**Module 12**

**Integrating Apache Web Server**

We have discussed how to use Apache Tomcat as a server for both static and dynamic pages of a web application. Apache Foundation, however, owns Apache Httpd Server Project – a high-performance, scalable HTTP server. As its name suggests, Apache Httpd Server (or Apache Web Server, or just Apache, as its mostly known) is optimized for high-performance content serving over the HTTP protocol. It does not know anything about Servlets or JSPs, though. It is possible to closely integrate Apache Web Server and Apache Tomcat so we can benefit from Tomcat’s servlet container, but at the same time serve static content via the high-performance Apache Web Server, and possibly use some of its great load balancing capabilities as well.

In this module we will:

* Describe Apache Web Server
* Learn how to use Apache proxy module to integrate with Apache Tomcat
* Learn how to use AJP protocol for Apache Web Server – Apache Tomcat integration
* Implement a simple load balancer using Apache Web Server and AJP protocol

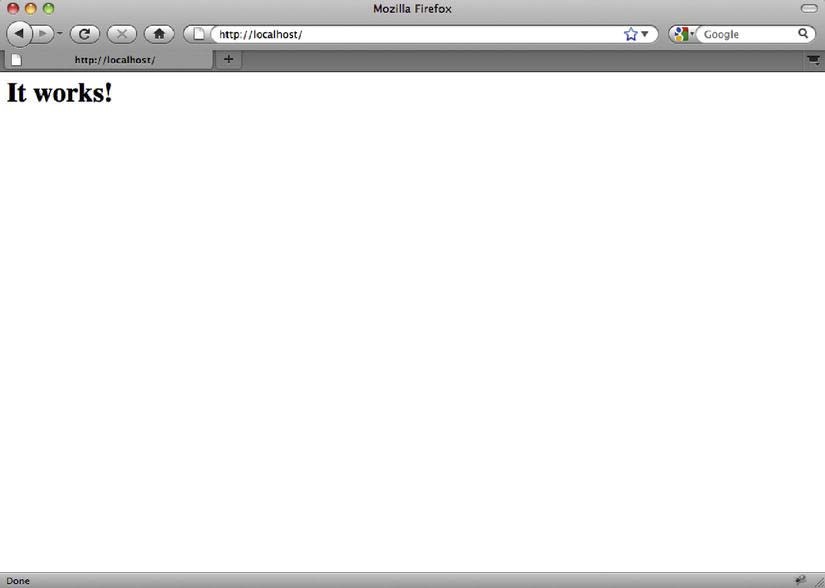
**What Is the Apache Web Server?**

It is fair to say that the Apache Web Server is the de-facto standard http web server available today. Apache Web Server is an open-source project (just like Apache Tomcat) hosted by the Apache Foundation. With its popularity, it has become a synonym for the Apache Foundation, widely known simply as Apache. Rob McCool is the original author of the HTTP daemon at the National Center for Supercomputing Applications, University of Illinois. After the development of the HTTP daemon at NCSA stalled, a group of webmasters came together and continued to develop it – eventually forming the Apache Foundation. The Apache Httpd Server as its main project.

The latest stable release of the Apache Web Server at the time of writing is 2.2.X, and it can be downloaded at [http://httpd.apache.org/.](http://httpd.apache.org/)

Before continuing with this chapter, you will have to download the version for your operating system, and complete installation as per the instructions available on the Apache Web Server site.

After a successful installation, you should see the test page shown in Figure 10-1 in your browser when you go to [http://localhost.](http://localhost/) The default port for Apache Web Server is port 80, which is also the default port for the most browsers, so you don’t have to include port in the test URL. The same page will be available if the port is set – <http://localhost:80/>.



***Figure 12-1.*** *Test page for Apache Web Server, confirming successful installation*

**Integrating Tomcat and Apache Web Server**

There are several reasons why you would consider integrating Tomcat with Apache, instead of using standalone Apache Tomcat as both a servlet container and HTTP server.

Historically, the first reason has been performance. Apache is a high-performance web server that served static content — such as html files, images, and videos — faster than Tomcat. However, with recent Tomcat releases, especially with Tomcat 7, its performance as a web server has improved significantly, and the performance gains of the integration protocol have diminished in single-server instances.

