CHAPTER 1

Introducing Tomcat

This, as befits a first chapter in a book on Tomcat, is a short history of dynamic web content and how Tomcat fits into that history. Once you've dealt with that, you'll learn about Tomcat's architecture and its modular approach to configuration.

Understanding the Web Today

The Web isn't solely made up of static pages that show the same document to every user; many pages contain content generated independently for each viewer. Although static files still have their place, many useful and necessary web sites would be unable to function without dynamic content. For example, Amazon.com is one of the major success stories of the Web and is often the reason people go online for the first time. Without dynamic content, such as shopping baskets, personal recommendations, and personalized welcome messages, Amazon.com wouldn't be the success it has been, and many people wouldn't be online.

The Common Gateway Interface (CGI) was the original dynamic content mechanism that executed programs on a web server and allowed webmasters to customize their pages, which was extremely popular in the early days of the Web. The CGI model is as follows:

1. The browser sends a request to the server just as it would for a Hypertext Markup Language (HTML) page.
2. The server maps the requested resource to an external program.
3. The server runs the external program and passes it the original Hypertext Transfer Protocol (HTTP) request.
4. The external program executes and sends its results to the server.
5. The server passes the program's output to the browser as an HTTP response.

CGI has been implemented in many programming languages, but Perl was, and still is, the most popular language for developing CGI applications. However, CGI isn't very efficient; each time the server receives a request, it must start a new copy of the external program.

So, if only a small number of users request a CGI program simultaneously, it's not too big of a problem. However, it's a different story if hundreds or thousands of users request the resource simultaneously. Every copy of the program requires a share of the server's processing

power, which is rapidly used as requests pile up. The situation is made even worse by CGI programs that are written in interpreted languages such as Perl, which result in the launch of large runtime interpreters with each request.

Looking Beyond CGI

Many alternative solutions to CGI have been developed since the Web began. The more successful of these provide an environment that exists inside an existing server or even functions as a server on its own.

Many CGI replacements have been built on top of the Apache server (`www.apache.org`) because of Apache's popular modular application programming interface (API). Developers can use the API to extend Apache's functionality with persistent programs, thus it's ideal for creating programs that create dynamic content. Apache loads modules into its memory when it starts and passes the appropriate HTTP requests to them as needed. It then passes the HTTP responses to the browser once the modules have processed the requests. Because the modules are already in the server's memory, the cost of loading an interpreter is removed, and scripts can execute faster.

Although few developers actually create modules themselves (they're relatively difficult to develop), many third-party modules provide a basis for applications that are much more efficient than normal CGI. The following are a few examples:

- `mod_perl`: This maintains the Perl interpreter in memory, thus removing the overhead of loading a new copy of the Perl interpreter for each request. This is an incredibly popular module.
- `mod_php4`: This module speeds up PHP in the same way that `mod_perl` speeds up Perl.
- `mod_fastcgi`: This is similar to straight CGI, but it keeps programs in memory rather than terminating them when each request is finished.

Microsoft provides an interface to its Internet Information Services (IIS) web server, called the Internet Server Application Programming Interface (ISAPI). Because of its complexity, this API doesn't have the following that Apache's API has, but it's nevertheless a high-performance API. However, IIS is widely used, mainly because it comes as part of many versions of Windows. In Chapter 9, you'll configure Tomcat to work with IIS, so you can combine the best features of both.

Microsoft also developed the Active Server Pages (ASP) technology, which lets you embed scripts, typically VBScript scripts, into standard HTML pages. This model has proved extremely successful and was the catalyst for Java web technology, which I'll discuss next.

Introducing Java on the Web

Java was initially released in the mid-1990s as a way to liven up static web pages. It was platform independent and allowed developers to execute their programs, called *applets*, in the user's browser. An incredible amount of hype surrounded applets: that they would make the Web more exciting and interactive, that they would change the way people bought computers, and that they would reduce all the various operating systems into mere platforms for web browsers.

Applets never really caught on; in fact, other technologies, such as Adobe Flash, became more popular ways of creating interactive web sites. However, Java isn't just for writing applets: you can also use it to create stand-alone, platform-independent applications.

The main contribution of Java to the web is *servlets*, which are another alternative technology to CGI. Just as CGI and its other alternatives aren't stand-alone programs (because they require a web server), servlets require a servlet container to load servlets into memory. The servlet container then receives HTTP requests from browsers and passes them to servlets that generate the response. The servlet container can also integrate with other web servers to use their more efficient static file abilities while continuing to produce the dynamic content. You'll find an example of this in Chapter 9, when you integrate Tomcat with Apache and IIS.

