressed as a module id would be:

```
<moduleId>
  <groupId>wrox</groupId>
  <artifactId>authorlist</artifactId>
  <version>1.1</version>
  <type>car</type>
</moduleId>
```

This compatibility allows Geronimo modules to be stored in Maven directory without any transformation effort.

Dependencies

Dependencies are other modules and artifacts on which the plugin depends to be installed successfully. Chapter 18 provides some insight into how dependencies are managed internally by Geronimo, but for the purpose of plugins, the specified dependencies must be started on the server before the plugin can be installed and started. Dependencies are specified via the `<dependency>` element, an example is:

```
<dependency>
      <groupId>geronimo</groupId>
      <artifactId>j2ee-server</artifactId>
      <type>car</type>
  </dependency>
```

The missing `<version>` tag in the preceding dependency implies that any version of the `j2ee-server` module should work with the plugin.The Geronimo plugin installer is quite smart when it comes to dependencies. By examining the metadata provided in the `META-INF/geronimo-plugin.xml`, specifically a `<default-repositories>` element, the plugin installer can download and run specified dependent modules if necessary. Once a dependency is downloaded, it is placed into the local repository and future invocation of the same plugin will not require further remote download. In addition, other modules or plugins can make use of these dependent modules without additional download. This automatic download mechanism is a big time saver for Geronimo administrators; the only other alternative being to manually download, configure, and deploy using the console

Bundling Decision on Dependencies

An important decision that every plugin creator must make is whether to include a specific dependency as part of the plugin bundle, or leave it for automatic download by the plugin installer. Of course, bundling a dependency as part of the plugin bundle will increase the bundle's size. However, it also ensures that the dependency is available and of the right version. This can both speed up the installation and startup, and also provide some guarantee that the plugin user is able to successfully start a plugin.

On the other hand, the dependencies can be specified in the metadata, but left out of the bundle. In this case, the plugin installer will search through the specified `<default-repositories>` and scan for the dependencies to download. This process takes additional time, and bears the risk that the specified dependencies cannot be downloaded. This can happen if the target repository with the required dependencies is down, or if the network connection is unavailable. In these cases, the plugin cannot be installed or started successfully.

However, leaving dependencies out of plugin bundles is often the best decision for several reasons. First, it keeps the size of the plugin bundle small. Second, many dependencies are system- or server-level dependencies, and leaving them out allows them to be shared by other modules or plugins that you install in the future. Third, often downloaded dependencies may have (transitive) dependencies of their own, and your decision to include them many need to be considered recursively, further bloating the bundled distribution size of a plugin. In fact, including all transitive dependencies may not even be possible. Chapter 18 provides an exploration on how Geronimo system modules deal with transitive dependencies internally.

Plugin Prerequisites

Prerequisites are environment considerations that must be met before the plugin can be installed. The difference between dependencies and prerequisites is the fact that a plugin installer cannot automatically satisfy prerequisites. Unsatisfied prerequisites always require external manual intervention.

Prerequisites for a plugin are specified in the metadata, typically in the `META-INF/geronimo-plugin.xml` file within `<prerequisite>` element.

Prerequsites sound like a to-do list, prior to plugin installation. However, prerequisites are more than a to-do list. The unique value added of plugin prerequisites includes the following:

* Geronimo can sometimes detect whether a prerequisite has been satisfied.
* The description and instructions on what a prerequisite entails, and documentation on how it may be satisfied, is included at the location where it matters — bundled with the plugin as metadata and displayed at the time of installation.

These two points are highly relevant to plugin users, and can save a lot of tedious work determining if a system has certain prerequisites, or is looking for an associated to-do list for a previously downloaded plugin.

Concrete examples of specific plugin prerequisites include the following:

* SUN’s JDK 1.4 for CORBA support
* Jetty container (instead of Tomcat)
* A connection from the Geronimo server to an external corporate Oracle database

Note that in these cases, the prerequisites cannot possibly be satisfied by the plugin installer.

Basic Plugin Operations

As a Geronimo administrator, your working experience with plugins is typically limited to the following:

* Searching for plugins
* Installing plugins

Advanced users may also perform the following activities:

* Creating plugins
* Hosting a plugin repository for the department, enterprise, or the Internet
* Using the plugin mechanism to clone an applications profile across a network of similarly configured Geronimo servers

The rest of this chapter will explore the myriads of ways you can perform some of these activities.

Plugin Repositories

Plugin repositories are locations where available plugin are maintained for searching and downloads. In practice, plugin directory can be public (on the Internet) or private (local to a department, enterprise, or group).