The second reason is the configurability of Apache Web Server. With Apache Web Server’s 15-year lifespan, huge popularity, and large developer community across different programming languages, Tomcat simply cannot match it for all its configuration possibilities. Hundreds of modules are currently available as extensions for Apache Web Servers. Modules range from language-specific extensions (for PHP, Perl, Python, and so on) and security features (SSL and authentication). Then there is the proxying module, mod\_proxy, a powerful URL rewriting module, mod\_rewrite, support for custom logging and request filtering, and so on. Most websites require some of these functionalities, and even if using Apache Tomcat as servlet container, Apache Web Server will most likely be public facing web server for your new website.

Finally, the Apache Web Server is very robust, highly scalable piece of software, which is a must for any production-grade web server of choice. Apache Web Server has good resource handing, minimal memory leaks, and graceful startup and shutdown. It’s not uncommon to see Apache Web Server instances happily running for months without a restart.

As a general rule of thumb, when deciding whether to run Tomcat standalone or integrate it with Apache Web Server, consider the requirements of your web application and deployment infrastructure. If you are likely to use some non-Java scripting for parts of your application – like PHP or Perl – Apache Web Server integration would be the best option, with its rich support for scripting languages. If your web application is expecting to serve a great number of users concurrently, or it needs to perform some heavy processing, consider load balancing multiple Tomcat instances behind Apache Web Server.

Because Apache Web Server has become the industry standard – and a very popular web server – you are more likely to find experienced network administrators with in-depth knowledge of Apache Web Server administration and find running Tomcat behind Apache Web Server more efficient option for maintenance. Apache Web Server has a lot of modules that can extend its functionality quickly and easily, so you can perform various search engine optimization tasks, modify headers, and introduce friendly URLs using URL rewriting with Apache. Finally, Apache Web Server’s startup time is much shorter then Tomcat’s, and with its robustness it can run for months without a restart. You can potentially perform Tomcat maintenance and upgrades while Apache is still running and redirect your users to the “Site under maintenance” page.

On the other hand, if you’re deploying an internal application that is accessible only from within your company’s network, for example, then a standalone Tomcat server would be sufficient. Any low- traffic web applications can also happily run on standalone Tomcat without any performance loss. More to the point, if the performance of serving static content is the only reason you’re considering running Apache Web Server in from of Tomcat, then you should probably think twice. With Tomcat 7 connectors, the performance of static content rendering matches that of Apache Web Server.

When it comes to Apache-Tomcat integration, here are the issues we need to cover in order to make Tomcat and Apache Web Server work together:

* Configure Apache Web Server to forward selected request to Tomcat.
* Configure Apache Web Server and Tomcat to communicate with one another, using a well-known protocol.
* Configure Tomcat to understand the requests coming from Apache Web Server, and process them accordingly.

We will cover two ways of achieving all of these points. In the next, section we’ll use Apache’s *mod\_proxy* module to simply forward selected requests from Apache to Tomcat’s Http connector. After that, we’ll take a look at more advanced integration over AJP protocol using *mod\_jk* module.

**Using mod\_proxy**

As its name suggests, mod\_proxy is an Apache extension module for proxying web traffic. By configuring the mod\_proxy, you can forward specific requests to the backend Tomcat server.

mod\_proxy comes in few flavors: *mod\_proxy\_http* is used for proxying traffic over HTTP protocol, while *mod\_proxy\_ajp* (as its name suggests) uses AJP protocol to proxy requests between Apache and Tomcat. The configuration for both of these flavors is very similar, and in this chapter we will be using *mod\_proxy\_http* to demonstrate the Apache/Tomcat integration using proxy.

* Note Although *mod\_proxy\_http* and *mod\_proxy\_ajp* both work well, in our experience *mod\_proxy\_http* implementation is more robust and has fewer bugs. Therefore, if you’re deciding which proxying flavor to use with your new application, *mod\_proxy\_http* is probably a safer bet. However, if you require load balancing over AJP protocol, *mod\_proxy\_ajp* is the only flavor that supports that.

The first step is to enable *mod\_proxy\_http* in your Apache Web Server installation. The easiest way to do that is to load the http proxy module dynamically. All you need to do is edit the main Apache

configuration file (*http.conf*, located in the home directory of the Apache Web Server installation). The path to the Apache’s configuration file, depending on the Apache version, is usually */etc/apache2/http.conf*, */etc/httpd/http.conf* or */usr/local/apache2/http.conf* (Depending on the Apache Web Server version you are using, sometimes the name of the file is *apache2.conf*.)

