

Chapter 5

Containers

A container is a module that processes the requests for a servlet and populates the response objects for web clients. A container is represented by the `org.apache.catalina.Container` interface and there are four types of containers: `Engine`, `Host`, `Context`, and `Wrapper`. This chapter covers `Context` and `Wrapper` and leaves `Engine` and `Host` to Chapter 13. This chapter starts with the discussion of the `Container` interface, followed by the pipelining mechanism in a container. It then looks at the `Wrapper` and `Context` interfaces. Two applications conclude this chapter by presenting a simple wrapper and a simple context respectively.

The Container Interface

A container must implement `org.apache.catalina.Container`. As you have seen in Chapter 4, you pass an instance of `Container` to the `setContainer` method of the connector, so that the connector can call the container's `invoke` method. Recall the following code from the `Bootstrap` class in the application in Chapter 4.

```
HttpConnector connector = new HttpConnector();
SimpleContainer container = new SimpleContainer();
connector.setContainer(container);
```

The first thing to note about containers in Catalina is that there are four types of containers at different conceptual levels:

- `Engine`. Represents the entire Catalina servlet engine.
- `Host`. Represents a virtual host with a number of contexts.
- `Context`. Represents a web application. A context contains one or more wrappers.
- `Wrapper`. Represents an individual servlet.

Each conceptual level above is represented by an interface in the `org.apache.catalina` package. These interfaces are `Engine`, `Host`,

Context, and Wrapper. All the four extends the Container interface. Standard implementations of the four containers are StandardEngine, StandardHost, StandardContext, and StandardWrapper, respectively, all of which are part of the `org.apache.catalina.core` package.

Figure 5.1 shows the class diagram of the Container interface and its sub-interfaces and implementations. Note that all interfaces are part of the `org.apache.catalina` package and all classes are part of the `org.apache.catalina.core` package.

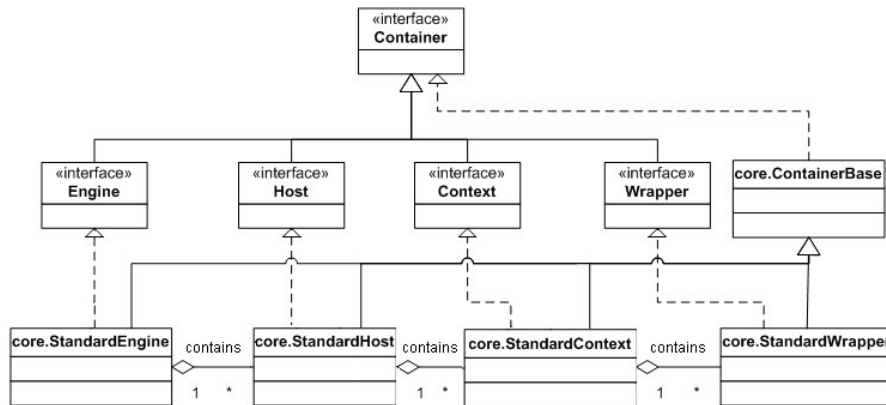


Figure 5.1: The class diagram of Container and its related types

Note

All implementation classes derive from the abstract class `ContainerBase`.

A functional Catalina deployment does not need all the four types of containers. For example, the container module in this chapter's first application consists of only a wrapper. The second application is a container module with a context and a wrapper. Neither host nor engine is needed in the applications accompanying this chapter.

A container can have zero or more child containers of the lower level. For instance, a context normally has one or more wrappers and a host can have zero or more contexts. However, a wrapper, being the lowest in the 'hierarchy', cannot contain a child container. To add a child container to a container, you use the `Container` interface's `addChild` method whose signature is as follows.

```
public void addChild(Container child);
```

To remove a child container from a container, call the `Container` interface's `removeChild` method. The `remove` method's signature is as follows.

```
public void removeChild(Container child);
```

In addition, the `Container` interface supports the finding of a child container or a collection of all child containers through the `findChild` and `findChildren` methods. The signatures of both methods are the following:

```
public Container findChild(String name);
public Container[] findChildren();
```

A container can also contain a number of support components such as `Loader`, `Logger`, `Manager`, `Realm`, and `Resources`. We will discuss these components in later chapters. One thing worth noting here is that the `Container` interface provides the `get` and `set` methods for associating itself with those components. These methods include `getLoader` and `setLoader`, `getLogger` and `setLogger`, `getManager` and `setManager`, `getRealm` and `setRealm`, and `getResources` and `setResources`.