One of the most popular public Geronimo plugin repositories can be found at the following URL:

```
http://www.geronimoplugins.com/
```

Geronimo comes, by default, with no knowledge of any plugin repositories. However, with the click of a single link on the Web console, a list of available repositories can be downloaded automatically from a plugin repository list maintainer.

Customizing the Plugin Repositories List Maintainer

The list of available plugins can be downloaded by clicking on the Update Repository List link in the Web console. When the link is clicked, the Web console fetches a list of repositories from a URL pointing to a list maintained by a plugin repositories list maintainer.

This list is a simple text file, with a repository entry on each line. Any Web server on the network can serve this list. For example, a list containing two repositories may be similar to the following:

```
http://accounting.wrox.com/repository/geronimo-1.1/
http://development.wrox.com/repository/geronimo-1.1/
```

Geronimo discovers the URL of the repositories list maintainer from the `repositoryList` attribute of the `DownloadedPluginRepos` GBean. This attribute can be customized in the `var/config/config.xml` file (make sure Geronimo is not running when you edit this file).

```
<?xml version="1.0" encoding="UTF-8"?>
<attributes xmlns="http://geronimo.apache.org/xml/ns/attributes-1.1">
  <module name="geronimo/rmi-naming/1.1/car">
    <gbean name="RMIRegistry">
```

```
      <attribute name="port">1099</attribute>
    </gbean>
    <gbean name="NamingProperties">
      <attribute name="namingProviderUrl">rmi://0.0.0.0:1099</attribute>
    </gbean>
    <gbean name="DownloadedPluginRepos">
      <attribute name="repositoryList">http://people.apache.org/~ammulder/plugin-
repository-list-1.1.txt</attribute>
      <attribute name="userRepositories">[]</attribute>
    </gbean>
  </module>
…
```

The default value for the `repositoryList` attribute is `http://people.apache.org/~ammulder/plugin-repository-list-1.1.txt` . This default maintainer provides a list of public repositories accessible over the Internet. By pointing this attribute to your own maintainer list, users can download a list of repositories specific to your department or enterprise on your Intranet.

The plugin installer will cache any downloaded repository entry back into the `config.xml` file — enabling the selection of the repository without accessing the plugins list maintainer again. This information is stored with a `downloadRepositories` attribute, shown in the following highlighted line:

```
<?xml version="1.0" encoding="UTF-8"?>
<attributes xmlns="http://geronimo.apache.org/xml/ns/attributes-1.1">
  <module name="geronimo/rmi-naming/1.1/car">
    <gbean name="RMIRegistry">
      <attribute name="port">1099</attribute>
    </gbean>
    <gbean name="NamingProperties">
      <attribute name="namingProviderUrl">rmi://0.0.0.0:1099</attribute>
    </gbean>
    <gbean name="DownloadedPluginRepos">
      <attribute name="repositoryList">http://people.apache.org/~ammulder/plugin-
repository-list-1.1.txt</attribute>
       <attribute
name="downloadRepositories">[http://www.geronimoplugins.com/repository/geronimo-
1.1/]</attribute>
      <attribute name="userRepositories">[]</attribute>
    </gbean>
  </module>
…
```

The `downloadRepositories` attribute is a comma-separated list of repository URLs.

Adding Plugin Repositories for the Plugin Installer

In addition to fetching a list of repositories from a repositories list maintainer, you can also add your own repositories to the available list without accessing any list maintainer. This may be useful in the case where a Web server used to serve the repositories list may not be available.

Repositories can be added to the `userRepositories` attribute of the `DowloadedPluginRepos` GBean, in the `var/config/config.xml` file. For example, after manually adding two repositories to the list, the `config.xml` will look similar to the following:

```
 <?xml version="1.0" encoding="UTF-8"?>
<attributes xmlns="http://geronimo.apache.org/xml/ns/attributes-1.1">
  <module name="geronimo/rmi-naming/1.1/car">
    <gbean name="RMIRegistry">
      <attribute name="port">1099</attribute>
    </gbean>
    <gbean name="NamingProperties">
      <attribute name="namingProviderUrl">rmi://0.0.0.0:1099</attribute>
    </gbean>
    <gbean name="DownloadedPluginRepos">
      <attribute name="repositoryList">http://people.apache.org/~ammulder/plugin-
repository-list-1.1.txt</attribute>
      <attribute name="userRepositories">[
http://accounting.wrox.com/repository/geronimo-1.1/,
http://development.wrox.com/repository/geronimo-1.1/]</attribute>
    </gbean>
  </module>
…
```

Searching and Installing Plugins

You can search for and install plugins into your Geronimo server using either the command-line deployer or from the Web console. The following sections show both styles of plugin installation.

Using Command Line Deployer

There are two plugin-specific commands for the command-line deployer, as shown in Table 17-1.

