

18

The Essence of Apache Geronimo: Flexible Assemblies

Choice is one of the central premises of the Apache Geronimo design. The intention is to allow users to easily plug in familiar, well-known software components to create unique server instances. Plugging a component into Geronimo allows it to be managed as part of Geronimo's JSR-77 lifecycle events. The ability to plug in components in this manner is achieved through the microkernel style of design behind the Geronimo kernel, and the ease of implementation of GBeans. These two technologies are central to the Geronimo project, and are backed by a fair amount of history.

A Bit of Kernel History

In the beginning stages of Geronimo development, the kernel was the first item that was designed and developed. The kernel was built on top of Java Management Extensions (JMX), and MX4J was the JMX implementation that was used. The rationale for using JMX was that it was intended to manage and monitor components. To accomplish this purpose, JMX also requires that all instrumented components be implemented as JMX MBeans. The complexity of this decision soon bore itself out, because the use of the JMX MBean server as a general-purpose platform beyond simply managing components just would not scale. This is not to say that JMX is bad at all; it's simply a result of pushing the use of JMX further than its original intention. Developing software via reflective invocations rather than traditional method calls is certainly a different paradigm, and this proved to be very limiting. In addressing this and other issues, the kernel was completely redesigned. The following sections describe these decisions and the changes that were made to certain kernel services.

GBeans

The first component in the original JMX-based kernel that was replaced were the JMX `ModelMBeans` with a new component named a *GBean*. The concept of GBeans is that of an *Inversion of Control (IoC)* container. IoC, also known as *Dependency Injection (DI)*, seems to have become a popular concept in recent years because of the level of flexibility it provides. The general notion behind IoC is the separation of dependencies based on the configuration. Using IoC, no longer do components wire

together their own dependencies. Instead, a configuration tells the container what dependencies each component requires, and the container's job is to assemble these components and inject dependencies into each component at run-time. This is a tremendous improvement over an object that hard-codes all of its dependencies at compile time. Removing the hard-coded aspect from the design and development of software delivers an incredible amount of flexibility. It's the flexible assembly of components that caused the Geronimo developers to choose IoC for the redesign of the kernel. This means that the kernel doesn't require any knowledge of the technology being provided by the Geronimo server. This is all determined by the modules provided to the kernel, and the assembly created from the configuration by the kernel.

Proxy Manager

The Geronimo kernel began its life by using hard references to managed objects. Rather quickly, it became apparent that the use of hard references was a poor design decision for a few reasons, but one in particular stands out. When components depend upon one another, and hard references are used between them, getting those components to release those references is impossible if they're in use. For example, in a situation where a component needs to be shut down, but it is currently in use via a hard reference, that component cannot be shut down without also shutting down its ancestors in the chain first. The eventual result of this is that the entire collection of components that are wired together must be shut down. In other words, the entire server must be shut down simply to release one specific component. The solution to this issue was to use proxy objects.

A *proxy object* simply serves as an intermediary to a real object effectively becoming a soft reference. One advantage is that the shutdown of a component in the middle of a component chain can be easily achieved, even if that component is in use. In addition, the Geronimo proxy manager can also create remote proxy objects. This hides from the requesting client the fact that a remote call is being made to access a component, therefore saving the client from the complexity of this call.

Unfortunately, there are some disadvantages to using proxies as well. Proxies are very expensive in terms of performance, especially with regard to the startup lifecycle. Optional proxies are good, in general, because they offer a choice, but use of proxies during debug sessions can make stack traces much longer and debugging more difficult. So, as a convenience for developers, a property was added to the `AbstractGBeanReference` class named `Xorg.apache.geronimo.noproxy` that, when set, will disable the use of proxies. This is useful, mainly when developing and debugging Geronimo.

Dependency Manager

The job of the Geronimo *dependency manager* is to ensure that all registered dependencies are satisfied before creating a service. It is simply a record keeper of dependencies. As such, the dependency manager is very tightly tied to the kernel. The dependency manager does not enforce any dependencies; it is simply a place where components can register their intent to be dependent on another component.

Lifecycle Monitor

The *lifecycle monitor* was added to replace the JMX notifications. JMX notifications are difficult to use because they require a listener to register to see if any components are registered, and only then can a second listener register with the component to begin listening to the lifecycle events. This process is just like lacing and unlacing a shoe — the process to deregister must occur in exactly the opposite manner

to that of registration. What's truly odd, however, is the fact that the MBean server allows components to be unregistered without the use of any lifecycle whatsoever. This was another motivating factor to change how the Geronimo lifecycle operated. When there is no lifecycle mechanism surrounding the unregistration of components, then there is no ability to look into the unregistration process beyond the `RUNNING` state and the `STOPPED` states. So, if something goes wrong during that process, troubleshooting any issues is much more difficult.

Index and Query Engine

The *index and query engine* was created to replace the MBean query service. Because of this addition, there is no requirement to write an MBean. The kernel simply maintains a registry of deployed services with the ability to query that registry. The one requirement was that query parameters be `javax.management.ObjectName` objects. But more recently, `ObjectNames` have been largely removed and replaced by an `AbstractName`. An `AbstractName` is a combination of the Maven artifact ID and an arbitrary map that is built using JSR-77 rules. There are two benefits to this:

- * The JMX APIs are no longer required, and
- * The `AbstractNames` are much easier to debug.

Generally, you just look for the module and the attribute name in the map. There were also many convenience methods added as a result of this change that make querying much easier.

All of this is not to say that JMX is no longer a part of the Geronimo kernel. Manageability is still a central concern for the kernel, but creating MBeans to wrap components is no longer a requirement. JMX is simply injected into components at the point of assembly construction by the kernel based on a configuration, instead of being a cornerstone of the kernel design.

Given that these inefficiencies have been identified, the next question is sure to ask what is being done to address them? The next section discusses some solutions that are already underway for these issues.

The Future of the Kernel

In addition to the refactoring described earlier, there are other aspects of the kernel that still need some attention to provide more flexibility. Some of these items include the XML syntax used in deployment plans, the lack of choice in terms of lifecycle strategy and classloader management strategy, and the lack of a pluggable architecture in the kernel.

XML Syntax

The XML syntax in Geronimo deployment plans was created to address the need to provide a well-known means of configuration. Nearly anyone performing enterprise software development today has experience using XML, and it is supported by many software development tools. So it was determined that XML should be used as the configuration language. This choice was a good one, but little consideration was given to the XML dialect that should be used. In recent years, the Spring Framework has become extremely common because of its ability to make J2EE development easier. It offers many extremely powerful and useful features that many people now consider a requirement when doing

enterprise software development. One of the inherent features of Spring is its XML syntax. The syntax is very powerful without getting overly complex. The only issue with the syntax is that it needed to be extended a bit to make use of XML Schema, and to tie an XML Element name to a particular set of Plain Old Java Objects (POJOs).

Classloader and Lifecycle Strategies

The architecture of the kernel allows (and requires) the use of the default classloader and lifecycle management strategies. The trouble with restricting these two aspects of the kernel means that it is prohibitively difficult to use any other strategies. When there's a valid reason to use another strategy, the current state of the kernel doesn't make this task very easy. The current classloading strategy is not very flexible at all. The only manner by which to influence the classloading in Geronimo is to declare certain classes hidden from within a module's configuration, but this is only useful at the application level. In addition, the lifecycle provided by Geronimo is JSR-77 compliant, but does not allow for any deviation whatsoever at this time.

Overall Pluggability

The kernel architecture requires the use of the default IoC container implementation that is built into the GBean design. While this container was designed specifically for Geronimo, there are many valid reasons to use another one. There are many different IoC containers out there in the Open Source world, and each one offers its own unique twist on how dependency injection and its supporting concepts should work. One of those unique features includes lifecycle.

Each container offers its own flavor of object lifecycle management. Each one offers various advantages and disadvantages for different situations. The most widely known IoC container is the Spring Framework, but there's also HiveMind, PicoContainer, and Plexus, just to name a few, and there's reason to allow for the use of these containers if someone desires to do so.

Given all of these aspects of the kernel that are fairly rigid, an alternative kernel was begun. Originally, it was named GBean, but was later changed to XBean. Following is an outline of its capabilities.

Apache XBean

Some of the goals of XBean include building a plugin-based, server-side architecture, an easily customizable XML dialect that allows the use of XML Schema, the choice of what reference style to use in the dependency manager, and the choice of naming system. This idea encompasses the ability to plug in any IoC container that will allow the use of different classloader and lifecycle management techniques.

Pluggable Architecture

The pluggable architecture for the server-side is akin to what the Eclipse architecture provides in terms of pluggability for the client-side. The advantage is complete and utter flexibility at the cost of forgoing a direct default implementation. Indirectly, the default implementation that is plugged in is the Spring Framework. But it allows developers to implement whatever they so choose. By using XBean as the

Geronimo kernel, this means that Geronimo is completely pluggable. If, for some reason, there is a need to customize any part of the kernel, that capability is offered via the XBean API.

A Well-Known XML Dialect

In order to not reinvent the wheel, it was decided that XBean would build upon the Spring XML dialect because it handles configurations in a very elegant way. The XML it uses is fairly terse, but with the Spring container, it offers a significant amount of power. The Spring XML dialect is really great for developers, but can get somewhat complex for end users because it requires a knowledge of a codebase to be used. In a situation where a user simply wants to configure an object, most typically that user has no interest in digging into the code just to write the configuration. What was lacking in the Spring XML was the ability to make it more concise for end users, while at the same time offering improved validation through the use of XML Schema, but without losing any of the power for developers. In addition, it would be extremely helpful if the XML Schema was self-documenting. What was devised was an extension to the Spring XML dialect to offer exactly these features. Interestingly, however, is the fact that the Spring developers added support for the use of XML Schema in Spring 2.0.

XML Schema

XML Schema has become the de facto standard used to govern XML dialects in recent years, offering a much more advanced control over an XML instance. XML Schema offers many advances over DTDs, including the following:

- * XML Schema is written as XML, so it can be tested for validity and well-formedness.
- * XML Schema is much easier to extend.
- * XML Schema uses the highly powerful concept of namespaces.
- * XML Schema offers the functionality of include and import for the use of externally defined schema fragments.
- * XML Schema offers users the ability to define their own types using restriction or extension.
- * XML Schema provides constraints to restrict legal values of a given type.
- * XML Schema supports an explicit null value.

There are many more reasons that XML Schema is much more powerful than DTDs. This list is by no means comprehensive, but it illustrates some of the high points. One of the subprojects of XBean extended the Spring Framework to add support for XML Schema, a very powerful extension for sure. But XBean is much more than just an extension of Spring.

References

As explained previously, the required use of hard references in the original kernel was determined to be a poor design decision, and in place of this, proxies were used. However, requiring the use of proxies wasn't necessarily the right decision either, especially when flexibility is of the utmost importance. This is yet another place where XBean shines because it offers its own style of proxy service. The

architecture of the XBean proxy service allows the choice of either hard references or proxy objects, giving users a choice of which approach to use.

Naming System

The object indexing and query service in XBean is yet another place for flexibility. XBean doesn't require managed objects to use `javax.management.ObjectNames`. Instead, it supports not only `javax.management.ObjectNames`, but also flat strings, LDAP style names, UDDI names, and URIs. The idea is that multiple naming systems can be running concurrently. The only requirements are that names are unique, and that no two names collide.

All of this information about the kernel is important because it goes toward the goal of flexibility in Geronimo. In addition to all of this information about the kernel, other extremely vital portions of the Geronimo architecture and providing flexibility are the modules and assemblies.

Modules and Assemblies

Modules and assemblies are the lifeblood of Geronimo. These two concepts together are what make up the notion of a Geronimo server. The flexibility afforded by using these two concepts is unmatched in the enterprise server architecture realm. So, what exactly are modules and assemblies anyway? These concepts will be briefly explained, and then a real module and a real assembly will be analyzed to demonstrate each concept.

Modules

Remember that the kernel's job is to assemble components. It's the modules that allow the kernel to perform this task. A *module* in Geronimo is a set of components, dependencies, and an initial configured state, all bundled up together. Modules are a central theme in constructing the personality, so to speak, of any given Geronimo server. They determine how a given component is arranged at run-time, and what dependencies are required for a particular instance of a component.