Add the following lines to the apache configuration file:

*LoadModule proxy\_module modules/mod\_proxy.so*

*LoadModule proxy\_http\_module modules/mod\_proxy\_http.so*

* Note The \*.so extension is typical for UNIX-flavored operating systems (UNIX, Linux-es, OSX, and so on).

LoadModule directive from previous code snippet loads the precompiled apache module “*proxy\_http\_module*”, from the library file *mod\_proxy\_httpd.so*, found in the module subdirectory of the Apache home installation directory. Apache Web Server distribution comes bundled with a number of modules included in the distribution. All you have to do is make sure you load the required module dynamically (using *LoadModule* directive). Proxy module itself is modular, so you’ll need to load the common functionality for all mod\_proxy flavors, which are contained in *mod\_proxy.so* library file.

Alternatively, you can compile the required module directly in the Apache Web Server, so it’s loaded automatically whenever Apache Web Server starts. This will give you some performance improvement when loading the module. However, you will lose flexibility if you want to disable the module at any given time, because recompilation of Apache would be required in that case. So, you should use the precompiled module loading only if you’re certain to need the module in question all the time. To check which modules are precompiled with your version of Apache, you can run the following command:

*[root@tomcat: ~]# httpd –l*

The results will list the modules precompiled, as you can see on Listing 10-1.

***Listing 10-1.*** *Checking the List of Precompiled Modules for Apache Web Server*

*[root@tomcat: ~]#* httpd -l

Compiled in modules:

core.c prefork.c http\_core.c mod\_so.c

Finally, after the proxy module is enabled, you’ll need to restart Apache for the changes to take effect. You can use the following command to restart Apache Web Server on Linux:

*[root@tomcat: ~]# apachectl restart*

If you made any typos or any other errors in your configuration files, the Apache Web Server will not start, and you will receive an error message. If no messages are displayed, that generally means that no problems were encountered, and the server restarted successfully. Go to <http://localhost/to> confirm you can still see the Apache test page, as shown in Figure 12-1.

Now we have proxy module loaded, and we can proceed with configuring Apache Web Server to forward specific requests to the Tomcat servlet container. First, let’s add a new *VirtualHost* directive to the Apache Web Server configuration. We are going to create a specific virtual host, so that any changes we make are applied only to the specific virtual host that will communicate with the Tomcat backend. Next, we are going to add a *ProxyPass* directive to forward all requests coming to Apache on the /examples path to the same path, but on the Tomcat server listening to port 8080. In addition to this, we will add a *ProxyPassReverse* directive to set information flow in the opposite direction – more details about *ProxyPassReverse* will be coming shortly.

* Note Both the *ProxyPass* and *ProxyPassReverse* directives are part of the *mod\_proxy* module,

and are only available if *mod\_proxy* is loaded, as per instructions we discussed previously.

Listing 12-2 shows the VirtualHost section of the Apache configuration after all required changes are made.

***Listing 12-2.*** *Proxy Configuration Within VirtualHost Directive*

*<VirtualHost \*:80>* ***#1***

*ServerAdmin webmaster@localhost ServerName localhost*

*DocumentRoot /var/www* ***#2***

*ProxyPass /examples* [*http://localhost:8080/examples*](http://localhost:8080/examples)***#3***