Table 17-1: The Plugins-Specific Commands of the Command-Line Deployer

Command	Description
`search-plugins`	Search for available plugins in a specified repository URL. If not specified, the command-line deployer will use the list in the `config.xml` file (if there is any entry there). This command will also display each of the available plugins at the repository. The user can select a plugin, and the command-line deployer will download and install the plugin.
`install-plugin`	This works with a plugin CAR file that has already been downloaded. The command-line deployer will download all required dependencies, extract the binaries from the bundle, and place them into the Geronimo repository.

For example, to search plugins from `geronimoplugins.com`, you can use the following command:

```
deploy search-plugins http://www.geronimoplugins.com/repository/geronimo-1.1/
```

The command-line deployer will fetch the `geronimo-plugins.xml` file from the repository and display all the plugins available. A typical output from this command may be the following:

```
    Security
        1 : Apache Directory 0.92 for Geronimo (1.1)
```

```
        Portal
            2 : Liferay Derby Database Pool (4.0.0)
            3 : Liferay Portal Enterprise version 4.0.0. (4.0.0)

        Core Geronimo
                Geronimo Admin Console (Jetty) (1.1)
                Geronimo Admin Console (Tomcat) (1.1)
                Geronimo Welcome Web App (Jetty) (1.1)
                Geronimo Welcome Web App (Tomcat) (1.1)

        Scheduling
            4 : Quartz Job Deployer (0.1)
            5 : Quartz Job Deployer (0.2)
            6 : Quartz Scheduler Integration (0.1)
            7 : Quartz Scheduler Integration (0.2)

        Examples
                Jakarta JSP Examples (Jetty) (1.1)
            8 : Jakarta JSP Examples (Tomcat) (1.1)
                LDAP Example Web App (Jetty) (1.1)
            9 : LDAP Example Web App (Tomcat) (1.1)
            10: LDAP Example Security Realm (1.1)
                Jakarta Servlet Examples (Jetty) (1.1)
            11: Jakarta Servlet Examples (Tomcat) (1.1)

        Resources
                Oracle XA Driver for Console (Jetty) (1.0)
            12: Oracle XA Driver for Console (Tomcat) (1.0)

    Install Service [enter number or 'q' to quit]:
```

You can select any one of the available plugins by number, and the command-line deployer will download and install the selected plugin.

Note that some plugins are not available for selection. In these cases, the plugin installer has detected that the Geronimo server only in this case only supports Tomcat, and all the plugins with Jetty as prerequisites are not available for selection. In addition, any plugin of matching version that is already installed in the server is also not available for installation.

Use of the `install-plugin` command is explained later in the section, “Installing Standalone Plugins.”

Using the Web Console

The Web console can also be used to search and install plugins. First, you will need to update the repository list. Select the Plugins@@raCreate/Install link from the Web console menu, then click on the Update Repository List link. Figure 17-3 shows this link on the Web console.

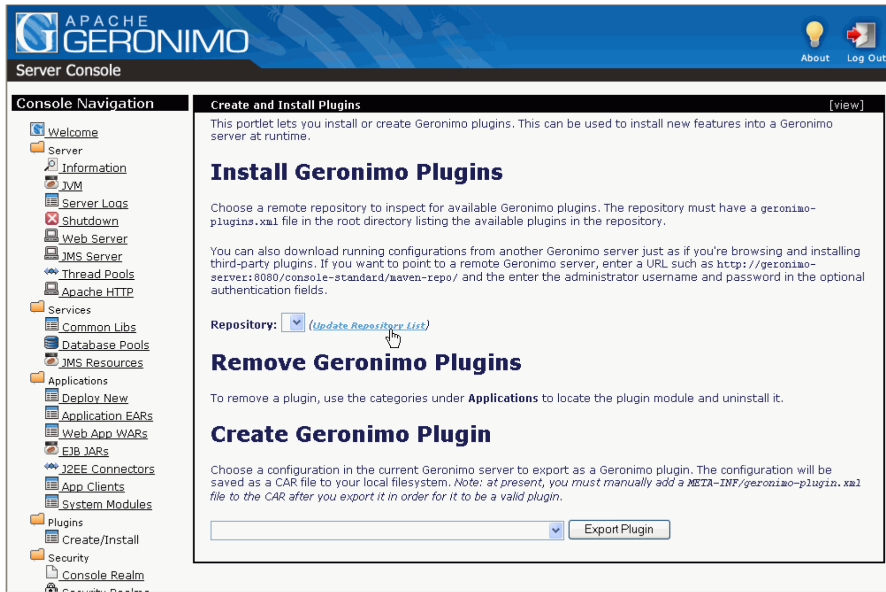

Figure 17-3: Updating the Repository list on the Web Console

After the update, a list of available repositories appears, and you can now search for plugins via the Search for Plugins button, as shown in Figure 17-4.