Unfortunately, although servlets are an improvement over CGI, especially with respect to performance and server load, they too have a drawback. They're primarily suitable for processing logic. For the creation of content (that is, HTML), they're less usable. First, hard-coding textual output, including HTML tags, in code makes the application less maintainable. This is because if text in the HTML must be changed, the servlet must be recompiled.

Second, this approach requires the HTML designer to understand enough about Java to avoid breaking the servlet. More likely, however, the programmer of the application must take the HTML from the designer and then embed it into the application: an error-prone task if ever there was one.

To solve this problem, Sun Microsystems created the JavaServer Pages (JSP) technology.

Adding to Servlets: JavaServer Pages

Although writing servlets requires knowledge of Java, a Java newbie can quickly learn some useful JSP techniques. As such, JSP represents a viable and attractive alternative to Microsoft's ASP.

Practically speaking, JSP pages are compiled into servlets, which are then kept in memory or on the file system indefinitely, until either the memory is required or the server is restarted. This servlet is called for each request, thus making the process far more efficient than ASP, since ASP requires the server to parse and compile the document every time a user comes to the site. This means that a developer can write software whose output is easy to verify visually and with a result that works like a piece of software. In fact, JSP took off mainly as a result of its suitability for creating dynamic visual content at a time when the Internet was growing in popularity.

One major practical difference between servlets and JSP pages is that servlets are provided in compiled form and JSP pages often are not (although precompilation is possible). What this means for a system administrator is that servlet files are held in the private resources section of the servlet container, and JSP files are mixed in with static HTML pages, images, and other resources in the public section of servlet container.

Introducing Servlet Containers

JSP pages and servlets require a servlet container to operate at all. Tomcat, the subject of this book, is the reference implementation (RI) servlet container, which means that Tomcat's first priority is to be fully compliant with the Servlet and JSP specifications published by Sun Microsystems. However, this isn't to say that Tomcat isn't worthy of use in production systems. Indeed, many commercial installations use Tomcat.

An RI has the added benefit of refining the specification, whatever the technology may be. As developers add code per the specifications, they can uncover problems in implementation requirements and conflicts within the specification.

As noted previously, the RI is completely compliant with the specification and is, therefore, particularly useful for people who are using advanced features of the specification. The RI is released with the specification, which means that Tomcat is always the first server to provide the new features of the specification when it's finished.

Looking at Tomcat

Tomcat has its origins in the earliest days of the servlet technology. Sun Microsystems created the first servlet container, the Java Web Server, to demonstrate the technology, but it wasn't terribly robust. At the same time, the Apache Software Foundation (ASF) created JServ, a servlet engine that integrated with the Apache web server.

In 1999, Sun Microsystems donated the Java Web Server code to the ASF, and the two projects merged to create Tomcat. Version 3.*x* was the first Tomcat series and was directly descended from the original code that Sun Microsystems provided to the ASF. It's still available and is the RI of the Servlet 2.2 and JSP 1.1 specifications.

In 2001, the ASF released Tomcat 4.0, which was a complete redesign of the Tomcat architecture and which had a new code base. The Tomcat 4.*x* series is the RI of the Servlet 2.3 and JSP 1.2 specifications.

Tomcat 5.*x* was the next version of Tomcat and is the RI of the Servlet 2.4 and JSP 2.0 specifications. Note that two branches of Tomcat 5.*x* exist: Tomcat 5.0.*x* and Tomcat 5.5.*x*. Tomcat 5.5.*x* branched at Tomcat 5.0.27 and is a refactored version that's intended to work with the Java 2 Platform Standard Edition 5.0 (you can use it with Java 2 Standard Edition 1.4, but it requires an additional Compatibility Kit patch).

This book covers the newly released Tomcat 6.x version. This version is the new RI for the Servlet 2.5 and JSP 2.1 specifications.

What's New in Tomcat 6

Tomcat 6 is built using several new features, such as generics, introduced in Java 5. The key new elements from the Tomcat 5 release are support for the latest Java Server Pages (JSP) 2.1 specification (JSR 245) and the Java Servlet 2.5 specification (JSR 154). In addition to JSP 2.1, Tomcat 6 fully supports the Unified Expression Language (Unified EL) 2.1. As you might know, Unified EL 2.1 was made into its own stand-alone package in the JSP 2.1 specification. This means that you should be able to use EL outside of a container such as Tomcat. Tomcat 6 is also the first container to support the Java Server Faces 1.2 specification.

In my experience with Tomcat 6, I have noticed it is a little faster (during startup and shutdown) than its predecessor. It also seems to have a slightly smaller memory footprint. Throughout this book, as we talk about the different aspects of Tomcat 6, you will notice that not a whole lot has changed from the Tomcat 5.5 release. However, you will notice some small changes to the directory structures and start scripts. Of course, if you have not used Tomcat 5.5, you will see more drastic changes in this version, such as a completely new logging mechanism and a lot more ease-of-use features and flexibility.