The concept of modules in Geronimo as explained earlier refers to the actual deployable artifact for Geronimo. What is explained throughout the rest of this chapter is the process used by the Geronimo code base to create these modules. Just for clarity, modules are not required to be constructed using this exact process. What matters is that the deployable artifact that is produced contains the necessary items to be deployed to the Geronimo kernel. One additional point of clarification, however, is the fact that the core services in the Geronimo server are deployed as modules, not hard-coded into the kernel. Deploying the core Geronimo services as modules is the power in the assemblies. This will be explained in much more detail a bit later in the chapter.

Modules are described in a XML document that is referred to as a *deployment plan* (or just a *plan*, for short). A plan in Geronimo is made up of a combination of the initial XML file, the Maven Project Object Model (POM) file, and the Maven `project.properties` file. Figure 18-1 shows how these two files are processed to create the final deployment plan.

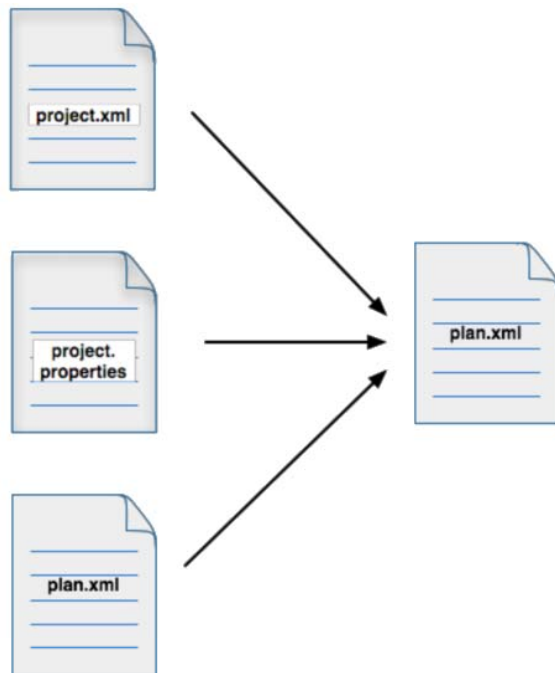


Figure 18-1: The initial deployment plan, the Maven POM file, and the Maven properties file are all processed to create the final deployment plan

Because the Maven POM file provides a mechanism for listing dependencies, rather than creating another custom mechanism for the same purpose, this mechanism was incorporated into the Geronimo build system for modules. Dependency artifacts (JARs, WARs, EARs, RARs, and so on) are listed in a Maven POM file, and the Geronimo Packaging Plugin for Maven is used to process all dependencies that are marked with a properties element. The Packaging Plugin recognizes the following properties:

- * `geronimo.dependency` — Marks the artifact as a dependency that must be listed in the final deployment plan when it's constructed by the packaging plugin.
- * `geronimo.import` — Marks the artifact as a parent in the final deployment plan when it's constructed by the packaging plugin.
- * `geronimo.include` — Marks the artifact as a dependency that must be listed in the final deployment plan when it's constructed, and it includes the artifact in the configuration archive (CAR file) that is created by the packaging plugin. (Additional information on CAR files is included later in this chapter.)

These properties are listed in a Maven POM file inside the properties element of a dependency element. Following is an example of this using the `geronimo.dependency` element:

```
<dependency>
  <groupId>geronimo</groupId>
  <artifactId>geronimo-directory</artifactId>
  <version>${directory_version}</version>
  <properties>
    <geronimo.dependency>true</geronimo.dependency>
```

```
</properties>  
</dependency>
```

Each JAR artifact has a unique name for the purpose of identification. The path to this dependency is pieced together using the child elements of the dependency element listed earlier. So, if `${directory-version}` is defined in a `project.properties` file as `0.9.2`, then the actual path to the dependency would appear as follows:

```
geronimo/jars/geronimo-directory-0.9.2.jar
```

This path is relative to the repository directory that is used by the Geronimo server for fetching dependencies to build up the CLASSPATH. (Additional information on the Geronimo repository is available later in this chapter.)

The `project.properties` file is simply a standard properties file that is recognized by Maven, and is used by Geronimo for defining properties used in the plan file. The only uniqueness is its name so that Maven will automatically pick it up and dereference any properties it contains so that they can be applied when the plan file is processed.

The preceding description explains how Geronimo creates plans for modules using Maven. This is not to say that Maven is a requirement for plan creation. The plans can be created using Ant, Java code, or a scripting language, just to name a few. The only requirement is that a properly defined plan file be provided to define the module.

What Belongs in a Plan?

As with most XML files, there is a finite amount of allowed content. The plan file itself is governed by an XML Schema Document (XSD) named `geronimo-module-1.1.xsd`. Because the entire XSD is fairly large in size, Figure 18-2 contains a logical view of the elements it contains, and Table 18-1 contains a description of these elements.

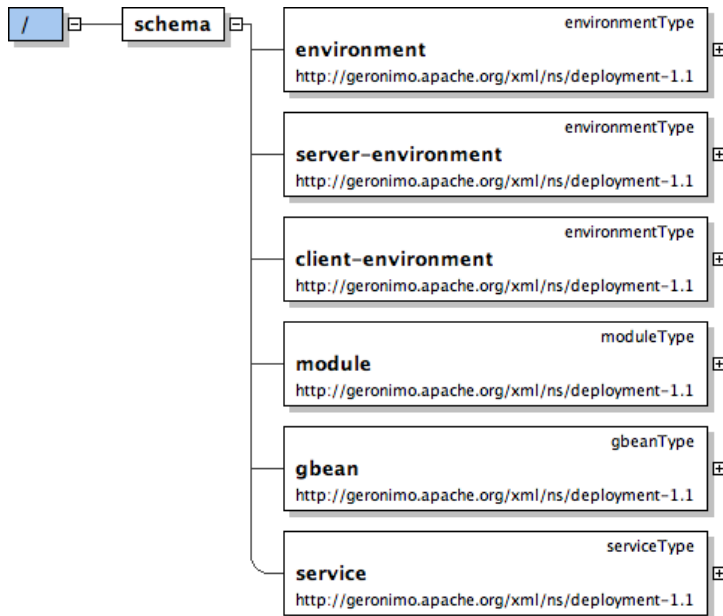


Figure 18-2: A high-level view of the `geronimo-module-1.0.xsd`

Table 18-1: Important Elements of `geronimo-module-1.0.xsd`

Element	Description
<code>module</code>	The parent container element for the definition of this module.
<code>environment</code>	A container element used to describe the environment for this module, including the following elements: <code>moduleId</code> , <code>dependencies</code> , <code>hidden-classes</code> , <code>non-overridable-classes</code> , <code>inverse-classloading</code> , and <code>suppress-default-environment</code> .
<code>server-environment</code>	A container element used to describe a server environment for this module, including the following elements: <code>moduleId</code> , <code>dependencies</code> , <code>hidden-classes</code> , <code>non-overridable-classes</code> , <code>inverse-classloading</code> , and <code>suppress-default-environment</code> .
<code>client-environment</code>	A container element used to describe an application client environment for this module, including the following elements: <code>moduleId</code> , <code>dependencies</code> , <code>hidden-classes</code> , <code>non-overridable-classes</code> , <code>inverse-classloading</code> , and <code>suppress-default-environment</code> .
<code>gbean</code>	Allows GBeans to be defined and/or referenced within this module description.

service	Used to describe any services that this module provides by default.
moduleId	Holds elements for the <code>groupId</code> , <code>artifactId</code> , and <code>version</code> of the module. <code>version</code> can be omitted, in which case a timestamp is used. These elements are flattened and separated by slashes to form the <code>moduleId</code> . For example, consider the following element structure defining a <code>moduleId</code> :
	<code><moduleId></code>
	<code><groupId>geronimo</groupId></code>
	<code><artifactId>directory</artifactId></code>
	<code><version>1.1.2-SNAPSHOT</version></code>
	<code><type>car</type></code>
	<code></moduleId></code>
	The unique name for this module would be flattened into the following string name:
	<code>geronimo/ directory /1.1.2-SNAPSHOT/car</code>
dependency	Holds all classloader and dependency information for the module, and looks just like the dependencies element in a Maven POM file. For example, the following is a single dependency:
	<code><dependency></code>
	<code><groupId>geronimo</groupId></code>
	<code><artifactId>j2ee-server</artifactId></code>
	<code><type>car</type></code>
	<code></dependency></code>
hidden-classes	A list of classes that will never be loaded from parent classloaders of this module. For example, if Log4J was listed here, the module would never see Geronimo's copy of Log4J. If the module provided its own Log4J JAR, it would use that; otherwise, it would not be able to load Log4J at all. The classes are specified in zero or more child filter elements where each filter element specifies a fully qualified class name or prefix. Essentially, any class that starts with one of the prefixes listed here will be treated as hidden. For example, if you specify two filter elements containing <code>'java.util'</code> and <code>'java.lang'</code> , then your application would not load correctly because any classes in those packages would be excluded.
non-overridable-classes	A list of classes that will only be loaded from parent

	classloaders of this module (never from the module's own classloader). For example, this is used to prevent a Web application from redefining <code>'javax.servlet'</code> , so those classes will <i>always</i> be loaded from the server instead of from the Web application's own <code>CLASSPATH</code> .
	The classes are specified in zero or more child filter elements, where each filter element specifies a fully qualified class name or prefix. Essentially, any class that starts with one of the prefixes listed here will be treated as hidden. For example, specifying two filter elements containing <code>'javax.servlet'</code> and <code>'javax.ejb'</code> would protect some of the core J2EE classes from being overridden.
<code>inverse-classloading</code>	If the <code>inverse-classloading</code> element is specified, the standard classloading delegation model is to be reversed for this module.
<code>suppress-default-environment</code>	If the <code>suppress-default-environment</code> element is specified, then any default environment build by a builder when deploying the plan will be suppressed.
	An example of where this is useful is when deploying a connector on an application client in a separate (standalone) module (not as part of a client plan).
	The connector builder default environment includes some server modules that won't work on an application client, so you need to suppress the default environment and supply a complete environment including all parents for a <code>non-application-client</code> module you want to run on an app client

Of the elements listed in Table 18-1, the `dependency` element refers to either another module running in the server, or an entry in the server's repository. In either case, this effectively uses a URI. When this is pointing to a repository entry, the URI must have a form acceptable to the repository, which is currently a URI of the form `group/type/name-version`.

When this is pointing to a module, the URI should match the module's `moduleId`. For standard Geronimo modules, this looks like a URI discussed earlier. However, user-deployed applications or modules can use arbitrary, simple URIs like `Foo` or `Bar`.

For example, two ways to map a URI are as follows:

- * Using a compact style that falls in a single line:

```
<uri>geronimo/jars/geronimo-directory-0.9.2.jar</uri>
```

- * Using multiple lines like Maven:

```
<groupId>geronimo</groupId>
```

```

<type>jar</type>
<artifactId>geronimo-directory</artifactId>
<version>0.9.2</version>

```

For more information and to see the entire schema, see the following URL for the actual XSD:

```

http://svn.apache.org/repos/asf/geronimo/server/branches/1.1/modules/service-
builder/src/schema/geronimo-module-1.1.xsd

```

This is all best demonstrated by looking at an actual module configuration. The following section examines the configuration used for the integration of the Apache Directory Server into Geronimo.

A Brief Overview of the Apache Directory Server Architecture

An example of a configuration is the one for the directory service provided by the Apache Directory project. The reason that this configuration is being used as the example here is that it's fairly easy to understand because it only contains one GBean.

The Apache Directory Server is an enterprise directory platform that is included as part of the Geronimo distribution. Figure 18-3 outlines the architecture of the Apache Directory Server.