More interestingly, the `Container` interface has been designed in such a way that at the time of deployment a Tomcat administrator can determine what a container performs by editing the configuration file (`server.xml`). This is achieved by introducing a pipeline and a set of valves in a container, which we will discuss in the next section, “`Pipelining Tasks`”.

Note

The `Container` interface in Tomcat 4 is slightly different from that in Tomcat 5. For example, in Tomcat 4 this interface has a `map` method, which no longer exists in the `Container` interface in Tomcat 5.

Pipelining Tasks

This section explains what happens when a container's `invoke` method is called by the connector. This section then discusses in the sub-sections the four related interfaces in the `org.apache.catalina` package: `Pipeline`, `Valve`, `ValveContext`, and `Contained`.

A pipeline contains tasks that the container will invoke. A valve represents a specific task. There is one basic valve in a container's pipeline, but you can add as many valves as you want. The number of valves is defined to be the number of additional valves, i.e. not including the basic valve. Interestingly, valves can be added dynamically by editing Tomcat's configuration file (`server.xml`). Figure 5.2 shows a pipeline and its valves.



Figure 5.2: Pipeline and valves

If you understand servlet filters, it is not hard to imagine how a pipeline and its valve work. A pipeline is like a filter chain and each valve is a filter. Like a filter, a valve can manipulate the request and response objects passed to it. After a valve finishes processing, it calls the next valve in the pipeline. The basic valve is always called the last.

A container can have one pipeline. When a container's `invoke` method is called, the container passes processing to its pipeline and the pipeline invokes the first valve in it, which will then invoke the next valve, and so on, until there is no more valve in the pipeline. You might imagine that you could have the following pseudo code inside the pipeline's `invoke` method:

```
// invoke each valve added to the pipeline
for (int n=0; n<valves.length; n++) {
    valve[n].invoke( ... );
}
// then, invoke the basic valve
basicValve.invoke( ... );
```

However, the Tomcat designer chose a different approach by introducing the `org.apache.catalina.ValveContext` interface. Here is how it works.

A container does not hard code what it is supposed to do when its `invoke` method is called by the connector. Instead, the container calls its pipeline's `invoke` method. The `Pipeline` interface's `invoke` method has the following signature, which is exactly the same as the `invoke` method of the `Container` interface.

```
public void invoke(Request request, Response response)
    throws IOException, ServletException;
```

Here is the implementation of the `Container` interface's `invoke` method in the `org.apache.catalina.core.ContainerBase` class.

```
public void invoke(Request request, Response response)
    throws IOException, ServletException {
    pipeline.invoke(request, response);
}
```

where `pipeline` is an instance of the `Pipeline` interface inside the container.

Now, the pipeline has to make sure that all the valves added to it as well as its basic valve must be invoked once. The pipeline does this by creating an instance of the `ValveContext` interface. The `ValveContext` is implemented as an inner class of the pipeline so that the `ValveContext` has access to all members of the pipeline. The most important method of the `ValveContext` interface is `invokeNext`:

```
public void invokeNext(Request request, Response response)
    throws IOException, ServletException
```

After creating an instance of `ValveContext`, the pipeline calls the `invokeNext` method of the `ValveContext`. The `ValveContext` will first invoke the first valve in the pipeline and the first valve will invoke the next valve before the first valve does its task. The `ValveContext` passes itself to each valve so that the valve can call the `invokeNext` method of the `ValveContext`. Here is the signature of the `invoke` method of the `Valve` interface.

```
public void invoke(Request request, Response response,
    ValveContext valveContext) throws IOException, ServletException
```

An implementation of a valve's `invoke` method will be something like the following.

```
public void invoke(Request request, Response response,
    ValveContext valveContext) throws IOException, ServletException {
    // Pass the request and response on to the next valve in our pipeline
    valveContext.invokeNext(request, response);
    // now perform what this valve is supposed to do
    ...
}
```

The `org.apache.catalina.core.StandardPipeline` class is the implementation of `Pipeline` in all containers. In Tomcat 4, this class has an inner class called `StandardPipelineValveContext` that implements the `ValveContext` interface. Listing 5.1 presents the `StandardPipelineValveContext` class.