*ProxyPassReverse /examples http://localhost:8080/examples* ***#4***

*</VirtualHost>*

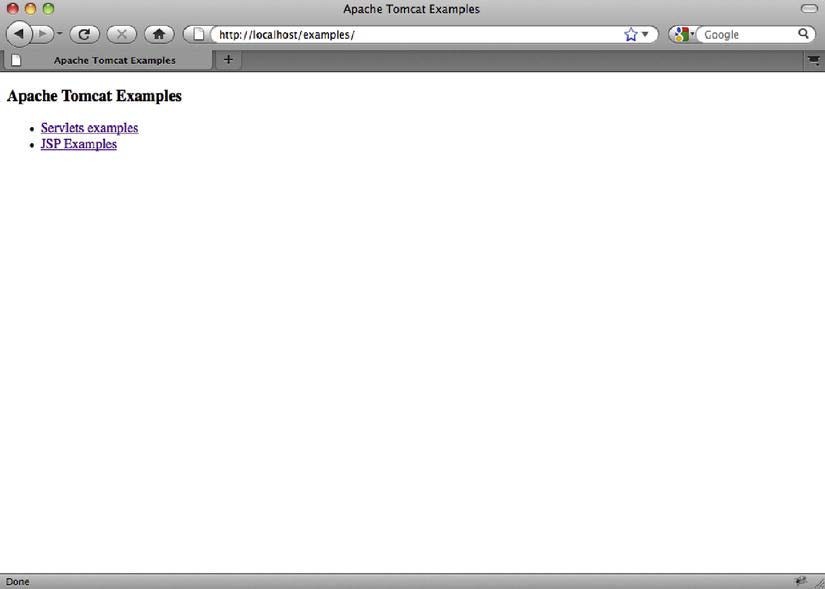
In our example, we first configured a virtual host for all requests coming on port 80 (#1), and set a few general properties for this virtual host – admin’s email address, name of the server as it will be accessed from the browser, and the root directory where all static content will be loaded from (#2). All the directives used so far (*VirtualHost*, *ServerAdmin* *ServerName*, and *DocumentRoot*) are part of the common Apache configuration, and not of major interest for our topic. If you want to learn more about Apache Web Server configuration, visit [http://httpd.apache.org/.](http://httpd.apache.org/)

Now comes the interesting part. In order for Apache Web Server proxy requests to be made to the backend Tomcat server, we must first invoke a *ProxyPass* directive (#3). What we are saying to Apache with this directive is the following: if a request comes that matches /examples URL (or any other URL that starts with examples, like /examples/index.html), forward the request to http://localhost:8080/examples (and append any additional URL parts from the original request). This will make any request to Apache on port 80 be forwarded to Tomcat – regardless if it’s a servlet or static content that needs to be served.

Finally, we invoke *ProxyPassDirective*, using exactly the same parameters as we did for the *ProxyPass* (#4). *ProxyPassReverse* takes care of any internal redirects that happen within a web application deployed on Tomcat. When such a redirect happens, the Location HTTP header will contain the redirect URL as it’s used on Tomcat (something like http://localhost:8080/example/index.html).

Because Tomcat itself is not available publicly on port 8080, when such a redirect occurs, it will result in a URL with the 8080 port appearing in the browser – which is something we’re trying to avoid when using Apache Web Server. That’s where *ProxyPassReverse* kicks in. For any such redirects, this directive modifies the Location header, using the rule specified in the directive parameters (so it will change any http://localhost:8080/examples redirects to /examples path on localhost).

Let’s now restart Apache (sudo apachectl restart) and open the JSP examples URL in the browser, but without the 8080 port: [http://localhost/examples.](http://localhost/examples) Figure 12-2 shows the familiar screen, under a different URL.



***Figure 12-2.*** *JSP examples web application page proxied by Apache Web Server*

What just happened? We typed <http://localhost/examples> URL in our browser. Invoked on default port (80), the request came to Apache Web Server, which recognized it as a proxied request (because it matched *ProxyPass* directive). The request was then forwarded to Tomcat, and served as part of the JSP examples web application we used before. All this time Tomcat is invisible to the user – the browser URL was always showing without the usual 8080 port.

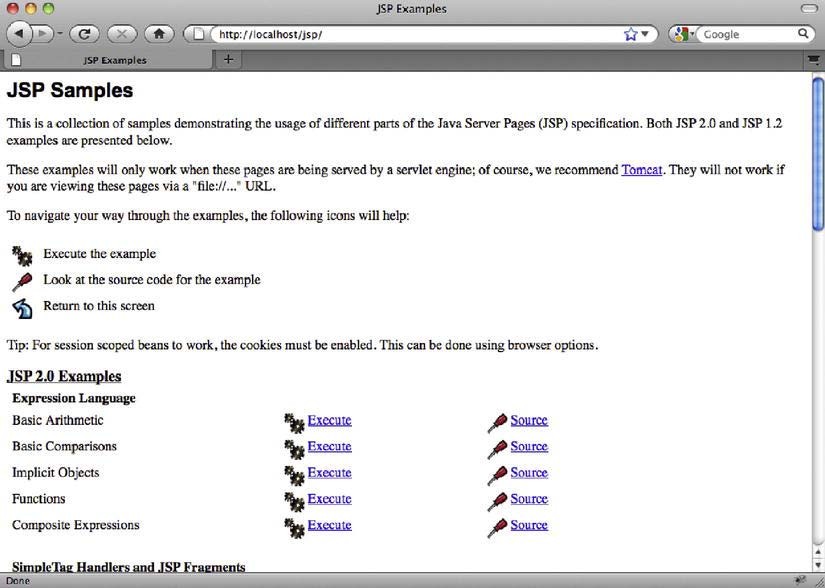
The sample shows the simple but effective way to integrate Apache Web Server with Tomcat.