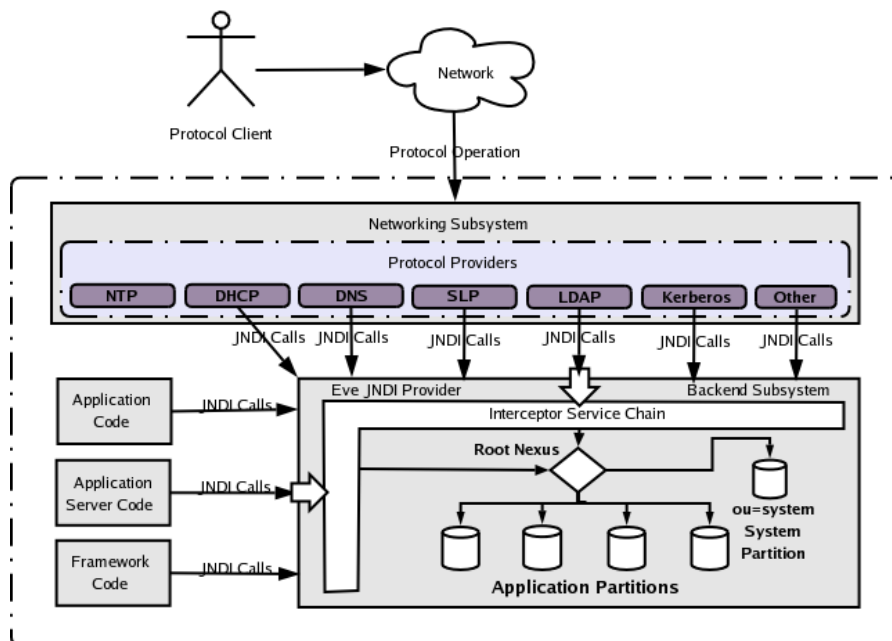


Figure 18-3: The Apache Directory Server architecture

The *Networking Subsystem* is provided by a subproject named the Multipurpose Infrastructure for Network Applications (MINA). MINA is a network application framework that wraps the networking layer by providing a high-level API that's easier to use than the low-level API provided for networking in the Java SE APIs. It provides such features as transport abstractions, filters above the transports, JMX management capabilities, traffic throttling, overload protection and much more.

The *Protocol Providers* are Java implementations of standard network services. As of this writing, available providers include LDAP, Kerberos, Change Password, DNS, NTP, and DHCP.

The *Backend Subsystem* provides an architecture based on the Staged Event Driven Architecture (SEDA) model. This subsystem acts as a layer above the schema and database subsystem.

For more information on the Apache Directory Server, see the project Web site (<http://directory.apache.org/>).

The Directory Server Configuration

To integrate the Apache Directory Server into Geronimo, a module configuration is required to tell the Geronimo kernel what to do. This configuration describes all the necessary components, and their dependencies for getting an instance of the directory server up and running within Geronimo. Listing 18-1 shows this configuration file:

Listing 18-1: The Apache Directory Server Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="http://geronimo.apache.org/xml/ns/deployment-1.1">
  <environment>
    <moduleId>
      <groupId>geronimo</groupId>
      <artifactId>directory</artifactId>
      <version>1.1.2-SNAPSHOT</version>
      <type>car</type>
    </moduleId>
    <dependencies>
      <dependency>
        <groupId>geronimo</groupId>
        <artifactId>j2ee-server</artifactId>
        <type>car</type>
      </dependency>
      <dependency>
        <groupId>geronimo</groupId>
        <artifactId>geronimo-directory</artifactId>
        <type>jar</type>
        <import>classes</import>
      </dependency>
    </dependencies>
    <hidden-classes/>
    <non-overridable-classes/>
  </environment>
  <gbean name="DirectoryService"
class="org.apache.geronimo.directory.DirectoryGBean">
    <attribute name="providerURL">ou=system</attribute>
    <attribute name="securityAuthentication">simple</attribute>
    <attribute name="securityPrincipal">uid=admin,ou=system</attribute>
    <attribute name="securityCredentials">secret</attribute>
    <attribute name="anonymousAccess">true</attribute>
    <attribute name="enableNetworking">true</attribute>
    <attribute name="host">0.0.0.0</attribute>
    <attribute name="port">1389</attribute>
    <reference name="ServerInfo">
      <name>ServerInfo</name>
```

```
</reference>
</gbean>
</module>
```

The unique name for this configuration is `geronimo/directory-1.1.2-SNAPSHOT/car`, and the parent for this configuration is the configuration named `j2ee-server` — that's the J2EE configuration in Geronimo (whose unique name is `geronimo/j2ee-server-1.0/car`). In other words, this configuration needs the J2EE server configuration to be available before it can be loaded. Beyond these items, this configuration contains all the necessary state information to be injected into the `DirectoryGBean` when the kernel instantiates it at run-time. But what are CAR files?

CAR Files

CAR is an acronym for Configuration ARchive that is a JAR file containing the serialized state of a configuration and any other supporting files and metadata. CAR files are created automatically by the build process via the Geronimo Packaging Plugin for Maven —they're not something that a developer creates by hand. The thought behind using a serialized file is that it can be reloaded much faster and easier than going through the parse and construct phases for all modules each time a server is started. The contents of a typical CAR file are described in Table 18-2.

Table 18-2: The Contents of a CAR File

Item	Required	Location
Serialized file	Yes	<code>META-INF/config.ser</code>
Documentation	No	Arbitrary
Dependencies	No	Arbitrary*

* The standard location of dependencies is in a directory whose name represents the artifact type

Following is a listing of the contents of the `directory-1.1.2-SNAPSHOT.car` file:

```
META-INF/config.info
META-INF/config.ser
META-INF/config.ser.sha1
```

The `config.ser` file listed above is the main content of the CAR file. The `config.info` file contains the module ID and some additional metadata, and the `config.ser.sha1` is simply a digital signature for the `config.ser` file. The `config.ser` file actually *is* the module in a serialized form. This is the file that gets loaded into the Geronimo server and deserialized to run the module. The advantage of using a serialized file to house a module is that it can be loaded into the Geronimo server very quickly.

Unfortunately, serialized files are not easy to work with from an administrative point of view. Serialized files are not easily inspected, not easily modified, and not very portable. The use of serialized files to house modules is being analyzed, and may change to a more user-friendly format in the future.

CAR files are specific to Geronimo. Beyond being used as reference points between modules, CAR files are also used inside of assemblies as reference points. But what are assemblies and how do they make use of modules?

Assemblies

A *module* describes a particular component. But any single module typically does not describe enough components to turn Geronimo into, for example, a J2EE application server. This is where assemblies come in. *Assemblies* are comprised of collections of modules. Assemblies are what define the server, and the types of applications it can support. They are a gathering point to describe everything that should be included in a particular instance of a Geronimo server.

Figure 18-4 shows how the assemblies and modules in Geronimo relate to one another. Notice that the J2EE Jetty Assembly and the J2EE Tomcat Assembly overlap. This is to illustrate that these assemblies share some modules. The dashed lines represent logical groupings, not physical ones. Each configuration is composed of GBeans, and their metadata and assemblies are composed of modules. The kernel services at the bottom of the diagram help to manage the modules and the repository, and the configuration manager provides additional services, but they are not directly a part of the kernel.

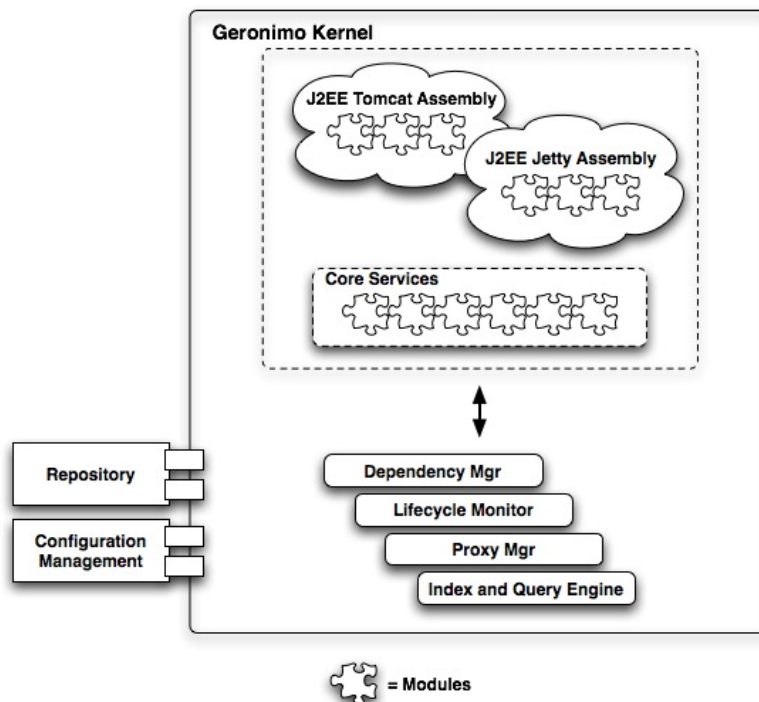


Figure 18-4: Assemblies and modules

For example, the module configuration for the directory module back in Listing 18-1 tells the kernel most of what it needs to know, but a bit more information is needed. Assemblies are packaged using the Geronimo Assembly Plugin for Maven. This is a plugin for Maven that handles the creation and installation of assemblies from a combination of the modules, the `project.xml` file (the POM), and the `project.properties` file. Within each dependency in the POM that needs to be included in a module configuration, a Geronimo-specific property must be listed. Similar to the properties supported

by the Geronimo Packaging Plugin for Maven as described previously, the Geronimo Assembly Plugin for Maven also supports some properties. Following is a list of those supported properties:

- * `geronimo.assemble` — This property tells the Geronimo Assembly Plugin how to handle the dependency. Following are the available values for this variable:
 - * `endorsed` — Place the dependency in the `lib/endorsed` directory.
 - * `extension` — Place the dependency in the `lib/ext` directory.
 - * `install` — Place the CAR file in `config-store` and copy all of its dependencies into the Geronimo repository.
 - * `library` — Place the dependency in the `lib` directory.
 - * `repository` — Place the artifact in the Geronimo repository.
 - * `unpack` — Expands the artifact in the server that is being assembled. This is used for XSDs, documentation, and the like.
- * `geronimo.assemble.executable` — This property tells the Geronimo Assembly Plugin to enable the executable bit on the dependency. This is used for `deployer.jar` and `server.jar`.

The Geronimo Assembly Plugin parses this property, and then each dependency marked with this property is copied into the proper location, most commonly into the Geronimo repository. Assemblies locate dependencies most often in the Geronimo repository. With all of this talk about the Geronimo repository, now is a fine time to explain what it is.

The Geronimo Repository

The default Geronimo repository is simply a directory hierarchy on the file system. In the binary distribution of Geronimo, there is a directory named `repository` that hold all the dependencies for all modules that are included in the given assembly. Listing 18-2 is just a slice of this directory hierarchy to demonstrate what it holds and how it's structured.

Listing 18-2: A Directory Listing of the Default Repository

```
+---repository
...
| +---directory
| | +---jars
| | | +---apacheds-core-0.9.2.jar
| | | +---apacheds-shared-0.9.2.jar
| | +---directory-asn1
| | | +---jars
| | | | +---asn1-ber-0.3.2.jar
| | | | +---asn1-codec-0.3.2.jar
| | | | +---asn1-der-0.3.2.jar
| | +---directory-network
| | | +---jars
| | | | +---mina-0.7.3.jar
| | +---directory-protocols
| | | +---jars
| | | | +---kerberos-protocol-0.5.jar
| | | | +---ldap-protocol-0.9.2.jar
```