Listing 5.1: The `StandardPipelineValveContext` class in Tomcat 4

```
protected class StandardPipelineValveContext implements ValveContext {
    protected int stage = 0;
    public String getInfo() {
        return info;
    }
    public void invokeNext(Request request, Response response)
        throws IOException, ServletException {

        int subscript = stage;
        stage = stage + 1;
```

```

// Invoke the requested Valve for the current request thread
if (subscript < valves.length) {
    valves[subscript].invoke(request, response, this);
}
else if ((subscript == valves.length) && (basic != null)) {
    basic.invoke(request, response, this);
}
else {
    throw new ServletException
        (sm.getString("standardPipeline.noValve"));
}
}
}

```

The `invokeNext` method uses `subscript` and `stage` to remember which valve is being invoked. When first invoked from the pipeline's `invoke` method, the value of `subscript` is 0 and the value of `stage` is 1. Therefore, the first valve (array index 0) is invoked. The first valve in the pipeline receives the `ValveContext` instance and invokes its `invokeNext` method. This time, the value of `subscript` is 1 so that the second valve is invoked, and so on.

When the `invokeNext` method is called from the last valve, the value of `subscript` is equal to the number of valves. As a result, the `basic` valve is invoked.

Tomcat 5 removes the `StandardPipelineValveContext` class from `StandardPipeline` and instead relies on the `org.apache.catalina.core.StandardValveContext` class, which is presented in Listing 5.2.

Listing 5.2: The `StandardValveContext` class in Tomcat 5

```

package org.apache.catalina.core;

import java.io.IOException;
import javax.servlet.ServletException;
import org.apache.catalina.Request;
import org.apache.catalina.Response;
import org.apache.catalina.Valve;
import org.apache.catalina.ValveContext;
import org.apache.catalina.util.StringManager;

public final class StandardValveContext implements ValveContext {
    protected static StringManager sm =
        StringManager.getManager(Constants.Package);
    protected String info =
        "org.apache.catalina.core.StandardValveContext/1.0";
    protected int stage = 0;
    protected Valve basic = null;
    protected Valve valves[] = null;
    public String getInfo() {
        return info;
    }
}

```

```

    }

    public final void invokeNext(Request request, Response response)
        throws IOException, ServletException {
        int subscript = stage;
        stage = stage + 1;
        // Invoke the requested Valve for the current request thread
        if (subscript < valves.length) {
            valves[subscript].invoke(request, response, this);
        }
        else if ((subscript == valves.length) && (basic != null)) {
            basic.invoke(request, response, this);
        }
        else {
            throw new ServletException
                (sm.getString("standardPipeline.noValve"));
        }
    }

    void set(Valve basic, Valve valves[]) {
        stage = 0;
        this.basic = basic;
        this.valves = valves;
    }
}

```

Can you see the similarities between the `StandardPipelineValveContext` class in Tomcat 4 and the `StandardValveContext` class in Tomcat 5?

We will now explain the `Pipeline`, `Valve`, and `ValveContext` interfaces in more detail. Also discussed is the `org.apache.catalina.Contained` interface that a valve class normally implements.

The Pipeline Interface

The first method of the `Pipeline` interface that I mentioned was the `invoke` method, which a container calls to start invoking the valves in the pipeline and the basic valve. The `Pipeline` interface allows you to add a new valve through its `addValve` method and remove a valve by calling its `removeValve` method. Finally, you use its `setBasic` method to assign a basic valve to a pipeline and its `getBasic` method to obtain the basic valve. The basic valve, which is invoked last, is responsible for processing the request and the corresponding response. The `Pipeline` interface is given in Listing 5.3.

Listing 5.3: The Pipeline interface

```

package org.apache.catalina;
import java.io.IOException;

```

```
import javax.servlet.ServletException;

public interface Pipeline {
    public Valve getBasic();
    public void setBasic(Valve valve);
    public void addValve(Valve valve);
    public Valve[] getValves();
    public void invoke(Request request, Response response)
        throws IOException, ServletException;
    public void removeValve(Valve valve);
}
```

The Valve Interface

The Valve interface represents a valve, the component responsible for processing a request. This interface has two methods: `invoke` and `getInfo`. The `invoke` method has been discussed above. The `getInfo` method returns information about the valve implementation. The Valve interface is given in Listing 5.4.