Apache’s *mod\_proxy* includes a lot more useful features that can be used for advanced tuning of Tomcat integration. For example, you don’t have to proxy Tomcat requests on the same path as

they come in. If you have your servlet web application deployed on Tomcat under /example servlet context path (http://localhost:8080/examples), you may want it to be accessible in ROOT context of your web server ([www.examples.com/](http://www.examples.com/) or <http://localhost/> for local Apache testing – note no examples path). This can easily be achieved using *ProxyPass* directive as follows:

*ProxyPass / http://localhost:8080/examples*

If you now go to http:/localhost/, you will again see the familiar JSP examples home page. If you follow the links, you’ll see that the functionality works as expected from a Java web application deployed on Tomcat – but the URL in the browser address field will stay in root context. The user will be unaware that the application actually has the /examples path on backend Tomcat server. Figure 12-3 illustrates this (note the address bar of the browser).



***Figure 12-3.*** *Changing context path of Java*

You may notice that *ProxyPass* and *ProxyPassReverse* work on the entire paths of the web applications (so the /examples path used in the *ProxyPass* directive applies to all URLs that start with that given path). Apache 2.2+ versions of *mod\_proxy* include the advanced directives *ProxyPassMatch* and *ProxyPassReverseMatch*, which take regular expressions of matching URLs as parameters. For more details on these directives, you can take a look at *mod\_proxy*, version 2.2, documentation at [http://httpd.apache.org/docs/2.2/mod/mod\_proxy.html.](http://httpd.apache.org/docs/2.2/mod/mod_proxy.html)

We have seen how to use *mod\_proxy* for integration of Tomcat with Apache Web Server. In the next section, we’ll explore how can we perform the same task using the *mod\_jk* Apache module.

**Using mod\_jk**

The http proxy mechanism, which we discussed in the preceding section, used the HTTP protocol for communicating between the web server (Apache Web Server) and the application server (Apache Tomcat). Another protocol used for this communication is Apache JServ Protocol (AJP). AJP protocol carries the same data as HTTP, but in an optimized, binary format. Apache provides an implementation of AJP protocol as part of its Tomcat Connectors subproject (or mod\_jk), which is a collection of libraries containing AJP protocol implementation for integration between its popular Apache Web Server and the Tomcat application server. The current development and latest release of mod\_jk is JK 1.2. Although other versions are available for download (such as the newer but now deprecated mod\_jk2 and the older *mod\_jserv* and *mod\_webapp*), we recommend working with JK 1.2, as it is the latest and only currently supported version available. The examples in this book will be using the *mod\_jk* 1.2.X version only.

* Note The latest version of AJP protocol is AJP 1.3. Earlier AJP 1.2 is deprecated from Tomcat 3.3, and has not been supported by Tomcat since version 4.

First step in using mod\_jk is to download the library binaries, which can be found here:

[http://tomcat.apache.org/download-connectors.cgi.](http://tomcat.apache.org/download-connectors.cgi) Next, you need to load this Apache module with your Apache Web Server. You can download a ready-to-install compiled version for your operating system. However, if a version for your system is not available, you’ll have to download the source code and compile the *mod\_jk.so* file yourself. For download and building instructions, you can visit the how-to page for Tomcat Connectors project (<http://tomcat.apache.org/connectors-doc/webserver_howto/> apache.html). Once you have the mod\_jk.so file ready, copy it to Apache’s modules directory, which is usually the /modules subdirectory of Apache’s home directory.

* Note The modules libraries directory varies across operating systems and Apache Web Server versions. Depending on your combination, you should check the following paths: *APACHE\_HOME/modules*, /*etc/apache2/modules*, */usr/libexec/apache2*, */usr/lib/apache2/modules*.

Similar to the mod\_proxy configuration, the next step is to add the LoadModule directive to Apache’s main configuration file (http.conf):

*LoadModule jk\_module modules/mod\_jk.so*

Now restart Apache, If everything goes smoothly, no messages will be visible in the console, and <http://localhost/> URL will load Apache’s test page in your browser. We have mod\_jk module loaded, so we can now proceed with the configuration of AJP connector to our Tomcat instance.