```

+---directory-shared
|
| +---jars
| | +---apache-ldapber-provider-0.9.2.jar
| | +---kerberos-common-0.5.jar
| | +---ldap-common-0.9.2.jar
+---geronimo
|
| +---cars
| | +---activemq-1.0-SNAPSHOT.car
| | +---activemq-broker-1.0-SNAPSHOT.car
| | +---directory-1.0-SNAPSHOT.car
| | +---geronimo-gbean-deployer-1.0-SNAPSHOT.car
| | +---j2ee-deployer-1.0-SNAPSHOT.car
| | +---j2ee-security-1.0-SNAPSHOT.car
| | +---j2ee-server-1.0-SNAPSHOT.car
| | +---j2ee-system-1.0-SNAPSHOT.car
| | +---jetty-1.0-SNAPSHOT.car
| | +---jetty-deployer-1.0-SNAPSHOT.car
| | +---rmi-naming-1.0-SNAPSHOT.car
| | +---system-database-1.0-SNAPSHOT.car
| | +---uddi-jetty-1.0-SNAPSHOT.car
| | +---webconsole-jetty-1.0-SNAPSHOT.car
| | +---welcome-jetty-1.0-SNAPSHOT.car
|
| +---jars
| | +---geronimo-activation-1.0-SNAPSHOT.jar
| | +---geronimo-axis-1.0-SNAPSHOT.jar
| | +---geronimo-axis-builder-1.0-SNAPSHOT.jar
| | +---geronimo-client-1.0-SNAPSHOT.jar
| | +---geronimo-client-builder-1.0-SNAPSHOT.jar
| | +---geronimo-common-1.0-SNAPSHOT.jar
| | +---geronimo-connector-1.0-SNAPSHOT.jar
| | +---geronimo-connector-builder-1.0-SNAPSHOT.jar
| | +---geronimo-console-core-1.0-SNAPSHOT.jar
| | +---geronimo-core-1.0-SNAPSHOT.jar
| | +---geronimo-deploy-tool-1.0-SNAPSHOT.jar
| | +---geronimo-deployment-1.0-SNAPSHOT.jar
| | +---geronimo-derby-1.0-SNAPSHOT.jar
| | +---geronimo-directory-1.0-SNAPSHOT.jar
| | +---geronimo-j2ee-1.0-SNAPSHOT.jar
| | +---geronimo-j2ee-builder-1.0-SNAPSHOT.jar
| | +---geronimo-j2ee-schema-1.0-SNAPSHOT.jar
| | +---geronimo-jetty-1.0-SNAPSHOT.jar
| | +---geronimo-jetty-builder-1.0-SNAPSHOT.jar
| | +---geronimo-jmxremoting-1.0-SNAPSHOT.jar
| | +---geronimo-kernel-1.0-SNAPSHOT.jar
| | +---geronimo-management-1.0-SNAPSHOT.jar
|
| ...

```

Listing 18-2 is a small slice of the listing of the directory tree for the default repository. This is nothing more than a directory hierarchy. The unique aspect to this hierarchy is the way it's constructed. Inside the top-level directory named `repository` is a collection of directories, one for each dependency group. Inside a group directory are directories representing each artifact type (for example, EARs, JARs, WARs, and so on). In each type directory are all the artifacts that belong to that group type. Be aware that any given group type directory may contain more than one version of a particular artifact. Figure 18-5 explains this concept graphically.

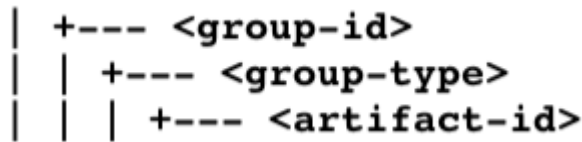


Figure 18-5: The structure of the default repository

To explain Figure 18-2, let's apply pieces of the directory listing in Listing 18-2. Notice that all the directories are bold, and there is a directory named `directory`. This directory (whose name happens to be `directory`) is the `group-id`, and it corresponds to the Apache Directory Server and the artifacts. Now notice that inside the directory whose name is `directory` is another directory named `jars`. The `jars` directory is the `group-type`. Inside this `jars` directory resides all the dependencies from the Apache Directory Server project (whose name is `directory`) and whose type is JAR.

The ideas behind the Geronimo repository were heavily borrowed from the Apache Maven project and its concepts of a repository for dependencies. Any knowledge of the Maven repository transfers easily into understanding the Geronimo repository.

The repository is not limited to a directory structure on the local filesystem. The repository is actually pluggable, so other implementations could be created. But none have come about from the community so far.

Adding Artifacts to the Repository

There are two ways to add artifacts (for example, EARs, JARs, WARs, RARs, and so on) to the Geronimo repository by marking them in the Maven POM, and by adding them via the Web console. Each one is used for a specific purpose, and each one has its own advantages and disadvantages.

The first and most common method is via a Maven POM. Each dependency that needs to be added to the repository can be marked with a property named `geronimo.assemble`, whose value is `repository`. Following is an example of a dependency in a Maven POM file that is marked using this property, and the `repository` value so it will be added to the repository:

```

<dependency>
  <groupId>activemq</groupId>
  <artifactId>activemq-ra</artifactId>
  <version>${activemq_version}</version>
  <type>rar</type>
  <properties>
    <geronimo.assemble>repository</geronimo.assemble>
  </properties>
</dependency>

```

The purpose of this method is to add the artifact to the repository when the assembly is compiled from source. The Geronimo Assembly Plugin for Maven handles this task automatically during the build process. Because the Maven build was already downloading the artifacts from remote Maven repositories, it was an easy decision that a Maven plugin should handle the movement of the artifact inside of the Geronimo server during build time as well.

If a custom assembly is being created, then using the `geronimo.assemble` property with the `repository` value is the best method. This is because creating a custom assembly means that code is already being written, and it's easier to mark needed artifacts to be copied at build time.

The second method is via the Web console. To get to the Web console, simply log in to the console using a Web browser. The default username and password for the Web console is `manager` and `system`. Upon logging in to the console, under the Console Navigation heading, click on the Common Libraries link to display the Repository Viewer as shown in Figure 18-6.

Repository Viewer [help \[view\]](#)

Add Archive to Repository

File:

Group:

Artifact:

Version:

Type:

Current Repository Entries

- activecluster/activecluster/1.2-20051115174934/jar
- activeio/activeio/2.0-r118/jar
- activemq/activemq-core/3.2.4/jar
- activemq/activemq-gbean-g1_1/3.2.4/jar
- activemq/activemq-gbean-management-g1_1/3.2.4/jar
- antlr/antlr/2.7.2/jar
- asm/asm/1.4.3/jar
- axis/axis/1.4/jar
- cglib/cglib-nodep/2.1_3/jar
- commons-cli/commons-cli/1.0/jar
- commons-discovery/commons-discovery/0.2/jar
- commons-el/commons-el/1.0/jar
- commons-logging/commons-logging/1.0.4/jar
- concurrent/concurrent/1.3.4/jar
- geronimo-spec/geronimo-spec-corba/1.0/jar
- qeronimo/activemq-broker/1.1.2-SNAPSHOT/car

Figure 18-6: The Web console can be used to add artifacts to the Geronimo repository

When using the Repository Viewer in Figure 18-6, artifacts are instantly loaded into the Geronimo repository and listed on the page. This tool is very useful when working with an existing Geronimo server because it's simply copying the artifacts into place. Once the artifact is part of the repository, it can be used anywhere within the server.

It's fairly well-known now that modules make a component a part of the Geronimo lifecycle, and that collections of modules are what make up an assembly. The next task is to take a detailed look at an existing assembly.

Breaking Down the J2EE Assembly

Here's a statement that shocks some people: The Geronimo kernel has absolutely no knowledge of J2EE whatsoever. The kernel simply deploys the assemblies it is provided. One such example assembly is a J2EE assembly. After all, it's a J2EE assembly that allowed Geronimo to achieve J2EE 1.4

certification by pulling together all the necessary software that comprises a J2EE server. So, what software is used to achieve this goal? Table 18-3 outlines this software.

Table 18-3: The Software Implementations that Provide J2EE 1.4

Item	Implementation
JSP and Servlets	Jetty, Apache Tomcat
EJB	OpenEJB
JMS	ActiveMQ
JAX-RPC, JAXR, WS-I Basic, SAAJ	Apache Axis, Apache Scout
JavaMail	Geronimo
JAF	Geronimo
JNDI	Geronimo
JTA	Geronimo
JAAS, JACC	Geronimo
Management/JSR-77	Geronimo, MX4J
Deployment/JSR-88	Geronimo
JMX	MX4J
JCA	Geronimo, TranQL
CORBA, IDL, IIOP	Geronimo and J2SE 1.4
Timers	Geronimo
WorkManager	Geronimo
Journaling	HOWL
Embedded Database	Apache Derby
Directory	Apache Directory

Table 18-3 contains a list of the software modules included in Geronimo and notes the project from which that software originates. The J2EE assembly is the first application server personality, so to speak, that Geronimo was given. This assembly is comprised using all the software listed in Table 18-3, and more. While it may seem like the list includes a high amount of custom code written specifically for Geronimo, this is a false conclusion to draw. From a lines-of-code count, the size of the custom Geronimo implementations pales in comparison to the projects included. What are not listed in that table are all the transitive dependencies. Transitive dependencies are explained in Figure 18-7.

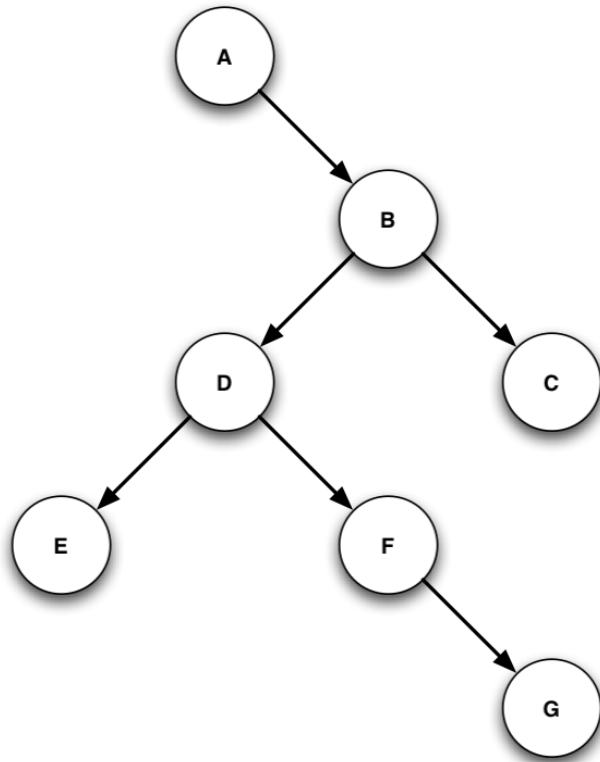


Figure 18-7: Transitive dependencies

Figure 18-7 depicts a graph of pieces of software where each circle represents a unique piece of software. In this image, A depends on B and B depends on C. But B also depends on D and E. And D depends on F and G. This is only a very small example of how large such a graph can grow. For a project the size of Geronimo, at first glance, the amount of dependencies can be surprising. But considering what Geronimo offers, it makes sense. None of these dependencies are actually included in the assembly. Only references to these dependencies are included.

The implementations in Table 18-3 to provide J2EE are certainly not limited to that list. Another one of the original goals of Geronimo was to provide an Open Source J2EE 1.4 certified implementation that uses a BSD-derived license. This is only a goal for the development of Geronimo; it is not a requirement that anything plugged into Geronimo be Open Source. Any software component can be plugged in to Geronimo to provide an implementation of one of the J2EE specs. For example, if a user wanted to plug in a different Web services implementation to Geronimo, that user has the ability to do that by creating a custom assembly. But why would someone want to create a custom assembly?

Examining an Assembly

Geronimo offers two flavors of Web containers in its assemblies: one that offers Jetty and one that offers Apache Tomcat. This is obviously to satisfy both Web development communities out there. The rest of the chapter will focus on digging into and understanding the J2EE assembly that includes the Tomcat Web container.

Examining the Module Dependencies

To achieve a more complete understanding of assemblies, the dependency hierarchy must be fully examined. The Geronimo kernel works by defining a dependency hierarchy of GBean objects in a module deployment plan. By tracing these dependencies in this hierarchy, the necessary components can be identified and removed. In addition, there are other modules that must be added to take the place of some required components. This will be explained a bit later.

The ultimate parent module within Geronimo is the `j2ee-server` module. This module can be found in the `configs` directory of the Geronimo source code and is shown in Listing 18-3.

Listing 18-3: The `j2ee-server` Configuration Plan

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="http://geronimo.apache.org/xml/ns/deployment-1.1">
  <environment>
    <moduleId>
      <groupId>geronimo</groupId>
      <artifactId>j2ee-server</artifactId>
      <version>1.1.2-SNAPSHOT</version>
      <type>car</type>
    </moduleId>
    <dependencies>
      <dependency>
        <groupId>geronimo</groupId>
        <artifactId>rmi-naming</artifactId>
        <type>car</type>
      </dependency>
      <dependency>
        <groupId>geronimo</groupId>
        <artifactId>geronimo-core</artifactId>
        <type>jar</type>
        <import>classes</import>
      </dependency>
      <dependency>
        <groupId>geronimo</groupId>
        <artifactId>geronimo-common</artifactId>
        <type>jar</type>
        <import>classes</import>
      </dependency>
      <dependency>
        <groupId>geronimo</groupId>
        <artifactId>geronimo-connector</artifactId>
        <type>jar</type>
        <import>classes</import>
      </dependency>
      <dependency>
        <groupId>geronimo</groupId>
        <artifactId>geronimo-webservices</artifactId>
        <type>jar</type>
        <import>classes</import>
      </dependency>
      <dependency>
        <groupId>asm</groupId>
        <artifactId>asm</artifactId>
        <type>jar</type>
      </dependency>
    </dependencies>
  </environment>
</module>
```

```

    <import>classes</import>
  </dependency>
  <dependency>
    <groupId>activeio</groupId>
    <artifactId>activeio</artifactId>
    <type>jar</type>
    <import>classes</import>
  </dependency>
</dependencies>
<hidden-classes/>
<non-overridable-classes/>
</environment>
  <gbean name="ConnectionTracker"
class="org.apache.geronimo.connector.outbound.connectiontracking.ConnectionTracking
CoordinatorGBean"></gbean>
  <gbean name="ConnectorThreadPool" class="org.apache.geronimo.pool.ThreadPool">
    <attribute name="keepAliveTime">5000</attribute>
    <attribute name="poolSize">30</attribute>
    <attribute name="poolName">ConnectorThreadPool</attribute>
  </gbean>
  <gbean name="DefaultWorkManager"
class="org.apache.geronimo.connector.work.GeronimoWorkManagerGBean">
    <reference name="SyncPool">
      <name>ConnectorThreadPool</name>
    </reference>
    <reference name="StartPool">
      <name>ConnectorThreadPool</name>
    </reference>
    <reference name="ScheduledPool">
      <name>ConnectorThreadPool</name>
    </reference>
    <reference name="TransactionContextManager">
      <name>TransactionContextManager</name>
    </reference>
  </gbean>
  <gbean name="HOWLTransactionLog"
class="org.apache.geronimo.transaction.log.HOWLLog">
    <attribute
name="bufferClassName">org.objectweb.howl.log.BlockLogBuffer</attribute>
    <attribute name="bufferSizeKBytes">32</attribute>
    <attribute name="checksumEnabled">>true</attribute>
    <attribute name="flushSleepTimeMilliseconds">50</attribute>
    <attribute name="logFileDir">var/txlog</attribute>
    <attribute name="logFileExt">log</attribute>
    <attribute name="logFileName">howl</attribute>
    <attribute name="maxBlocksPerFile">-1</attribute>
    <attribute name="maxBuffers">0</attribute>
    <attribute name="maxLogFiles">2</attribute>
    <attribute name="minBuffers">4</attribute>
    <attribute name="threadsWaitingForceThreshold">-1</attribute>
    <reference name="XidFactory">
      <name>XidFactory</name>
    </reference>
    <reference name="ServerInfo">
      <name>ServerInfo</name>
    </reference>
  </gbean>