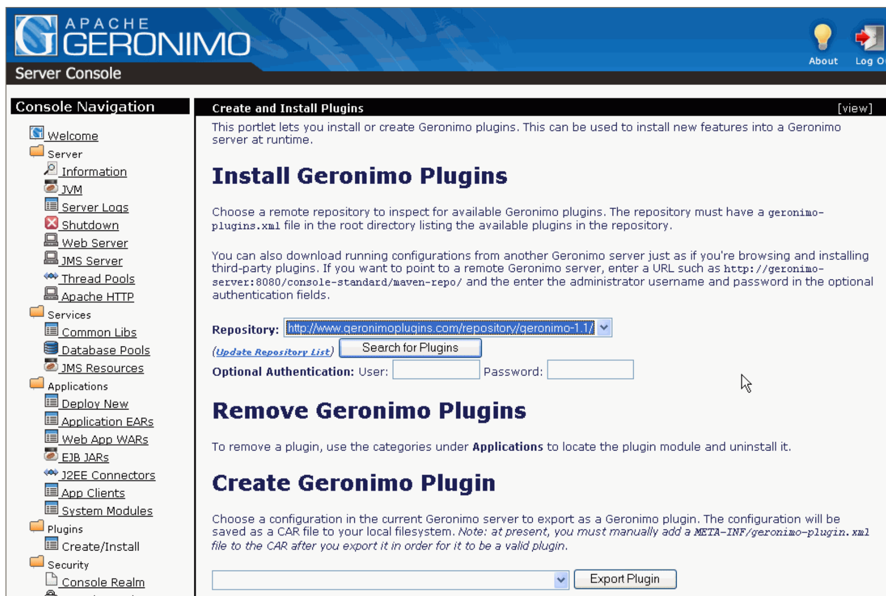

Figure 17-4: Updated repository list and the Search for Plugins button

If you are using a repository that requires authentication, you can enter the user and password required in the screen shown in Figure 17-4. If you use the plugins installer to clone Geronimo server

configurations (covered later in this chapter), you will need to enter the login information to the staging server's console.

After selecting a repository from the available repositories list and clicking the Search for Plugins button, the Web console generates a description of the available plugins at the selected repository. Each plugin description contains a link that can be clicked to install the associated plugin, as shown in Figure 17-5.

Figure 17-5: Display of available plugins for download and installation

The entire list in Figure 17-5 is generated from the plugin metadata information contained in the `geronimo-plugins.xml` file of the selected Geronimo repository.

Installing Standalone Plugins

Intead of installing plugins live over a network, sometimes it may be more convenient to simply to receive and install plugins as a standalone bundle. For example, a department and/or an enterprise may elect to distribute new features on writable DVDs. In these cases, you can use the command line deployer's `install-plugin` command. The basic syntax is as follows:

```
deploy install-plugin  <name of the CAR file>
```

This will start the plugin installer and it will look inside the bundle for metadata in the `META-INF/geronimo-plugin.xml` file. From the metadata, it will determine the dependencies that must be downloaded and finally install the plugin.

For example, Figure 17-6 shows the installation of the JSP samples plugin. This plugin is downloaded as a CAR file from `geronimoplugins.com` and simply adds the reference JSP examples to a Geronimo server.

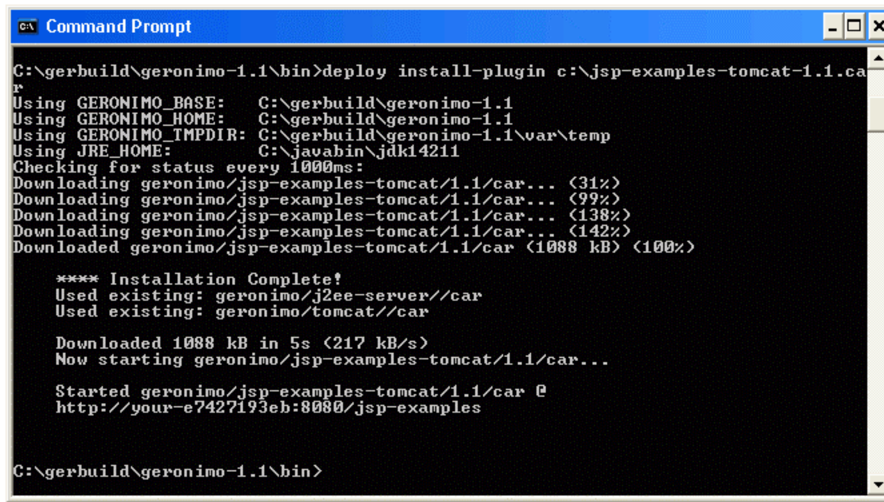

Figure 17-6: Using the command-line deployer to standalone install plugin

Once you have completed plugin installation successfully, the modules that are installed are indistinguishable from manually configured and deployed modules. To uninstall them, you must perform explicit undeployment of the plugin's module. Undeploying the module will remove it from both the local repository and the hot deploy directory.