AJP integration requires one additional configuration file to be included with Apache: workers.properties. This file will contain the entire required configuration for the Tomcat instance that Apache will talk to. Tomcat’s processes in the language of JK are called workers; hence the name of the file workers.properties. As the name suggests, you can configure multiple Tomcat instances to integrate with a single Apache Web Server. For the examples in this section, we will use a single Tomcat server – but multiple Tomcats will be covered later in this module.

Listing 12-3 shows the minimal workers.properties file configuration, for a single Tomcat instance.

***Listing 12-3.*** *Tomcat Process Defined in workers.properties File*

*worker.list=mytomcat #1*

*worker.mytomcat.port=8009 #2*

*worker.mytomcat.host=127.0.0.1 #3*

*worker.mytomcat.type=ajp13 #4*

First, we list the Tomcat instances we wish to configure (#1). In our example, we are going to use a single Tomcat instance, and call it mytomcat. Next, we set the port of the Tomcat’s AJP connector, as defined in server.xml file, in our case 8009 (#2). The host is the IP address of the Tomcat server (#3). We are running everything locally, so we have set host to localhost’s IP - 127.0.0.1. Finally, we set the protocol we’re going to use – we default to standard AJP 1.3 protocol (#4).

The settings in this configuration match the AJP connecter of our Tomcat installation that can be found in the CATALINA\_HOME/conf/server.xml file. Listing 10-4 shows a snippet of server.xml that contains the AJP connector configuration.

***Listing 10-4.*** *AJP Connector Configuration in Tomcat’s server.xml File*

…

*<!-- Define an AJP 1.3 Connector on port 8009 -->*

*<Connector port="8009" protocol="AJP/1.3" redirectPort="8443" />*

…

Now that we have Tomcat workers configured, we have to configure Apache Web Server to use mod\_jk and read the workers.properties configuration. As with the mod\_proxy configuration from the previous section, we will add the required configuration within virtual host directive, as Listing 12-5 shows.

***Listing 12-5.*** *Simple Apache Configuration for mod\_jk Integration*

*<VirtualHost \*:80>*

*ServerAdmin webmaster@localhost ServerName localhost DocumentRoot /var/www*

*JkWorkersFile /etc/httpd/conf/workers.properties #1 JkMount /examples/\*.jsp mytomcat #2*

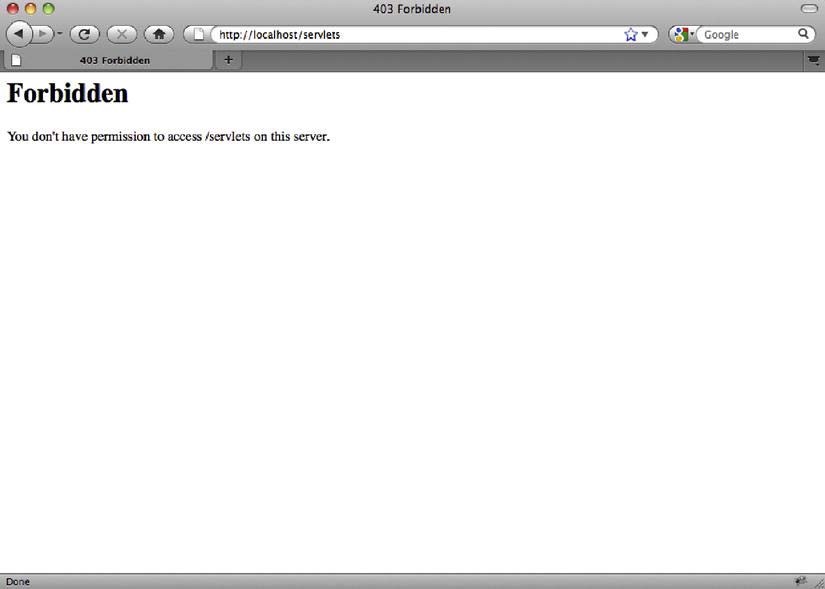
*JkUnMount /examples/servlet/\* mytomcat #3*

*</VirtualHost>*

First, using *JsWorkersFile* directive, we specify where Apache can find the *workers.properties* file (#1). After that, we “mount” the URLs that we want *mod\_jk* to forward to Tomcat (#2). The URL specified as first parameter allows the use of the wildcard character (\*) for matching extensions (\*.jsp in our example), or for matching entire sub-paths (for example, /examples/\*). The second parameter is the name of the Tomcat worker we want to forward requests to, as specified in the *workers.properties* file.