Listing 5.4: The Valve interface

```
package org.apache.catalina;
import java.io.IOException;
import javax.servlet.ServletException;

public interface Valve {
    public String getInfo();
    public void invoke(Request request, Response response,
        ValveContext context) throws IOException, ServletException;
}
```

The ValveContext Interface

This interface has two methods: the `invokeNext` method, which has been discussed above, and the `getInfo` method, which returns information about the ValveContext implementation. The ValveContext interface is given in Listing 5.5.

Listing 5.5: The ValveContext interface

```
package org.apache.catalina;
import java.io.IOException;
import javax.servlet.ServletException;

public interface ValveContext {
    public String getInfo();
    public void invokeNext(Request request, Response response)
        throws IOException, ServletException;
}
```

```
}
```

The Contained Interface

A valve class can optionally implement the `org.apache.catalina.Contained` interface. This interface specifies that the implementing class is associated with at most one container instance. The `Contained` interface is given in Listing 5.6.

Listing 5.6: The Contained interface

```
package org.apache.catalina;
public interface Contained {
    public Container getContainer();
    public void setContainer(Container container);
}
```

The Wrapper Interface

The `org.apache.catalina Wrapper` interface represents a wrapper. A wrapper is a container representing an individual servlet definition. The `Wrapper` interface extends `Container` and adds a number of methods. Implementations of `Wrapper` are responsible for managing the servlet life cycle for their underlying servlet class, i.e. calling the `init`, `service`, and `destroy` methods of the servlet. Since a wrapper is the lowest level of container, you must not add a child to it. A wrapper throws an `IllegalArgumentException` if its `addChild` method is called.

Important methods in the `Wrapper` interface include `allocate` and `load`. The `allocate` method allocates an initialized instance of the servlet the wrapper represents. The `allocate` method must also take into account whether or not the servlet implements the `javax.servlet.SingleThreadModel` interface, but we will discuss this later in Chapter 11. The `load` method loads and initializes an instance of the servlet the wrapper represents. The signatures of the `allocate` and `load` methods are as follows.

```
public javax.servlet.Servlet allocate() throws
    javax.servlet.ServletException;
public void load() throws javax.servlet.ServletException;
```

The other methods will be covered in Chapter 11 when we discuss the `org.apache.catalina.core.StandardWrapper` class.

The Context Interface

A context is a container that represents a web application. A context usually has one or more wrappers as its child containers.

Important methods include `addWrapper`, `createWrapper`, etc. This interface will be covered in more detail in Chapter 12.

The Wrapper Application

This application demonstrates how to write a minimal container module. The core class of this application is `ex05.pyrmont.core.SimpleWrapper`, an implementation of the `Wrapper` interface. The `SimpleWrapper` class contains a `Pipeline` (implemented by the `ex05.pyrmont.core.SimplePipeline` class) and uses a `Loader` (implemented by the `ex05.pyrmont.core.SimpleLoader`) to load the servlet. The `Pipeline` contains a basic valve (`ex05.pyrmont.core.SimpleWrapperValve`) and two additional valves (`ex05.pyrmont.core.ClientIPLoggerValve` and `ex05.pyrmont.core.HeaderLoggerValve`). The class diagram of the application is given in Figure 5.3.

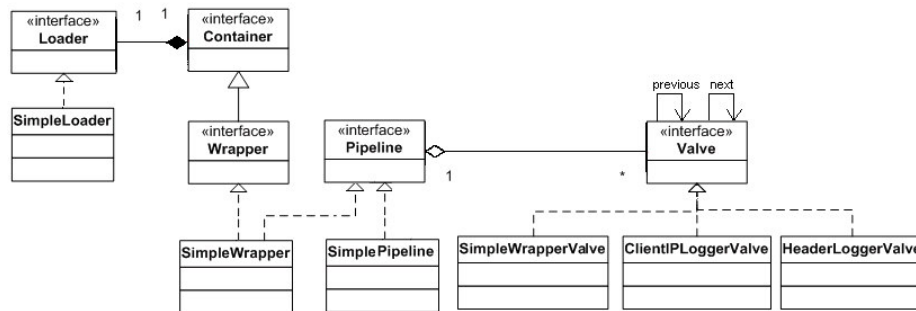


Figure 5.3: The Class Diagram of the Wrapper Application

Note

The container uses Tomcat 4's default connector.