Understanding Tomcat's Architecture

The latest version of Tomcat is 6, which supports the Servlet 2.5 and JSP 2.1 specifications. It consists of a nested hierarchy of components.

- *Top-level components* exist at the top of the configuration hierarchy in a rigid relationship with one another.
- *Connectors* connect the servlet container to requests, either from a browser or another web server that's handling the static resources.
- *Container components* contain a collection of other components.
- *Nested components* can reside in containers but can't contain other components.

Figure 1-1 illustrates the structure of a Tomcat configuration.

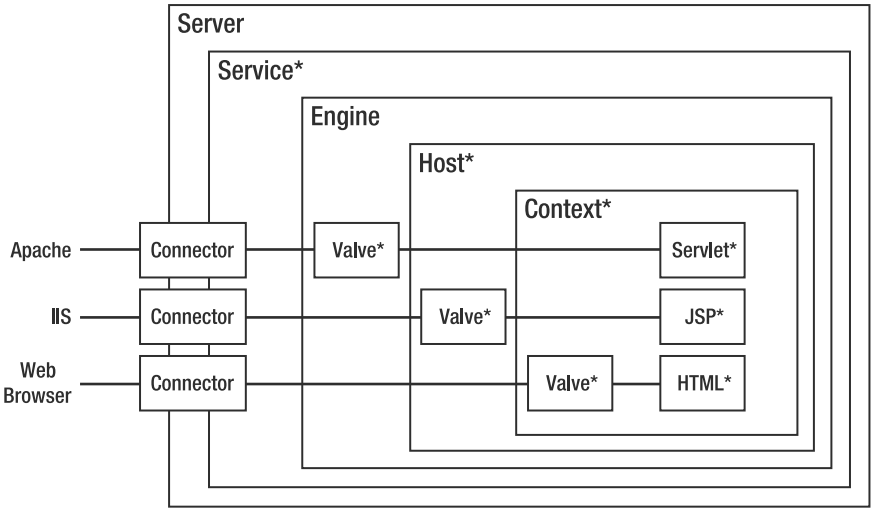

Figure 1-1. *An example Tomcat configuration. The components marked with a star can occur multiple times.*

When configuring Tomcat, you can remove some of these objects without affecting the server. Notably, the engine and host may be unnecessary if you're using a web server such as Apache.

You won't be surprised to hear that Tomcat is configured with an Extensible Markup Language (XML) file that mirrors the component hierarchy. You'll learn about this file, called `server.xml`, in Chapter 4.

In the next couple of sections, you'll look into each component in turn.

Top-Level Components

The top-level components are the Tomcat server, as opposed to the other components, which are only parts of the server.

The Server Component

The server component is an instance of the Tomcat server. You can create only one instance of a server inside a given Java virtual machine (JVM).

You can set up separate servers configured to different ports on a single server to separate applications so that you can restart them independently. So, if a given JVM crashes, the other applications will be safe in another instance of the server. This is sometimes done in hosting environments where each customer has a separate instance of a JVM so that a badly written application won't cause others to crash.

The Service Component

A service component groups an engine component with its connectors. An engine is a request-processing component that represents the servlet engine. It examines the HTTP headers to determine to which host or context (that is, which web application) it should pass the request. Each service is named so that administrators can easily identify log messages sent from each service.

This component accepts requests, routes them to the appropriate web application, and returns the result of the request processing.

The Connector Components

Connectors connect web applications to clients. They're the point where requests are received from clients, and each has a unique port on the server. Tomcat's default HTTP port is 8080 to avoid interference with any web server running on port 80, the standard HTTP port. However, you can change this as long as the new port doesn't already have a service associated with it.

The default HTTP connector implements HTTP 1.1. The alternative is the Apache JServ Protocol (AJP) connector, which is a connector for linking with Apache to use its Secure Sockets Layer (SSL) and static content-processing capabilities. I'll discuss each of these in Chapter 9.

The Container Components

The container components receive the requests from the top-level components as appropriate. They then deal with the request process and return the response to the component that sent it to them.

The Engine Component

The engine component is the top-level container and can't be contained by another container component. Only one may be contained in each service component.

The top-level container doesn't have to be an engine, because it only has to implement the container interface. This interface ensures the object implementing it is aware of its position in the component hierarchy, provides a realm for user authentication and role-based authorization, and has access to a number of resources including its session manager and some important internal structures.

The container at this level is usually an engine, so you'll see it in that role. As mentioned earlier, the container components are request-processing components, and the engine is no exception. In this case, it represents the Catalina servlet engine. It examines the HTTP headers to determine to which virtual host or context to pass the request. In this way, you can see the progression of the request from the top-level components down the hierarchy of components.