Finally, we “unmount” the /examples/servlet/\* previously mounted (#3). Again, we specify the URL to be unmounted with the wildcard as first parameter and the worker’s name as second parameter. When using JkMount and JkUnMount at the same time, you have to take care of the order, as directives will be executed in the order they appear in the configuration file.

When everything is ready, you’ll have to restart Apache Web Server. Point your browser to <http://localhost/examples> and, again, the familiar JSP examples page will be displayed – handled by Tomcat, but without port 8080 part in the address bar, as in Figure 12-2. If, however, you click on the Servlet examples link, the error page will be displayed, as shown in Figure 12-4.



***Figure 10-4.*** *An error page after trying to access the unmounted servlet examples page*

What happened here? Apache Web Server responded to request at [http://localhost/servlets/,](http://localhost/servlets/) but because of the *JkUnMount* directive (#3 in Listing 10-5), the request wasn’t forwarded to the Tomcat instance. Instead, Apache tried to resolve the request itself, but because file system access wasn’t allowed, the 403 Forbidden error page was received.

**Which Approach to Use**

Let’s discuss some of the pros and cons for both the *mod\_proxy* and *mod\_jk* approaches.

The most notable differences between http proxy and AJP protocol exist in the security and encryption areas.

You can easily encrypt the traffic between Apache and Tomcat using *mod\_http\_proxy*. All you have to do is specify https as the protocol for the Tomcat target instance, and it will just work. On the other hand, AJP protocol does not support encryption. To have encrypted communication between Apache Web Server and Tomcat backend using *mod\_jk*, you’ll have to configure that separately, using IPSec (Internet Protocol Security) or SSH tunneling, for example.

If, however, you need to have HTTPS connection details (SSL certificate details, for example) exposed to backend Tomcat instance, then *mod\_jk* has an advantage. If you configure your Apache Web Server to expose SSL details, the *mod\_jk* will make these available to Tomcat. *Mod\_proxy\_http* does not pass the HTTPS connection details to Tomcat.

Finally, the style of configuration is quite different for each of the modules. Although *mod\_jk* configuration style feels natural to Tomcat users, developers, and administrators, it seems a bit weird to the experienced Apache Web Server administrator. On the other hand, *mod\_proxy\_http* configuration looks very similar to standard Apache configuration.

* Note We have mentioned before that the mod\_proxy\_ajp flavor of mod\_proxy allows proxying using AJP protocol. Although using this module is an option, in our opinion it fares worse than the two approaches described in this chapter – mainly because of some stability issues. That is the reason we haven’t included it in our comparison.

When it comes to selecting Apache integration technology, there are many opinions. After considering the points discussed in this section, it is up to you and your requirements to pick one. But, in the end, both approaches are capable of accomplishing the job of stable integration between Tomcat application server and Apache Web Server.

**Load Balancing**

So far, we have demonstrated how to integrate Apache Tomcat with Apache Web Server using *mod\_proxy\_http* and *mod\_jk* modules. We used one instance of Tomcat server for all examples so far. But in the production environment, the amount of requests and users requires multiple application servers running so that all requests can be handled in timely manner, and to have a safety option if one of the servers crashes. One of the common approaches in situations like that is to use Apache Web Server as a load balancer for the web traffic targeted to multiple Tomcat application servers. Because Apache Web Server is quick and has only one responsibility – to forward requests to Tomcat servers, which do the actual work, such as loading data from the database and returning it to the user – it makes sense to use it as a load balancer to easily relieve the pressure on hard-worked Tomcat instances.

We will use *mod\_jk* to demonstrate load-balancing configuration. Let’s assume we have two Tomcat instances, on IPs 192.168.1.101 and 192.168.1.102, respectively, and let’s call these workers tomcat1 and tomcat2. We will configure both tomcat1 and tomcat2 as workers, but we’ll add another, third worker – let’s call it tomcat-load-balancer. The tomcat-load-balancer worker won’t have its own backed Tomcat instance, but will be solely responsible for splitting traffic and forwarding one set of requests to tomcat1 and another set of requests to tomcat2. Listing 10-6 shows the configuration required in the *worker.properties* file.