The wrapper wraps the `ModernServlet` that you have used in the previous chapters. This application proves that you can have a servlet container consisting only of one wrapper. All classes are not fully developed, implementing only methods that must be present in the class. Let's now look at the classes in detail.

ex05.pyrmont.core.SimpleLoader

The task of loading servlet classes in a container is assigned to a `Loader` implementation. In this application, the `SimpleLoader` class is that implementation. It knows the location of the servlet class and its `getClassLoader` method returns a `java.lang.ClassLoader` instance that searches the servlet class location. The `SimpleLoader` class declares three variables. The first is `WEB_ROOT`, which points to the directory where the servlet class is to be found.

```
public static final String WEB_ROOT =
    System.getProperty("user.dir") + File.separator + "webroot";
```

The other two variables are object references of type `ClassLoader` and `Container`:

```
ClassLoader classLoader = null;
Container container = null;
```

The `SimpleLoader` class's constructor initializes the class loader so that it is ready to be returned to the `SimpleWrapper` instance.

```
public SimpleLoader() {
    try {
        URL[] urls = new URL[1];
        URLStreamHandler streamHandler = null;
        File classPath = new File(WEB_ROOT);
        String repository = (new URL("file", null,
            classPath.getCanonicalPath() + File.separator)).toString() ;
        urls[0] = new URL(null, repository, streamHandler);
        classLoader = new URLClassLoader(urls);
    }
    catch (IOException e) {
        System.out.println(e.toString() );
    }
}
```

The code in the constructor has been used to initialize class loaders in the applications in previous chapters and won't be explained again.

The `container` variable represents the container associated with this loader.

Note

Loaders will be discussed in detail in Chapter 8.

ex05.pyrmont.core.SimplePipeline

The `SimplePipeline` class implements the `org.apache.catalina.Pipeline` interface. The most important method in this class is the `invoke` method, which contains an inner class called `SimplePipelineValveContext`. The `SimplePipelineValveContext` implements the `org.apache.catalina.ValveContext` interface and has been explained in the section "Pipelining Tasks" above.

ex05.pyrmont.core.SimpleWrapper

This class implements the `org.apache.catalina Wrapper` interface and provides implementation for the `allocate` and `load` methods. Among others, this class declares the following variables:

```
private Loader loader;
protected Container parent = null;
```

The `loader` variable is a `Loader` that is used to load the servlet class. The `parent` variable represents a parent container for this wrapper. This means that this wrapper can be a child container of another container, such as a `Context`.

Pay special attention to its `getLoader` method, which is given in Listing 5.7.

Listing 5.7: The `SimpleWrapper` class's `getLoader` method

```
public Loader getLoader() {
    if (loader != null)
        return (loader);
    if (parent != null)
        return (parent.getLoader());
    return (null);
}
```

The `getLoader` method returns a `Loader` that is used to load a servlet class. If the wrapper is associated with a `Loader`, this `Loader` will be returned. If not, it will return the `Loader` of the parent container. If no parent is available, the `getLoader` method returns null.

The `SimpleWrapper` class has a pipeline and sets a basic valve for the pipeline. You do this in the `SimpleWrapper` class's constructor, given in Listing 5.8.

Listing 5.8: The `SimpleWrapper` class's constructor

```
public SimpleWrapper() {
```

```
    pipeline.setBasic(new SimpleWrapperValve());
}
```

Here, pipeline is an instance of SimplePipeline as declared in the class:

```
private SimplePipeline pipeline = new SimplePipeline(this);
```

ex05.pyrmont.core.SimpleWrapperValve

The SimpleWrapperValve class is the basic valve that is dedicated to processing the request for the SimpleWrapper class. It implements the org.apache.catalina.Valve interface and the org.apache.catalina.Contained interface. The most important method in the SimpleWrapperValve is the invoke method, given in Listing 5.9.