If Tomcat is used as a stand-alone server, the defined engine is the default. However, if Tomcat is configured to provide servlet support with a web server providing the static pages, the default engine is overridden, as the web server has normally determined the correct destination for the request.

The host name of the server is set in the engine component if required. An engine may contain hosts representing a group of web applications and contexts, each representing a single web application.

The Host Component

A host component is analogous to the Apache virtual host functionality. It allows multiple servers to be configured on the same physical machine and be identified by separate Internet Protocol (IP) addresses or host names. In Tomcat's case, the virtual hosts are differentiated by a fully qualified host name. Thus, you can have `www.apress.com` and `www.moodie.com` on the same server. In this case, the servlet container routes requests to the different groups of web applications.

When you configure a host, you set its name; the majority of clients will usually send both the IP address of the server and the host name they used to resolve the IP address. The engine component inspects the HTTP header to determine which host is being requested.

The Context Component

The final container component, and the one at the lowest level, is the context, also known as the web application. When you configure a context, you inform the servlet container of the location of the application's root folder so that the container can route requests effectively. You can also enable dynamic reloading so that any classes that have changed are reloaded into memory. This means the latest changes are reflected in the application. However, this is resource intensive and isn't recommended for deployment scenarios.

A context component may also include error pages, which will allow you to configure error messages consistent with the application's look and feel.

Finally, you can also configure a context with initialization parameters for the application it represents and for access control (authentication and authorization restrictions). More information on these two aspects of web application deployment is available in Chapter 5.

The Nested Components

The nested components are nested within container components and provide a number of administrative services. You can't nest all of them in every container component, but you can nest many of them this way. The exception to the container component rule is the global resources component, which you can nest only within a server component.

The Global Resources Component

As already mentioned, this component may be nested only within a server component. You use this component to configure global Java Naming and Directory Interface (JNDI) resources that all the other components in the server can use. Typically these could be data sources for database access or serverwide constants for use in application code.

The Loader Component

The loader component may be nested only within a context component. You use a loader to specify a web application's class loader, which will load the application's classes and resources into memory. The class loader you specify must follow the Servlet specification, though it's unlikely you'll find it necessary to use this component because the default class loader works perfectly well.

The Logger Component

With Tomcat 6, you should use a logging implementation such as Log4J, which is covered in more depth in Chapter 4. The logger component, as it exists in Tomcat 5.0.x and previous versions, has not been available since the Tomcat 5.5.x release.

The Manager Component

The manager component represents a session manager for working with user sessions in a web application. As such, it can be included only in a context container. A default manager component is used if you don't specify an alternative, and, like the loader component mentioned previously, you'll find that the default is perfectly good.

The Realm Component

The realm for an engine manages user authentication and authorization. As part of the configuration of an application, you set the roles that are allowed to access each resource or group of resources, and the realm is used to enforce this policy.

Realms can authenticate against text files, database tables, Lightweight Directory Access Protocol (LDAP) servers, and the Windows network identity of the user. You'll see more of this in Chapter 11.

A realm applies across the entire container component in which it's included, so applications within a container share authentication resources. By default, a user must still authenticate separately to each web application on the server. (This is called *single sign-on*.) You'll see how you can change this in Chapter 7.

The Resources Component

You can add the resources component to a context component. It represents the static resources in a web application and allows them to be stored in alternative formats, such as compressed files. The default is more than sufficient for most needs.

The Valve Component

You can use valve components to intercept a request and process it before it reaches its destination. Valves are analogous to filters as defined in the Servlet specification and aren't in the JSP or Servlet specifications. You may place valve components in any container component.

Valves are commonly used to log requests, client IP addresses, and server usage. This technique is known as *request dumping*, and a request dumper valve records the HTTP header information and any cookies sent with the request. Response dumping logs the response headers and cookies (if set) to a file.

Valves are typically reusable components, so you can add and remove them from the request path according to your needs; web applications can't detect their presence, so they shouldn't affect the application in any way. (However, performance may suffer if a valve is added.) If your users have applications that need to intercept requests and responses for processing, they should use filters as per the Servlet specification.

You can use other useful facilities, such as listeners, when configuring Tomcat. However, filters aren't defined as components. You'll deal with them in Chapter 7.

Summary

This chapter was a quick introduction to dynamic web content and the Tomcat web server. You learned about the emergence of CGI, its problems, and the various solutions that have been developed over the years. You saw that servlets are Java's answer to the CGI problem and that Tomcat is the reference implementation of the Servlet specification as outlined by Sun Microsystems.

The chapter then discussed Tomcat's architecture and how all its components fit together in a flexible and highly customizable way. Each component is nested inside another to allow for easy configuration and extensibility.

Now that you're familiar with Tomcat, you'll learn about how to install it on various platforms.