```

```

    <gbean name="XidFactory"
class="org.apache.geronimo.transaction.manager.XidFactoryImplGBean">
    <attribute name="tmId">71,84,77,73,68</attribute>
    </gbean>
    <gbean name="TransactionManager"
class="org.apache.geronimo.transaction.manager.TransactionManagerImplGBean">
    <attribute name="defaultTransactionTimeoutSeconds">600</attribute>
    <reference name="XidFactory">
        <name>XidFactory</name>
    </reference>
    <reference name="TransactionLog">
        <name>HOWLTransactionLog</name>
    </reference>
    <references name="ResourceManagers">
        <pattern>
            <type>JCAManagedConnectionFactory</type>
        </pattern>
        <pattern>
            <type>ActivationSpec</type>
        </pattern>
    </references>
    </gbean>
    <gbean name="TransactionContextManager"
class="org.apache.geronimo.transaction.context.TransactionContextManagerGBean">
    <reference name="TransactionManager">
        <name>TransactionManager</name>
    </reference>
    <reference name="XidImporter">
        <name>TransactionManager</name>
    </reference>
    </gbean>
    <!--JSR77 Management Objects-->
    <gbean name="geronimo.server"
class="org.apache.geronimo.j2ee.management.impl.J2EEDomainImpl">
    <reference name="Servers"/>
    </gbean>
    <gbean name="geronimo"
class="org.apache.geronimo.j2ee.management.impl.J2EEServerImpl">
    <reference name="ServerInfo">
        <name>ServerInfo</name>
    </reference>
    <reference name="JVMs"/>
    <references name="Resources">
        <pattern>
            <type>JCAResource</type>
        </pattern>
        <pattern>
            <type>JavaMailResource</type>
        </pattern>
        <pattern>
            <type>JDBCResource</type>
        </pattern>
        <pattern>
            <type>JMSResource</type>
        </pattern>
        <pattern>
            <type>JNDIResource</type>
        </pattern>
    </references>
    </gbean>

```

```

    <pattern>
      <type>JTAResource</type>
    </pattern>
  </pattern>
  <pattern>
    <type>RMI_IIOPResource</type>
  </pattern>
  <pattern>
    <type>URLResource</type>
  </pattern>
</references>
<reference name="Applications"/>
<reference name="AppClientModules"/>
<reference name="WebModules"/>
<reference name="EJBModules"/>
<reference name="ResourceAdapterModules"/>
<reference name="WebManagers"/>
<reference name="EJBManagers"/>
<reference name="JMSManagers"/>
<reference name="ThreadPools"/>
<reference name="Repositories"/>
<reference name="WritableRepos"/>
<reference name="SecurityRealms"/>
<reference name="LoginServices"/>
<reference name="KeystoreManagers"/>
<reference name="PluginRepoLists"/>
<reference name="PluginInstaller">
  <name>PluginInstaller</name>
</reference>
<reference name="ConfigurationManager">
  <name>ConfigurationManager</name>
</reference>
</gbean>
<gbean name="JVM" class="org.apache.geronimo.j2ee.management.impl.JVMImpl">
  <reference name="SystemLog">
    <name>Logger</name>
  </reference>
</gbean>
</module>

```

As explained earlier in this chapter, the final `plan.xml` file is the heart of a configuration, and this is created from a combination of the initial `plan.xml` file, as well as the `project.properties` and `project.xml` files from Maven. Let's begin by examining the initial `plan.xml` file. This plan is described in Table 18-4.

Table 18-4: The j2ee-server Configuration

GBean Name	Description
ConnectionTracker	For tracking outbound J2EE Connector Architecture (JCA) connections.
ConnectorThreadPool	A GBean for managing a pool of threads.
DefaultWorkManager	A JCA WorkManager GBean to allow work to be scheduled synchronously or asynchronously on a thread from the thread pool usually within a

	transaction.
HOWLTransactionLog	For the journaling log for the Highspeed ObjectWeb Logger (HOWL).
XidFactory	A factory for generating transaction IDs.
TransactionManager	Wraps the Geronimo transaction manager.
TransactionContextManager	Provides transaction related event management.
geronimo.server	A GBean that provides JSR-77 information for the management of the core services in Geronimo.
geronimo	Provides JSR-77 information for statistics on the current JVM.
JVM	Provides metadata and stats.

The `J2EEServer` GBean depends on the `ServerInfo` object, but notice that the dependency uses a reference. The reference element refers to a reference to another GBean that's registered with the kernel. This means that the `ServerInfo` object must be started before the `J2EEServer` object. The other GBeans defined in this plan don't define a direct dependency on the `J2EEServer` GBean, but without it, the `ServerInfo` would not be present, and this object is very important. The `ServerInfo` object provides basic metadata about Geronimo and methods for resolving pathnames, and is defined in the `j2ee-system` configuration shown in Listing 18-4.

Listing 18-4: The `j2ee-system` Configuration Plan

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="http://geronimo.apache.org/xml/ns/deployment-1.1">
  <!--ServerInfo service-->
  <environment>
    <moduleId>
      <groupId>geronimo</groupId>
      <artifactId>j2ee-system</artifactId>
      <version>1.1.2-SNAPSHOT</version>
      <type>car</type>
    </moduleId>
    <dependencies>
      <dependency>
        <groupId>geronimo</groupId>
        <artifactId>geronimo-common</artifactId>
        <type>jar</type>
        <import>classes</import>
      </dependency>
      <dependency>
        <groupId>geronimo</groupId>
        <artifactId>geronimo-kernel</artifactId>
        <type>jar</type>
        <import>classes</import>
      </dependency>
      <dependency>
        <groupId>geronimo</groupId>
        <artifactId>geronimo-system</artifactId>
        <type>jar</type>
        <import>classes</import>
      </dependency>
    </dependencies>
  </environment>
</module>
```

```
</dependency>
<dependency>
  <groupId>cglib</groupId>
  <artifactId>cglib-nodep</artifactId>
  <type>jar</type>
  <import>classes</import>
</dependency>
<dependency>
  <groupId>concurrent</groupId>
  <artifactId>concurrent</artifactId>
  <type>jar</type>
  <import>classes</import>
</dependency>
<dependency>
  <groupId>commons-logging</groupId>
  <artifactId>commons-logging</artifactId>
  <type>jar</type>
  <import>classes</import>
</dependency>
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <type>jar</type>
  <import>classes</import>
</dependency>
<dependency>
  <groupId>mx4j</groupId>
  <artifactId>mx4j</artifactId>
  <type>jar</type>
  <import>classes</import>
</dependency>
<dependency>
  <groupId>mx4j</groupId>
  <artifactId>mx4j-remote</artifactId>
  <type>jar</type>
  <import>classes</import>
</dependency>
<dependency>
  <groupId>org.apache.geronimo.specs</groupId>
  <artifactId>geronimo-qname_1.1_spec</artifactId>
  <type>jar</type>
  <import>classes</import>
</dependency>
<dependency>
  <groupId>xerces</groupId>
  <artifactId>xercesImpl</artifactId>
  <type>jar</type>
  <import>classes</import>
</dependency>
<dependency>
  <groupId>xerces</groupId>
  <artifactId>xmlParserAPIs</artifactId>
  <type>jar</type>
  <import>classes</import>
</dependency>
<dependency>
  <groupId>xstream</groupId>
  <artifactId>xstream</artifactId>
```

```

        <type>jar</type>
        <import>classes</import>
    </dependency>
</dependency>
<dependency>
    <groupId>xpp3</groupId>
    <artifactId>xpp3</artifactId>
    <type>jar</type>
    <import>classes</import>
</dependency>
</dependencies>
<hidden-classes/>
<non-overridable-classes/>
</environment>
<gbean name="ServerInfo"
class="org.apache.geronimo.system.serverinfo.BasicServerInfo"/>
<!--Repository-->
<gbean name="Repository"
class="org.apache.geronimo.system.repository.Maven2Repository">
    <attribute name="root">repository</attribute>
    <reference name="ServerInfo">
        <name>ServerInfo</name>
    </reference>
</gbean>
<!--Configuration Store service-->
<gbean name="Local "
class="org.apache.geronimo.system.configuration.RepositoryConfigurationStore">
    <reference name="Repository">
        <name>Repository</name>
    </reference>
</gbean>
<!--User-editable attribute service-->
<gbean name="AttributeManager"
class="org.apache.geronimo.system.configuration.LocalAttributeManager">
    <reference name="ServerInfo">
        <name>ServerInfo</name>
    </reference>
    <attribute name="configFile">var/config/config.xml</attribute>
</gbean>
<!--ArtifactManager-->
<gbean name="ArtifactManager"
class="org.apache.geronimo.kernel.repository.DefaultArtifactManager"/>
<!--ArtifactResolver-->
<gbean name="ArtifactResolver"
class="org.apache.geronimo.kernel.repository.DefaultArtifactResolver">
    <reference name="ArtifactManager">
        <name>ArtifactManager</name>
    </reference>
    <reference name="Repositories">
        <!--<gbean-name>*:name=Repository,*</gbean-name-->
    </reference>
</gbean>
<!--Configuration Manager service-->
<gbean name="ConfigurationManager"
class="org.apache.geronimo.kernel.config.EditableKernelConfigurationManager">
    <reference name="Repositories"></reference>
    <reference name="Stores"></reference>
    <reference name="Watchers"></reference>
    <reference name="AttributeStore">

```