Listing 5.9: The SimpleWrapperValve class's invoke method

```
public void invoke(Request request, Response response,
    ValveContext valveContext)
    throws IOException, ServletException {

    SimpleWrapper wrapper = (SimpleWrapper) getContainer();
    ServletRequest sreq = request.getRequest();
    ServletResponse sres = response.getResponse();
    Servlet servlet = null;
    HttpServletRequest hreq = null;
    if (sreq instanceof HttpServletRequest)
        hreq = (HttpServletRequest) sreq;
    HttpServletResponse hres = null;
    if (sres instanceof HttpServletResponse)
        hres = (HttpServletResponse) sres;
    // Allocate a servlet instance to process this request
    try {
        servlet = wrapper.allocate();
        if (hres!=null && hreq!=null) {
            servlet.service(hreq, hres);
        }
        else {
            servlet.service(sreq, sres);
        }
    }
    catch (ServletException e) {
    }
}
```

Because SimpleWrapperValve is used as a basic valve, its invoke method does not need to call the invokeNext method of the ValveContext passed to it. The invoke method calls the allocate method of the SimpleWrapper class to obtain an instance of the servlet the wrapper represents. It then calls the servlet's service method. Notice that the basic

valve of the wrapper's pipeline invokes the servlet's `service` method, not the wrapper itself.

ex05.pyrmont.valves.ClientIPLoggerValve

The `ClientIPLoggerValve` class is a valve that prints the client's IP address to the console. This class is given in Listing 5.10.

Listing 5.10: The `ClientIPLoggerValve` class

```
package ex05.pyrmont.valves;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.ServletException;
import org.apache.catalina.Request;
import org.apache.catalina.Response;
import org.apache.catalina.Valve;
import org.apache.catalina.ValveContext;
import org.apache.catalina.Contained;
import org.apache.catalina.Container;

public class ClientIPLoggerValve implements Valve, Contained {
    protected Container container;
    public void invoke(Request request, Response response,
        ValveContext valveContext) throws IOException, ServletException {

        // Pass this request on to the next valve in our pipeline
        valveContext.invokeNext(request, response);
        System.out.println("Client IP Logger Valve");
        ServletRequest sreq = request.getRequest();
        System.out.println(sreq.getRemoteAddr());
        System.out.println("-----");
    }

    public String getInfo() {
        return null;
    }
    public Container getContainer() {
        return container;
    }
    public void setContainer(Container container) {
        this.container = container;
    }
}
```

Pay attention to the `invoke` method. The first thing the `invoke` method does is call the `invokeNext` method of the valve context to invoke the next valve in the pipeline, if any. It then prints a few lines of string including the output of the `getRemoteAddr` method of the request object.

ex05.pyrmont.valves.HeaderLoggerValve

This class is very similar to the ClientIPLoggerValve class. The HeaderLoggerValve class is a valve that prints the request header to the console. This class is given in Listing 5.11.

Listing 5.11: The HeaderLoggerValve class

```
package ex05.pyrmont.valves;

import java.io.IOException;
import java.util.Enumeration;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import org.apache.catalina.Request;
import org.apache.catalina.Response;
import org.apache.catalina.Valve;
import org.apache.catalina.ValveContext;
import org.apache.catalina.Contained;
import org.apache.catalina.Container;

public class HeaderLoggerValve implements Valve, Contained {
    protected Container container;

    public void invoke(Request request, Response response,
        ValveContext valveContext) throws IOException, ServletException {

        // Pass this request on to the next valve in our pipeline
        valveContext.invokeNext(request, response);
        System.out.println("Header Logger Valve");
        ServletRequest sreq = request.getRequest();
        if (sreq instanceof HttpServletRequest) {
            HttpServletRequest hreq = (HttpServletRequest) sreq;
            Enumeration headerNames = hreq.getHeaderNames();
            while (headerNames.hasMoreElements()) {
                String headerName = headerNames.nextElement().toString();
                String headerValue = hreq.getHeader(headerName);
                System.out.println(headerName + ":" + headerValue);
            }
        }
        else
            System.out.println("Not an HTTP Request");

        System.out.println("-----");
    }

    public String getInfo() {
        return null;
    }
    public Container getContainer() {
        return container;
    }
    public void setContainer(Container container) {
```

```

        this.container = container;
    }
}

```

Again, pay special attention to the `invoke` method. The first thing the `invoke` method does is call the `invokeNext` method of the valve context to invoke the next valve in the pipeline, if any. It then prints the values of some headers.

ex05.pyrmont.startup.Bootstrap1

The `Bootstrap1` class is used to start the application. It is given in Listing 5.12.