Applying Plugins

Plugins can be used to distribute new J2EE applications or resources to Geronimo server instances, without tedious re-configuration at every site. They can also be used (by Geronimo system developers — either the Apache team or third-party) to distribute new server features to Geronimo users.

Enhancing Server Features

System developers are taking advantage of the plugin architecture to provide enhancements to the Geronimo server in the form of easy-to-install plugins. For example, the Apache Directory service can be installed as a plugin from `geronimoplugins.com` if you need to add directory service to your Geronimo system. Another example is the Quartz job scheduler (allowing cron-like scheduling from your Geronimo server) also available as a simple plugin install.

An example of useful resource-based plugin is the Oracle XA driver. This is a JDBC driver module, supporting XA, that has been bundled up as a plugin for ease of distribution and configuration.

Deploying Across a Farm of Geronimo Servers

When you need to similarly install a bank of Geronimo servers, there is no need to individually configure each server. It is possible to make good use of the plugin mechanism to "clone" a server configuration across the server instances.

The idea of cloning is to prepare a staging or reference server with all the applications and system enhancement, and then make that server available as a plugin repository. Other server instances that

must be set up can then access this staging server's repository to pull the configured modules as plugins and install them. Figure 17-7 illustrates the cloning process.

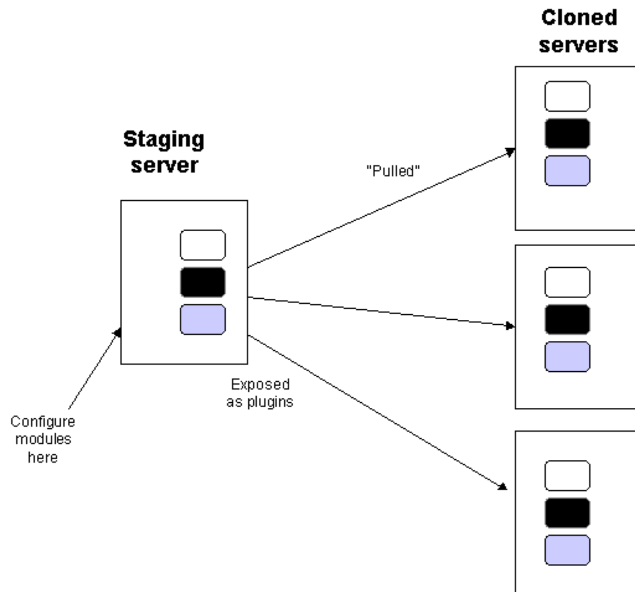

Figure 17-7: Cloning a farm of Geronimo servers

Creating Plugins

While it is possible to create your own CAR archive manually, the system support built into the Geronimo Web console makes creating a plugin straightforward.

First of all, it is vital to reiterate that a plugin is simply a packaged, preconfigured module. This implies that the plugin should have been successfully deployed and running previously on some Geronimo server. It would be quite pointless to bundle up a badly configured module as a plugin. Because of this, the Geronimo Web console provides the ability for you to create a plugin of any running application or resource modules on your system. Since the module is already running, the deployment process must have been successful, and the configuration must be valid.

Creating a Plugin from a Running Module

Using the Web console to create a plugin from a running module is quite straightforward. First, select the Plugins@@raCreate/Install menu on the left. This will cause the `Create and Install Plugins` portlet to display on the right. On the bottom of the page, you will have a selection of currently running modules to select from, as shown in Figure 17-8.