```

    <name>AttributeManager</name>
  </reference>
  <reference name="PersistentConfigurationList">
    <type>AttributeStore</type>
    <name>AttributeManager</name>
  </reference>
  <reference name="ArtifactManager">
    <name>ArtifactManager</name>
  </reference>
  <reference name="ArtifactResolver">
    <name>ArtifactResolver</name>
  </reference>
</gbean>
<!--Logging service-->
<gbean name="Logger"
class="org.apache.geronimo.system.logging.log4j.Log4jService">
  <attribute name="configFileName">var/log/server-log4j.properties</attribute>
  <attribute name="refreshPeriodSeconds">60</attribute>
  <reference name="ServerInfo">
    <name>ServerInfo</name>
  </reference>
</gbean>
</module>

```

This configuration contains additional GBean definitions that provide key infrastructure services for Geronimo as outlined in Table 18-5.

Table 18-5: The j2ee-system Configuration

GBean Name	Description
ServerInfo	Provides basic metadata about Geronimo and methods for resolving pathnames.
Repository	This GBean wraps the default implementation of the Geronimo repository.
Local	A GBean for handling manageable attributes in GBeans.
AttributeManager	A GBean for handling manageable attributes in GBeans.
ArtifactManager	Tracks all artifacts loaded by Geronimo.
ArtifactResolver	Provides the ability to query and resolve artifacts loaded by Geronimo.
ConfigurationManager	Manages all modules loaded into Geronimo.
Logger	A GBean for the logging service provided by Log4J.

The `j2ee-server` configuration in Listing 18-3 and `j2ee-system` configuration in Listing 18-4 are the most important modules in all of Geronimo. These modules are like the engine in a car. If the engine doesn't start, the car will not be functional. They provide a base for all other modules. Without these, the server would be unable to function properly because the GBeans defined in these two modules wrap

some of the very core features in the Geronimo kernel and the services it provides. All J2EE assemblies depend upon these two modules.

Continuing to trace downward through the rest of the modules and the entire dependency hierarchy is certainly one way to discover all the dependencies, but this physical approach would take quite a while. A more logical approach would be to pick a well-known component and begin to examine its configuration.

The Geronimo Configuration for Apache Tomcat

One of the most familiar services integrated with Geronimo is Apache Tomcat. Tomcat is the reference implementation for the Servlet/JSP spec, it is used by developers the world over. The Tomcat integration with Geronimo is comprised of two modules: one named `tomcat` and the other named `tomcat-deployer`. The `tomcat` configuration is for all the services offered by Tomcat, and the `tomcat-deployer` is for deploying modules in Tomcat. These modules are shown below in Listings 18-5 and 18-6.

Listing 18-5: The `tomcat` Plan

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="http://geronimo.apache.org/xml/ns/deployment-1.1">
  <environment>
    <moduleId>
      <groupId>geronimo</groupId>
      <artifactId>tomcat</artifactId>
      <version>1.1.2-SNAPSHOT</version>
      <type>car</type>
    </moduleId>
    <dependencies>
      <dependency>
        <groupId>geronimo</groupId>
        <artifactId>rmi-naming</artifactId>
        <type>car</type>
      </dependency>
      <dependency>
        <groupId>geronimo</groupId>
        <artifactId>geronimo-tomcat</artifactId>
        <type>jar</type>
        <import>classes</import>
      </dependency>
    </dependencies>
    <hidden-classes/>
    <non-overridable-classes/>
  </environment>
  <gbean name="TomcatResources"
class="org.apache.geronimo.system.util.DirectoryInitializationGBean">
    <attribute name="prefix">META-INF/geronimo-tomcat</attribute>
    <attribute name="path">var/catalina</attribute>
    <reference name="ServerInfo">
      <name>ServerInfo</name>
    </reference>
  </gbean>
  <!--david jencks is not convinced that this gbean should be in this plan and
thinks it might
belong in the console plan-->
```

```

    <gbean name="TomcatWebManager"
class="org.apache.geronimo.tomcat.TomcatManagerImpl"></gbean>
    <!--The following is the equivalent of the server.xml file, but done with GBeans-->
    <!--The TomcatContainer/Service-->
    <gbean name="TomcatWebContainer"
class="org.apache.geronimo.tomcat.TomcatContainer">
    <attribute name="catalinaHome">var/catalina</attribute>
    <reference name="EngineGBean">
    <name>TomcatEngine</name>
    </reference>
    <reference name="ServerInfo">
    <name>ServerInfo</name>
    </reference>
    <reference name="WebManager">
    <name>TomcatWebManager</name>
    </reference>
    </gbean>
    <gbean name="TomcatWebConnector"
class="org.apache.geronimo.tomcat.ConnectorGBean">
    <attribute name="name">HTTP</attribute>
    <attribute name="host">localhost</attribute>
    <attribute name="port">8090</attribute>
    <attribute name="maxHttpHeaderSizeBytes">8192</attribute>
    <attribute name="maxThreads">150</attribute>
    <attribute name="minSpareThreads">25</attribute>
    <attribute name="maxSpareThreads">75</attribute>
    <attribute name="hostLookupEnabled">>false</attribute>
    <attribute name="redirectPort">8453</attribute>
    <attribute name="acceptQueueSize">100</attribute>
    <attribute name="connectionTimeoutMillis">20000</attribute>
    <attribute name="uploadTimeoutEnabled">>false</attribute>
    <reference name="TomcatContainer">
    <name>TomcatWebContainer</name>
    </reference>
    </gbean>
    <gbean name="TomcatAJPConnector"
class="org.apache.geronimo.tomcat.ConnectorGBean">
    <attribute name="protocol">AJP</attribute>
    <attribute name="name">AJP</attribute>
    <attribute name="host">localhost</attribute>
    <attribute name="port">8019</attribute>
    <attribute name="hostLookupEnabled">>false</attribute>
    <attribute name="redirectPort">8453</attribute>
    <reference name="TomcatContainer">
    <name>TomcatWebContainer</name>
    </reference>
    </gbean>
    <gbean name="TomcatWebSSLConnector"
class="org.apache.geronimo.tomcat.HttpsConnectorGBean">
    <attribute name="name">HTTPS</attribute>
    <attribute name="host">localhost</attribute>
    <attribute name="port">8453</attribute>
    <attribute name="maxHttpHeaderSizeBytes">8192</attribute>
    <attribute name="maxThreads">150</attribute>
    <attribute name="minSpareThreads">25</attribute>
    <attribute name="maxSpareThreads">75</attribute>
    <attribute name="hostLookupEnabled">>false</attribute>

```

```

    <attribute name="acceptQueueSize">100</attribute>
    <attribute name="uploadTimeoutEnabled">false</attribute>
    <attribute name="clientAuthRequired">false</attribute>
    <attribute name="algorithm">Default</attribute>
    <attribute name="secureProtocol">TLS</attribute>
    <attribute name="keystoreFileName">var/security keystores/geronimo-
default</attribute>
    <attribute name="keystorePassword">secret</attribute>
    <reference name="TomcatContainer">
      <name>TomcatWebContainer</name>
    </reference>
    <reference name="ServerInfo">
      <name>ServerInfo</name>
    </reference>
  </gbean>
  <!--Engine-->
  <gbean name="TomcatEngine" class="org.apache.geronimo.tomcat.EngineGBean">
    <attribute name="className">org.apache.geronimo.tomcat.TomcatEngine</attribute>
    <attribute name="initParams">name=Geronimo</attribute>
    <reference name="DefaultHost">
      <name>TomcatHost</name>
    </reference>
    <references name="Hosts">
      <pattern>
        <type>Host</type>
      </pattern>
    </references>
    <reference name="RealmGBean">
      <name>TomcatJAASRealm</name>
    </reference>
    <reference name="TomcatValveChain">
      <name>FirstValve</name>
    </reference>
    <dependency>
      <name>TomcatResources</name>
    </dependency>
  </gbean>
  <gbean name="TomcatAccessLogManager"
class="org.apache.geronimo.tomcat.TomcatLogManagerImpl">
    <reference name="ServerInfo">
      <name>ServerInfo</name>
    </reference>
    <references name="LogGBeans">
      <pattern>
        <name>FirstValve</name>
      </pattern>
    </references>
  </gbean>
  <!--Valve-->
  <gbean name="FirstValve" class="org.apache.geronimo.tomcat.ValveGBean">
    <attribute
name="className">org.apache.catalina.valves.AccessLogValve</attribute>
    <attribute name="initParams">prefix=localhost_access_log.
      suffix=.txt
      pattern=common</attribute>
  </gbean>
  <!--Realm-->
  <!--This is an example TomcatJAASRealm-->

```

```

    <gbean name="TomcatJAASRealm" class="org.apache.geronimo.tomcat.RealmGBean">
      <attribute
name="className">org.apache.geronimo.tomcat.realm.TomcatJAASRealm</attribute>
      <attribute
name="initParams">userClassNames=org.apache.geronimo.security.realm.providers.Geron
imoUserPrincipal

roleClassNames=org.apache.geronimo.security.realm.providers.GeronimoGroupPrincipal<
/attribute>
    </gbean>
    <!--Host-->
    <gbean name="TomcatHost" class="org.apache.geronimo.tomcat.HostGBean">
      <attribute name="className">org.apache.catalina.core.StandardHost</attribute>
      <attribute name="initParams">name=localhost
        appBase=
        workDir=work</attribute>
    </gbean>
  </module>

```

Table 18-6 describes the tomcat configuration from Listing 18-5.

Table 18-6: An Explanation of the GBeans in the tomcat Configuration

GBean Name	Description
TomcatResources	Manages a directory for the Tomcat configuration files.
TomcatWebManager	Provides for the management of the Tomcat services (for example, HTTP, HTTPS, and so on).
TomcatWebContainer	This is the embedded Tomcat container.
TomcatWebConnector	The HTTP protocol service for Tomcat.
TomcatAJPConnector	The AJP protocol service for Tomcat.
TomcatWebSSLConnector	The HTTPS protocol service for Tomcat.
TomcatEngine	Allows a default JAAS realm to be injected into Tomcat.
TomcatAccessLogManager	Provides for management of the Tomcat access log.
FirstValve	An example is Tomcat Valve.
TomcatJAASRealm	An example is JAAS Realm.
TomcatHost	Wraps the Tomcat Host object

The tomcat module integrates Tomcat with Geronimo, allowing it to be the default Web container in the assembly and to be managed via Geronimo's JSR-77 lifecycle.

Listing 18-6: The tomcat-deployer Plan

```

<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="http://geronimo.apache.org/xml/ns/deployment-1.1">
  <environment>

```

```

<moduleId>
  <groupId>geronimo</groupId>
  <artifactId>tomcat-deployer</artifactId>
  <version>1.1.2-SNAPSHOT</version>
  <type>car</type>
</moduleId>
<dependencies>
  <dependency>
    <groupId>geronimo</groupId>
    <artifactId>j2ee-deployer</artifactId>
    <type>car</type>
  </dependency>
  <dependency>
    <groupId>geronimo</groupId>
    <artifactId>tomcat</artifactId>
    <type>car</type>
    <import>classes</import>
  </dependency>
  <dependency>
    <groupId>geronimo</groupId>
    <artifactId>geronimo-tomcat-builder</artifactId>
    <type>jar</type>
    <import>classes</import>
  </dependency>
</dependencies>
<hidden-classes/>
<non-overrideable-classes/>
</environment>
<gbean name="TomcatWebBuilder"
class="org.apache.geronimo.tomcat.deployment.TomcatModuleBuilder">
  <attribute name="tomcatContainerName">?name=TomcatWebContainer</attribute>
  <references name="WebServiceBuilder">
    <pattern>
      <name>WebServiceBuilder</name>
    </pattern>
    <pattern>
      <name>UnavailableWebServiceBuilder</name>
    </pattern>
  </references>
  <xml-attribute name="defaultEnvironment">
    <environment>
      <dependencies>
        <dependency>
          <groupId>geronimo</groupId>
          <artifactId>j2ee-server</artifactId>
          <!--<version>1.1.2-SNAPSHOT</version>-->
          <type>car</type>
        </dependency>
        <dependency>
          <groupId>geronimo</groupId>
          <artifactId>tomcat</artifactId>
          <!--<version>1.1.2-SNAPSHOT</version>-->
          <type>car</type>
        </dependency>
      </dependencies>
      <hidden-classes/>
      <non-overrideable-classes>
        <filter>java.</filter>

```

```

    <filter>javax.</filter>
    <filter>org.apache.geronimo.</filter>
    <filter>org.apache.jasper.</filter>
    <filter>org.apache.naming.</filter>
    <filter>org.apache.catalina.</filter>
    <filter>org.apache.tomcat.</filter>
    <filter>org.apache.tomcat.</filter>
    <filter>org.xml.</filter>
    <filter>org.w3c.</filter>
  </non-overridable-classes>
</environment>
</xml-attribute>
</gbean>
</module> >

```

Table 18-7 describes the tomcat-deployer GBean.

Table 18-7: The Single GBean in the tomcat-deployer Configuration

GBean Name	Description
TomcatWebBuilder	A GBean for deploying modules in Tomcat

Reviewing the configuration for a familiar component oftentimes allows people to apply what they know about the component to its integration into Geronimo. The reason this is important is for consideration when building a custom assembly. This is exactly the exercise in the next section.

Why Create a Custom Assembly?

Anyone can create a custom assembly in Geronimo, but why would you care to create such a thing? This feature is the true power of Geronimo. So far, this feature has yet to be fully utilized by the Geronimo community. There's certainly a lot of talk about creating custom assemblies to address different needs, but redistribution and internal reuse are the biggest reasons for creating a custom assembly. In addition, the Geronimo kernel could possibly be installed on handheld and/or small devices (for example, network equipment, entertainment consoles, phones, and so on), and a micro assembly could be deployed to any of these small devices.