***Listing 10-6.*** *Load Balancer Workers Configuration*

worker.list=mytomcat #1

worker.tomcat1.port=8009

worker.tomcat1.host=192.168.1.101 worker. tomcat1.type=ajp13

worker.tomcat1.lbfactor=5 #2

worker.tomcat2.port=8009

worker. tomcat2.host=192.168.1.102 worker. tomcat2.type=ajp13

worker.tomcat2.lbfactor=5 #3

worker.tomcat-load-balancer.type=lb #4

worker.tomcat-load-balancer.balance\_workers=tomcat1,tomcat2 #5

worker.tomcat-load-balancer.sticky\_session=True #6

# Load balancing method can be [R]equest, [S]ession, [T]raffic, or [B]usiness

worker.tomcat-load-balancer.method=R #7

First, we set the list of Tomcat workers (#1). Note that we only list tomcat-load-balancer (our load balancer) as the worker – the tomcat1 and tomcat2 will be configured as part of load balancer, as you will see shortly.

Next, we configure our two physical Tomcat instances (#2 and #3). The configuration is very similar as before, with the addition of the *lbfactor* property. This property specifies the relative proportion of the requests that should be handled by each instance. In our example, both instances are configured to handle the same proportion of the requests (*lbfactor* has the same value, 5). The higher the value of the *lbfactor* for Tomcat instance, the more work the server will do, and vice versa. In a case of two different servers, where one has better characteristics (better CPU or more memory), it makes sense to tweak the *lbfactor* so that the more powerful instance handles more requests.

Let’s now configure the load balancer itself – or the tomcat-load-balancer worker, as we named it. The first and most important step is to set the type of the tomcat-load-balancer worker as load balancer, by setting the type property to value lb (#4). This will tell *mod\_jk* to treat tomcat-load-balancer worker not as any worker, but as special one: load-balancer. Next, we list the physical workers that are going to be used for load balancing – tomcat1 and tomcat2 – using the *balance\_workers* property (#5). And our load balancer is almost ready to go.

Before we restart Tomcat and try our load balancer, we’re going to set and explain few advanced properties.

The first of these is *sticky\_session* property, which can be set to True or False (#6). The sticky session property tells load balancer to keep requests belonging to the same session (which means the same user) forwarded to the same worker. This is particularly important if your user session contains the information that is used by Tomcat on subsequent requests. If one request of the session goes to tomcat1 and the next one goes to tomcat2, tomcat2 won’t have the required session information previously set on the tomcat1 instance. There is a complex solution for this problem, called session replication, where Tomcat instances are configured to send the session information to each other. However, for the simplicity of the configuration, we’ll use sticky sessions here.

* Note Sticky session and session replication are two main strategies for dealing with user sessions across load- balanced servers.

The *method* property (#7) specifies what method should load balancer use to determine what is the best worker to use for each request. Table 10-1 lists the allowed values for method property.

***Table 10-1.*** *Permitted values for method load balancing property*

|  |  |
| --- | --- |
| **Property** | **Value Description** |
| *[R]equest* | Number of request is used to determine the best worker – the requests will be distributed as per lbfactor for each worker. This is the default value. |
| *[S]ession* | Number of session will be determined to distribute requests. Should be used when session resources (memory) are scarce. |
| *[T]raffic* | Load balancer will monitor network traffic between workers; the one with highest bandwidth available will be used. |
| *[B]usiness* | This method will check how many requests each worker is currently serving, and pick the one with lowest load. Should be used for applications that have long request processing times. |

If you restart Apache, you can see your load balancer at work, by monitoring Tomcat logs, for example. You’ll see that requests will be distributed between configured Tomcat instances. In addition, if any of the workers stop responding (if you experience server crashes or network problems), the load balancer will be smart enough not to use it until it’s back online. Because other workers will still be available, all traffic will be redirected to the working ones – giving you time to address the failing worker problem without any downtime for your users.

**Summary**

In this chapter, we discussed how Tomcat can be integrated with Apache Web Server in production environments. We have demonstrated the integration using the mod\_proxy\_http and *mod\_jk* modules, and discussed some of the reasons for using one approach over the other. Finally, we have shown you how to use Apache Web Server as a load balancer by demonstrating load-balancing configuration to split the load of a high-usage application between multiple independent Tomcat instances.