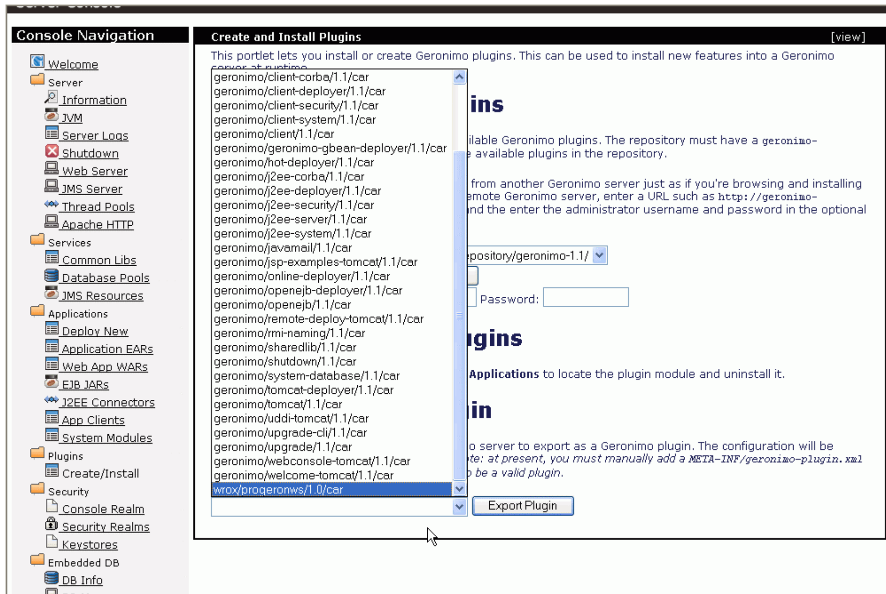

Figure 17-8: Creating a plugin from a running module

After selecting the running module to be made into a plugin, you can then click the Export Plugin button. The next screen will prompt you to enter metadata for the plugin. This is the very same metadata that will end up in the `META-INF/geronimo-plugin.xml` file. Figure 17-9 shows the prompt for the metadata.

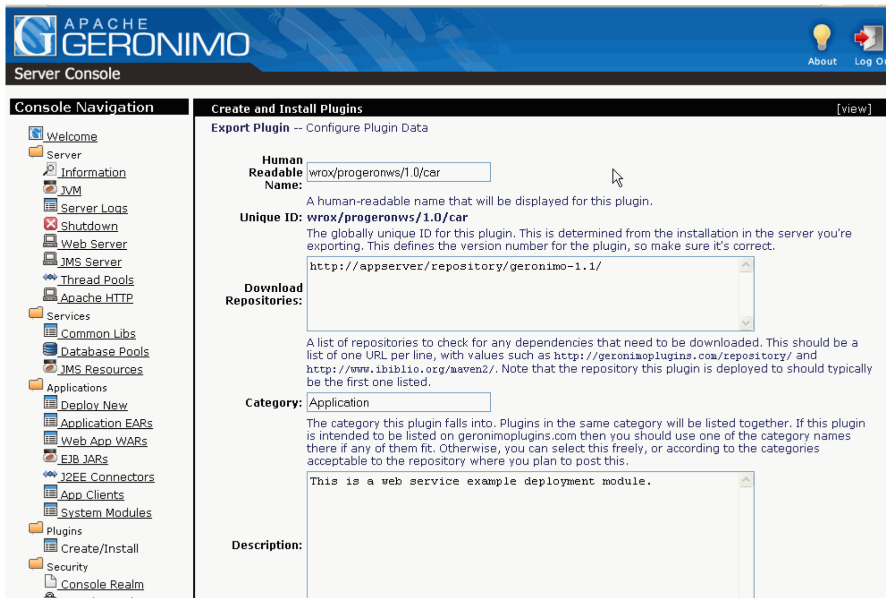

Figure 17-9: Entering metadata for the plugin

The next section takes a detailed look at the metadata that can be included with the plugin.

The Plugin Descriptor (geronimo-plugin.xml)

The document (main) element in a `geronimo-plugin.xml` metadata file is the `<geronimo-plugin>` element. This element contains all the auxiliary information about the plugin, including dependencies, prerequisites, repository, and so on. This is also the same metadata information element that must be placed into the `geronimo-plugins.xml` (notice the "s") file if you are staging a plugin repository. In the case of `geronimo-plugins.xml`, there must be one `<geronimo-plugin>` element for each plugin that the repository serves. Table 17-2 tabulates the subelement of the `<geronimo-plugin>` element.

Table 17-2: Subelements of the <geronimo-plugin> Element

Subelement	Description
`name`	The name of the plugin, this will appear in the selection when users search for plugins.
`module-id`	The Geronimo module ID associated with the plugin (for example, `wrox/authorlist/1.1/car`). If this is left empty, a plugin group will be installed. Each of the plugins in a group will be specified by an entry in the `<dependency>` element.
`category`	The category that this plugin belongs to. Arbitrary classification at this time, no standards yet. Although there may be conventions to follow if you are distributing via public repositories.
`description`	Description of a plugin. This is optionally displayed by plugin installation tools. Provides more information on the content and how the plugin may be used.
`url`	A URL to a site where more information on the plugin can be found. This is often the project site from where the plugin originates.
`author`	The person, company, organization, or entity that authored the plugin.
`license`	The release license of the software. Specify type (such as `bsd` or `gnu`), followed by a name that qualifies the license further (such as `Apache License 2.0`).
`hash`	Hash code for validation of the plugin integrity. The type attribute specified the algorithm, such as "`md5`" or "`sha-1`".
`geronimo-version`	The version of Geronimo for which this plugin is designed to support.