Redistribution

Many companies already distribute Open Source software as part of a product line, and Geronimo will be no different in this regard. Just as many companies redistribute Apache Tomcat today, Geronimo will also become a target of redistribution. More importantly, users who need to distribute a very flexible application server with their own changes intact will now have the ability to do so without fear of having to pay licensing fees. To date, this has not been possible.

Geronimo has opened the door to many new possibilities when considering redistribution of a Java application server. The ability to redistribute is made possible by another important requirement of Geronimo — its use of the Apache License. The Apache License is an Open Source Initiative (OSI) approved license that is very commercial-friendly in the sense that it allows for modification and redistribution without any penalty. But redistribution is not the only benefit to creating a custom assembly.

Internal Reuse

Internal reuse of a custom assembly is a very compelling concept as well. Outside the needs of redistribution, many companies have internal needs that can be addressed by creating custom assemblies. Instead of wrangling many different pieces of software independent of one another, why not bring them all under a single management platform? Geronimo is perfect for this task. What's more, Geronimo places no restrictions on what can be integrated. No matter whether a piece of software is Open Source or commercial, as long as its license allows, the software can be integrated with Geronimo.

Small Devices

Deploying the Geronimo kernel to smaller devices is yet another reason for creating custom assemblies. Devices such as network equipment, PDAs, and other handheld devices, kiosks, and so on, are all targets for this discussion. Because the kernel has such a small footprint, this can easily become a reality. The fact that modules are not that big in terms of size means that there are no heavy files to weigh down the size of what's being deployed to a device.

Using the Default Configuration

The binary distribution of Geronimo can most certainly be used with the default configuration. The real power in Geronimo is the fact that it provides the ability to create your own customized version. Customization can be achieved using either the software that ships with Geronimo, or the software of your choice. Geronimo is most certainly not limited to integrating only Apache-licensed software. But the onus is on the user to integrate software outside this realm, which brings us to the discussion on how to create your own customized version of Geronimo.

How to Create a Custom Assembly

The best way to understand custom assemblies is to walk through the creation of one. In this section, a custom assembly will be created that is comprised of Tomcat and the Geronimo core services like logging transactionality, and so on. What follows will fully describe the steps of creating this custom assembly. Because creating a custom assembly requires compilation, this section will outline compiling Geronimo from source.

Determine What Components Are not Needed

When creating a custom assembly, most people will begin to think of a custom assembly in terms of what components should be included. This is more of a bottom-up approach. The reality is that most people will quickly recognize that beginning with *something* is better than beginning with *nothing*. In other words, starting to create a custom assembly by using one of the existing J2EE assemblies in a top-down approach will prove to be much easier. While the green field approach might seem like fun, it can prove to be almost impossible if a familiarity with the Geronimo internals does not exist. Even the Geronimo developers choose the top-down approach because it's easier.

When starting with one of the J2EE assemblies, it's important to understand the complexities of the modules and how they play into assemblies. Much of this was described earlier in this chapter. But when performing hands-on work, most of us discover that the reality is even deeper. For the sake of demonstration, this chapter will use the J2EE Tomcat Server as the base to begin the creation of a

custom assembly. Now, we begin to consider the fact that we've already determined that we want to include only Tomcat.

Download the Source

To get started, you'll need to have a copy of the source code locally. A prerequisite for this task is to have the Subversion client installed. Then, check out the source code from the Subversion repository, specifically the 1.1 branch directory. Following is an example of how to do this on the command line:

```
$ svn co https://svn.apache.org/repos/asf/geronimo/server/branches/1.1 geronimo-1.1-src
A   geronimo-1.1-src/xdocs
A   geronimo-1.1-src/xdocs/links.xml
A   geronimo-1.1-src/xdocs/news.html
A   geronimo-1.1-src/xdocs/20031031_jboss.pdf
A   geronimo-1.1-src/xdocs/mailling.html
A   geronimo-1.1-src/xdocs/source.html
A   geronimo-1.1-src/xdocs/faq.fml
A   geronimo-1.1-src/xdocs/images
A   geronimo-1.1-src/xdocs/images/geronimo-logo.png
A   geronimo-1.1-src/xdocs/images/deployer-arch.png
A   geronimo-1.1-src/xdocs/images/white-pixel.png
A   geronimo-1.1-src/xdocs/images/geronimo.gif
A   geronimo-1.1-src/xdocs/images/webservices1.gif
A   geronimo-1.1-src/xdocs/images/security
A   geronimo-1.1-src/xdocs/images/security/AbstractSecurityRealmProvider.png
A   geronimo-1.1-src/xdocs/images/webservices2.gif
A   geronimo-1.1-src/xdocs/images/apache-incubator.png
...
```

The output shown here is just the initial output. After the source code has been completely checked out, the next step is to build the source code.

Build the Source

Building the source code is no small task but the whole process is made easier through the use of Maven. Prerequisites for this step include the following:

- * Java 1.4.2
- * Maven 1.1.
- * A high-speed Internet connection

The entire Geronimo code base will need to be built, and this will take some time because Maven must download all of the necessary dependencies. Following is the command that should be run from the command line to build the entire codebase:

```
$ maven -Dmaven.test.skip=true

  _ _ _ _ _
 |   \   |   _ _ Apache   _ _
 |  | \  | / _ ` \ v / -_) ' \ ~ intelligent projects ~
 |__|  |_\_\_|_| \ / \___|_|_| v. 1.1-beta-2
```

```
DEPRECATED: the default goal should be specified in the <build> section of
project.xml instead of maven.xml
DEPRECATED: the default goal should be specified in the <build> section of
project.xml instead of maven.xml
Starting the reactor...
Our processing order:
build:start:

new:
new0:

Starting the reactor...
Our processing order:
new00:

Starting the reactor...
Our processing order:
Geronimo :: Activation
Geronimo :: ActiveMQ Embedded RAR
Geronimo :: Kernel
Geronimo :: Common
Geronimo :: Util
Geronimo :: System
Geronimo :: Deployment
Geronimo :: Deploy :: Common Config
Geronimo :: Management API
Geronimo :: Core
Geronimo :: J2EE
Geronimo :: J2EE Schema
Geronimo :: Service :: Builder
Geronimo :: Maven Dependency Plugin
Geronimo :: Naming
Geronimo :: Security
Geronimo :: Web Services
Geronimo :: Axis
Geronimo :: Test :: DDBBeans
Geronimo :: Transaction
Geronimo :: Connector
Geronimo :: Security :: Builder
Geronimo :: J2EE
Geronimo :: Naming :: Builder
Geronimo :: Connector :: Builder
Geronimo :: Axis :: Builder
Geronimo :: Client
Geronimo :: Client Builder
Geronimo :: Console Web
Geronimo :: Configuration Converter
Geronimo :: Web :: Builder
Geronimo :: Deploy :: JSR-88
Geronimo :: Deploy :: CLI Tool
Geronimo :: Derby
Geronimo :: Directory
Geronimo :: Deploy :: Hot Deployer
Geronimo :: Installer Processing
Geronimo :: Installer Support
Geronimo :: JavaMail Transport
Geronimo :: Jetty
Geronimo :: Jetty :: Builder
```

```
Geronimo :: JMX Remoting
Geronimo :: Mail
Geronimo :: Scripts
Geronimo :: Timer
Geronimo :: Tomcat
Geronimo :: Tomcat :: Builder
Geronimo :: Upgrade
Geronimo :: Maven Assembly Plugin
Geronimo :: Maven Deployment Plugin
Geronimo :: IZPack Installer Build Plugin
Geronimo :: Maven Packaging Plugin
...
```

This command executes a Maven goal named `new` that will build all the source code. The output shown above is just the initial output. The build will take some time to complete because Maven will need to download many dependencies via the Internet. The preceding command contains a property that is passed in, `maven.test.skip=true`. This property tells Maven to skip all tests, which will speed up the build considerably. However, even with this little hint, the entire build will still take some time. Upon completing the build, the last output from Maven should look something like the following:

```
...
build:end:

BUILD SUCCESSFUL
Total time   : 36 minutes 55 seconds
Finished at  : Monday, January 9, 2006 10:16:55 PM MST
```

If the `BUILD SUCCESSFUL` message is output, then that means the code has finished being compiled and packaged. Now, it's time to actually begin creating the new assembly.

Creating the Assembly

The easiest way to begin is to make a copy of an existing assembly. For this task, the `j2ee-tomcat-server` assembly will be used as the base to begin the work to create a new assembly. Simply copy the entire directory:

```
$ cd assemblies
$ cp -R ../j2ee-tomcat-server ./activemq-derby-tomcat-assembly
```

Once the copy command is completed, the name and description should be changed so that when the assembly is being built, the output from it is accurate and discernable from the real `j2ee-tomcat-server`. The ID and name elements of the Maven POM file (`project.xml`) should be changed from this:

```
<id>geronimo-tomcat-j2ee</id>
<name>Geronimo Assembly for a J2EE Server running Tomcat</name>
```

to this:

```
<id>geronimo-activemq-derby-tomcat</id>
<name>Geronimo Assembly for a Server running only Tomcat</name>
```

The next step is to remove all modules and GBeans excepting Tomcat, the core Geronimo services, and all the necessary dependencies for these items.

Removing the Unneeded Parts

With the knowledge of the components being included, working backward to remove other modules is both faster and easier. For example, following are known modules that are not needed in this new assembly:

- * Anything with EJBs
- * Anything to do with Web Services
- * Anything not needed by Tomcat or the Geronimo core

The modules associated with these items include `activemq`, `activemq-broker`, `axis`, the `axis-deployer`, `openejb`, and the `openejb-deployer`. These modules will be removed. To determine what additional modules should be removed, take a look at the Maven POM file (`project.xml`). By simply perusing the list of dependencies, it's easy to see a number of other modules that can easily be targeted for removal. Additional modules that should be removed from the new assembly include `client` and all `client-*` modules, `j2ee-corba`, `javamail`, and `uddi-tomcat`.

To actually remove these modules from the assembly, they need to be commented out or removed from the Maven POM file in the assembly. Once these are commented out, there are some required items that must be replaced with null implementations.

In addition, there are modules that will need to be commented out or removed from the `src/var/config/config.xml` file as well. The list of modules that must be removed from the `config.xml` file correspond directly to the list of modules that needed to be removed from the Maven POM file.

Plan Files Are Not Distributed in the Binary Downloads

Unfortunately, the plan files for modules are not distributed with the binary downloads of Geronimo. These are only available in the Geronimo source code. To check out a copy of the source code for Geronimo, please follow the instructions found on the Geronimo Web site (<http://geronimo.apache.org/svn.html>).

Upon downloading, the source code, the plan files can be found in their respective `configs` directory. For example, relative to the trunk directory, the plan file for the ActiveMQ broker configuration can be found in the following:

```
configs/activemq-broker/src/plan/plan.xml
```

Upon changing anything in a plan file, it can be deployed to an assembly by building the modules and the assemblies using the following command:

```
maven -o -Dmaven.test.skip new4 new5
```

Now that the unneeded modules have been removed, it's time to address the required GBeans.

Replacing Required Parts

Certain GBeans in the Geronimo configuration are required. Remember that the `j2ee-server` and `j2ee-system` modules are the base from which every other configuration inherits in some way. Table 18-8 outlines the GBeans that will be replaced.

Table 18-8: Required GBeans that Need Replacing in the New Assembly

GBean Name	Description
AppClientModuleBuilder	
AxisBuilder	Used to deploy Axis web service components
EJBReferenceBuilder	Used to deploy EJB reference objects
OpenEJBModuleBuilder	Used to deploy EJB JAR modules
ServiceReferenceBuilder	

The builder components in Table 18-8 each need to be replaced with a null implementation whose methods simply return null instead of a real object. The idea is to slide in a new implementation without disrupting the entire environment. There's even a little trick in the deployment plan to keep the disruption to a minimum. See Listings 18-7 and 18-8 for an example.