jvm-version	The Java VM version that this plugin supports.
prerequisite	The modules that must be pre-installed before the plugin will run. The <resource-type> and <description> subelements can be used to further describe the prerequisite, or to provide instructions to the user on how to install or prepare the prerequisite.
dependency	The libraries and modules on which this plugin depend. The plugin installer will search for and download (if necessary) these modules before starting the plugin. If the <module-id> element is blank. This list represents a group of plugins to install.
obsoletes	The modules that this plugin replaces, or make obsolete. This often lists prior versions of the same plugin.
source-repository	A list of repositories from which dependencies for this plugin can be located and downloaded. The plugin installer will search the repositories in the order specified in the list.
copy-file	Files that needs to be copied from the plugin distribution the Geronimo directory tree.
config-xml-content	The additional content in the config.xml file that relates to this plugin. Allows the plugin to expose GBeans and properties that can be configured by the user by editing the config.xml file.

Plugin Groups

It is sometimes desirable to install a group of plugins all with the same prerequisites and dependencies. In theses cases, a plugin group can be specified in a <geronimo-plugin> element. This is done as follows:

* Leaving the module-id element empty
* Placing the module-id of plugins to be installed as a group in the <dependency> element

An example of a plugin group, consisting of two plugins, is shown here with the plugins highlighted:

```
  <?xml version="1.0" encoding="UTF-8" ?>
- <geronimo-plugin-list>
- <plugin>
  <name>wrox/plugingroup/1.1/car</name>
  <module-id> </module-id>
  <category>plugin group example</category>
  <description>From the Professional Geornimo</description>
  ...
```

```
  <dependency>
     wrox/authorlist/1.1/car
  </dependency>
  <dependency>
     wrox/bookstore/1.1/car
  </dependency>
 </plugin>
  ...
</geronimo-plugin-list>
```

Your Own Plugin Repository

To host your own plugin repository, you have at least two choices:

* Host a Maven 2 repository on a Web server, and then create your own `geronimo-plugins.xml` file
* Use an existing Geronimo server as a Maven repository

The latter alternative is by far the simplest. If you are interested in setting up a full Maven repository, see the Maven information at the following URL:

```
http://maven.apache.org/
```

Note that in Maven documentation, you will frequently see the mention of plugins. Maven plugins are for building software projects, and are quite different from Geronimo plugins — which are features and applications for the Geronimo server.

The geronimo-plugins.xml Repository Descriptor

All Geronimo plugin repositories must have a `geronimo-plugins.xml` file at the root URL. For example, say the repository's URL is the following:

```
http://accounting.wrox.com/repository/geronimo-1.1/
```

Then there should be an accessible `geronimo-plugins.xml` at the root:

```
http://accounting.wrox.com/repository/geronimo-1.1/geronimo-plugins.xml
```

The format of this `geronimo-plugins.xml` file is as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<geronimo-plugin-list
  xmlns="http://geronimo.apache.org/xml/ns/plugins-1.1">
  <plugin>
   ...
  </plugin>
  <plugin>
   ...
  </plugin>
    ...
  <plugin>
    ...
  </plugin>
  <default-repository>
```

```
    http://www.geronimoplugins.com/repository/geronimo-1.1
  </default-repository>
  <default-repository>
    http://www.ibiblio.org/maven2/
  </default-repository>
</geronimo-plugin-list>
```

The root element is `<geroimo-plugin-list>` and it contains a set of `<plugin>` elements. Each `<plugin>` element describes a single plugin hosted on the repository. Each of the `<plugin>` elements must contain the metadata of a plugin, in the exact same format as the `<geronimo-plugin>` element in Table 17-2.

The list of `<default-repsitory>` enables you to list other servers where the plugin(s) can be downloaded from. This list will be used if the plugin does not have its own repository specified.

The `geronimo-plugins.xml` file can either be manually assembled, or you can generate it dynamically using Geronimo. The next section illustrates this technique.

Using Geronimo as a Maven Repository for Plugins

To use Geronimo as a Maven repository for plugins, you must first install the plugins that you want to make available to others in your own server.

Once you have the plugins installed successfully into your server instance, others will be able to use their Web console to access your repository via the followingURL:

```
http://pluginserver:8080/console-standard/maven-repo/
```

You will need to replace *`pluginserver`* with your server's hostname or IP address.

When you use this URL, you are, in essence, accessing a system Web application called `console-standard`. This Web application has a servlet mapped to the URI `maven-repo` that will generate the required `geronimo-plugins.xml`.

When accessing the repository from another Geronimo server, make sure you enter the username and password required to access the `console-standard` application. By default, this is `system` and `manager`, respectively.

The next section will show a typical usage for Geronimo acting as a plugin repository.

Cloning System Configurations

One of the most compelling uses of a Geronimo acting as a plugin repository is in the area of cloning of system configurations. The idea is that you can configure a single Geronimo instance with all the modules that you need, and then "clone" that instance to a set of similar Geronimo server instances across the work. This is especially useful in enterprises where multiple Geronimo servers must be configured similarly.

In this scenario, you set up a "staging server" with all the modules that you want to install to the servers. Once the staging server is configured, you can point additional Geronimo servers to this staging server and "pull" the configuration as a set of plugins. Each of the system being cloned must add the

staging server's repository URL to the `config.xml` file. The following is a modified `config.xml`, with added line highlighted. You will need to change `stagingserver` to your staging servers's IP addressor host name.

```
<?xml version="1.0" encoding="UTF-8"?>
<attributes xmlns="http://geronimo.apache.org/xml/ns/attributes-1.1">
  <module name="geronimo/rmi-naming/1.1/car">
    <gbean name="RMIRegistry">
      <attribute name="port">1099</attribute>
    </gbean>
    <gbean name="NamingProperties">
      <attribute name="namingProviderUrl">rmi://0.0.0.0:1099</attribute>
    </gbean>
    <gbean name="DownloadedPluginRepos">
      <attribute name="repositoryList">http://people.apache.org/~ammulder/plugin-
repository-list-1.1.txt</attribute>
      <attribute name="userRepositories">[ http://stagingserver:8080/console-
standard/maven-repo/]</attribute>
    </gbean>
  </module>
…
```

After setting up the repository in the `config.xml`, start Geronimo and access the Web console. Click Plugins@@raCreate/Install and select the new URL for the repository. You can then enter the username and password to the staging server, and then click the Search for Plugins buttons. You should see the modules on the staging server, ready to be installed as a plugin.

The Future of Geronimo Plugins

By now, it should be clear that Geronimo plugins can be used to either

* Add new preconfigured application to a Geronimo instance
* Add new server features to a Geronimo instance

While the former is the most practical and useful aspect of the plugins infrastructure for Geronimo administrators, the latter is most exciting to Geronimo system designers. This aspect of flexible, dynamic server composition at the user's premises is one that the Geronimo team is looking to exploit in the near future.

Starting from the bare minimal server (e.g.,the Little-G minimal server distribution), you will soon be able to add only the features that you need by installing plugins.

For example, you may want to have Tomcat with Derby for your e-commerce application; or maybe you need Jetty with Derby and ActiveMQ; or maybe Tomcat with OpenEJB, but not Derby or ActiveMQ. This will be possible once these system components become available as plugins.

On the other side of the coin, since the Geronimo plugin infrastructure is both simple and well-documented, it is anticipated that it will open the door to both Open Source and commercial contributions. Any Open Source Web application can now be installed onto a Geronimo server with one-click ease. A large base of readily available plugins (on either the application or system level) will enhance the usability and appeal of the Geronimo server.

Summary

Plugins are a versatile feature of Geronimo 1.1 and beyond. Plugins are preconfigured, ready-to-run modules bundled in a convenient package.

The highly modular internals of the Geronimo server, coupled with the flexible dependency management provided by a Maven 2 repository, provide Geronimo with the ability to incrementally add features to a server via dynamically installed plugins. While this approach may not be reasonable for other fixed-purpose J2EE servers, it makes perfect sense for Geronimo, since J2EE is just one of the many personalities that a Geronimo server can facilitate.

Plugins are maintained in repositories. Repositories can be public or private. Repositories are typically Maven 2 repositories, but any installation of Geronimo can also act as a repository for plugin hosting purposes. It is possible to automatically and dynamically generate a list of installed system modules as plugins when Geronimo is used as a repository. This dynamic generation enables cloning of a Geronimo server configuration across a set of Geronimo server instances.

The Web console or the command-line deployer tool can be used to search for and install plugins. Once installed, plugins are indistinguishable from hand-configured and deployed modules. Plugins can be used to automate the tedious task of configuring and deploying applications. System module plugins can be used to incrementally add features to a minimally configured server.

It is often mentioned that the J2EE 1.4 compatible nature of the Geronimo server is just one personality that a Geronimo server can offer. Up until this point, it is not clear on just what the other personalities might be or how you can get at them. Chapter 18, “The Essence of Apache Geronimo: Flexible Assemblies,” shows how assemblies for Geronimo are constructed and how to make your own customized assemblies.