Listing 18-7: The j2ee-deployer Module's Deployment Plan

```

<gbean name="EARBuilder"
  class="org.apache.geronimo.j2ee.deployment.EARConfigBuilder">
  <attribute name="defaultParentId">
    ${pom.groupId}/j2ee-server/${pom.currentVersion}/car
  </attribute>
  <attribute name="transactionContextManagerObjectName">
    *:name=TransactionContextManager,*
  </attribute>
  <attribute name="connectionTrackerObjectName">
    *:name=ConnectionTracker,*</attribute>
  <attribute name="transactionalTimerObjectName">
    geronimo.server:name=TransactionalThreadPooledTimer,*</attribute>
  <attribute name="nonTransactionalTimerObjectName">
    geronimo.server:name=NonTransactionalThreadPooledTimer,*</attribute>
  <attribute name="corbaGBeanObjectName">
    geronimo.server:J2EEApplication=null,J2EEModule=geronimo/server-
corba/${pom.currentVersion}/car,J2EEServer=geronimo,j2eeType=CORBABean,name=Server
  </attribute>
  <reference name="Repository">
    <gbean-name>*:name=Repository,*</gbean-name>
  </reference>
  <reference name="WebConfigBuilder">
    <name>WebBuilder</name>
  </reference>
  <reference name="EJBConfigBuilder">
    <module>*</module>
    <name>EJBBuilder</name>
  </reference>
  <reference name="ConnectorConfigBuilder">
    <name>ConnectorBuilder</name>
  </reference>
  <reference name="AppClientConfigBuilder">
    <name>AppClientBuilder</name>
  </reference>
  <reference name="ResourceReferenceBuilder">
    <name>ConnectorBuilder</name>
  </reference>
  <reference name="ServiceReferenceBuilder">

```

```

        <module>*</module>
        <name>WebServiceBuilder</name>
    </reference>
    <reference name="EJBReferenceBuilder">
        <module>*</module>
        <name>ServerEJBReferenceBuilder</name>
    </reference>
</gbean>

```

The Builders in Geronimo are actually what are most commonly known as deployers. The word deployer was not used in this context because of the confusion with JSR-88 deployer implementations.

Notice in the `EARBuilder` GBean's definition from the `j2ee-deployer` modules in Listing 18-7 that there is a reference to another GBean whose name is `EJBBuilder`. This reference is pointing to a GBean named `EJBBuilder` in the `openejb-deployer` configuration. Listing 18-8 shows this GBean definition.

Listing 18-8: The `EJBBuilder` GBean Definition

```

<gbean name="EJBBuilder" class="org.openejb.deployment.OpenEJBModuleBuilder">
  <attribute name="defaultParentId">
    ${pom.groupId}/openejb/${pom.currentVersion}/car,${pom.groupId}/axis/${pom.currentVersion}/car
  </attribute>
  <attribute name="listener">
    geronimo.server:J2EEApplication=null,J2EEModule=${pom.groupId}/j2ee-server/${pom.currentVersion}/car,J2EEServer=geronimo,j2eeType=GBean,name=WebContainer
  </attribute>
  <reference name="WebServiceLinkTemplate">
    <name>WebServiceEJBLinkTemplate</name>
  </reference>
  <reference name="WebServiceBuilder">
    <module>*</module>
    <name>WebServiceBuilder</name>
  </reference>
  <reference name="Repository">
    <gbean-name>*:name=Repository,*</gbean-name>
  </reference>
</gbean>

```

Instead of making the `j2ee-deployer` configuration point to the `EJBBuilder` in the `openejb-deployer` configuration, it will be pointed at the new unavailable definition shown in Listing 18-9.

Listing 18-9: The unavailable-`openejb-deployer`'s Definition

```

<gbean name="EJBBuilder"
class="org.apache.geronimo.j2ee.deployment.UnavailableModuleBuilder" />

```

Instead of assigning the new unavailable reference its own unique name, it is given the same name as the GBean that it is replacing. Then the `openejb-deployer` configuration is excluded from the new assembly's `config.xml` file and the `unavailable-openejb-deployer` is included. Listing 18-10 shows this inclusion.

Listing 18-10: The Inclusion of the unavailable-open-ejb Configuration in the New Assembly

```
<configuration name="geronimo/unavailable-openejb-  
deployer/${pom.currentVersion}/car"/>
```

Now that all the unneeded components have been removed and all required parts replaced with null implementations, everything needs to be built.

Building the New Modules and Assembly

Because the entire code base was built earlier in this chapter, building all the new additions will be much easier and much faster. The reason for this is that Maven has already downloaded all external dependencies to the local Maven repository. Maven provides to build modes for building — online and offline. The *online mode* tells Maven to fetch external dependencies via the Web and the *offline mode* tells Maven not to fetch external dependencies. There are some additional rules surrounding this, but those are best left to the Maven documentation.

To build all of these new changes to the assemblies and modules, there are two Maven goals that must be executed to just rebuild those items. However, rebuilding all the assemblies and all the modules is not necessary either. The fastest way to build all the changes made is by hand in the following order:

1. Rebuild the `j2ee-builder` module:

```
$ cd ./modules/j2ee-builder  
$ maven -o -Dmaven.test.skip=true  
...
```

2. Build the unavailable modules:

```
$ cd ../../configs/unavailable-appclient-deployer  
$ maven -o  
...  
$ cd ../../configs/unavailable-axis-deployer  
$ maven -o  
...  
$ cd ../../configs/unavailable-openejb-deployer  
$ maven -o  
...
```

3. Build the new assembly:

```
$ cd ../../assemblies/my-minimal-tomcat-server  
$ maven -o  
...
```

Everything should be compiled and packaged and ready for testing now.

Starting the New Assembly

One manner in which to start the new assembly is to execute the `geronimo.sh` script and feed it the `'run'` command. Listing 18-11 shows the start commands and the output from the new assembly.

Listing 18-11: Starting Up the New Assembly

```
$ cd ./target/geronimo-1.1.2-SNAPSHOT/
$ ./bin/geronimo.sh run
Using GERONIMO_BASE: /tmp/geronimo-1.1-src/assemblies/minimal-tomcat-
server/target/geronimo-1.1.2-SNAPSHOT
Using GERONIMO_HOME: /tmp/geronimo-1.1-src/assemblies/minimal-tomcat-
server/target/geronimo-1.1.2-SNAPSHOT
Using GERONIMO_TMPDIR: /tmp/geronimo-1.1-src/assemblies/minimal-tomcat-
server/target/geronimo-1.1.2-SNAPSHOT/var/temp
Using JRE_HOME:
/System/Library/Frameworks/JavaVM.framework/Versions/1.5/Home
Booting Geronimo Kernel (in Java 1.5.0_06)...
Starting Geronimo Application Server v1.1.2-SNAPSHOT
[*****] 100% 29s Startup complete
Listening on Ports:
 1099 0.0.0.0 RMI Naming
 4242 0.0.0.0 Remote Login Listener
 8009 0.0.0.0 Tomcat Connector AJP
 8080 0.0.0.0 Tomcat Connector HTTP
 8443 0.0.0.0 Tomcat Connector HTTPS
 9999 0.0.0.0 JMX Remoting Connector

Geronimo Application Server started
```

As long as the new assembly starts correctly, use the console Web application as a quick test by visiting the following URL:

`http://localhost:8080/console`

Figures 18-8 and 18-9 show some initial screenshots of the console when it is first visited. The pages for the new assembly should match those screenshots.

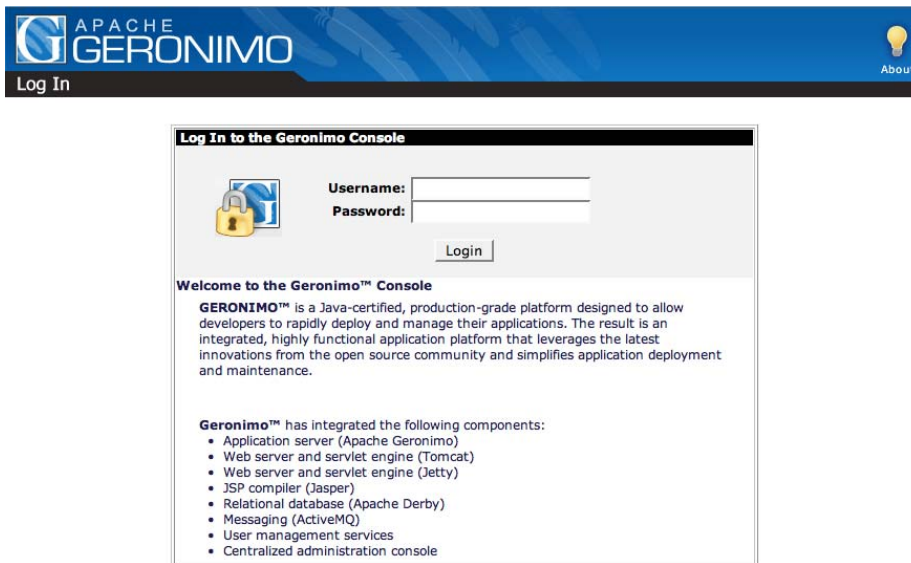


Figure 18-8: The default Geronimo Web console login page

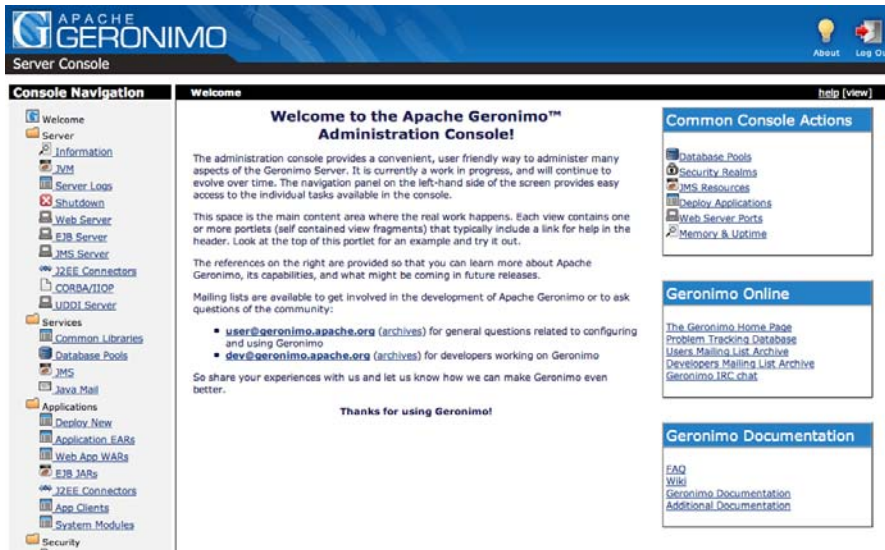


Figure 18-9: The default Geronimo Web console home page

The Best Is Yet to Come

It has been said that the most powerful parts of Geronimo are the assemblies, simply because they offer an unmatched capability in the realm of Java application servers. All the alternate choices of application servers available are built specifically for particular specifications and technologies. Because Geronimo's kernel has no knowledge of any one specification or technology, it can address just about any software need (within the scope of Java). Such custom assemblies will only come from the Geronimo community. This chapter demonstrated how to build a custom assembly in the hopes that more custom assemblies will continue to be created.

In addition to simply creating new custom assemblies, making them available to the world is very important. The hope of the Geronimo team is that, as new assemblies are created, the creators of those assemblies consider either contributing them to the Geronimo project or simply make them available to the community in some way, possibly through the use of Geronimo plugins.

Community

The Geronimo community is composed of people and companies from countries all over the world. This blend of people from all walks of life is what makes the community so vibrant. These people offer experiences from many different industries and markets, and this expertise is what will bring more modules and assemblies to the project.

As stated many times now, Geronimo is not only about J2EE. With the work being scheduled for the support of JEE 5, there are so many more untapped areas to which the community can contribute greatly. As this chapter demonstrates, modules are building blocks in a sense. By offering the ability to so easily create new building blocks, the possibilities are endless.

Pre-Packaged Modules as Building Blocks

There are currently enough modules that are part of Geronimo to pass J2EE 1.4 certification, but this is only the beginning. While creating GBeans to bring other software into the Geronimo lifecycle is not very difficult, this is still a small barrier to overcome. What's needed to offer more custom assemblies (and to make building them easier) are more pre-packaged modules in the form of Geronimo plugins. Once modules become prolific, then creating new assemblies using them is only an idea away. With the release of Geronimo 1.1, these ideas have not yet been tapped, but the future looks awfully bright.

Summary

This chapter discussed the kernel, some history behind it, and why certain changes in its architecture and design were made. Modules and how collections of modules play into assemblies were also explained. Lastly, the steps to building custom assemblies were presented in an effort to help readers get started with this task. Custom assemblies and modules are the true future of Geronimo. To make this happen, everyone must help to continue building a healthy, strong, and vibrant community. After all, an Open Source project is only as good as its community.