Listing 5.12: The `Bootstrap1` class

```

package ex05.pyrmont.startup;
import ex05.pyrmont.core.SimpleLoader;
import ex05.pyrmont.core.SimpleWrapper;
import ex05.pyrmont.valves.ClientIPLoggerValve;
import ex05.pyrmont.valves.HeaderLoggerValve;
import org.apache.catalina.Loader;
import org.apache.catalina.Pipeline;
import org.apache.catalina.Valve;
import org.apache.catalina Wrapper;
import org.apache.catalina.connector.http.HttpConnector;

public final class Bootstrap1 {
    public static void main(String[] args) {
        HttpConnector connector = new HttpConnector();
        Wrapper wrapper = new SimpleWrapper();
        wrapper.setServletClass("ModernServlet");
        Loader loader = new SimpleLoader();
        Valve valve1 = new HeaderLoggerValve();
        Valve valve2 = new ClientIPLoggerValve();

        wrapper.setLoader(loader);
        ((Pipeline) wrapper).addValve(valve1);
        ((Pipeline) wrapper).addValve(valve2);

        connector.setContainer(wrapper);

        try {
            connector.initialize();
            connector.start();

            // make the application wait until we press a key.
            System.in.read();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

After creating an instance of `HttpConnector` and `SimpleWrapper`, the main method of the `Bootstrap` class assigns `ModernServlet` to the `setServletClass` method of `SimpleWrapper`, telling the wrapper the name of the class to be loaded.

```
wrapper.setServletClass("ModernServlet");
```

It then creates a loader and two valves and sets the loader to the wrapper:

```
Loader loader = new SimpleLoader();
Valve valve1 = new HeaderLoggerValve();
Valve valve2 = new ClientIPLoggerValve();
wrapper.setLoader(loader);
```

The two valves are then added to the wrapper's pipeline.

```
((Pipeline) wrapper).addValve(valve1);
((Pipeline) wrapper).addValve(valve2);
```

Finally, the wrapper is set as the container of the connector and the connector is initialized and started.

```
connector.setContainer(wrapper);

try {
    connector.initialize();
    connector.start();
}
```

The next line allows the user to stop the application by typing Enter in the console.

```
// make the application wait until we press Enter.
System.in.read();
```

Running the Application

To run the application in Windows, from the working directory, type the following:

```
java -classpath ./lib/servlet.jar;./ ex05.pyrmont.startup.Bootstrap1
```

In Linux, you use a colon to separate two libraries.

```
java -classpath ./lib/servlet.jar:./ ex05.pyrmont.startup.Bootstrap1
```

You can invoke the servlet using the following URL:

```
http://localhost:8080
```

The browser will display the response from the `ModernServlet`. Note also that something similar to the following is printed on the console.

```
ModernServlet -- init
Client IP Logger Valve
127.0.0.1
-----
Header Logger Valve
accept:image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
application/vnd.ms-excel, application/msword, application/vnd.ms-
powerpoint, */*
accept-language:en-us
accept-encoding:gzip, deflate
user-agent:Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; .NET CLR
1.1.4322)
host:localhost:8080
connection:Keep-Alive
-----
```

The Context Application

In the first application in this chapter, you learned how to deploy a simple web application consisting of only one wrapper. This application only had one servlet. While it is possible that some applications might only need one single servlet, most web applications require more. In such applications, you need a different type of container than a wrapper. You need a context.

This second application demonstrates how to use a context with two wrappers that wrap two servlet classes. Having more than one wrapper, you need a mapper, a component that helps a container--in this case a context--select a child container that will process a particular request.

Note

A mapper can only be found in Tomcat 4. Tomcat 5 uses another approach to finding a child container.

In this application your mapper is an instance of the `ex05.pyrmont.core.SimpleContextMapper` class, which implements the `org.apache.catalina.Mapper` interface in Tomcat 4. A container can also use multiple mappers to support multiple protocols. In this case, one mapper supports one request protocol. For example, a container may have a mapper for the HTTP protocol and another mapper for the HTTPS protocol. Listing 5.13 offers the `Mapper` interface in Tomcat 4.

Listing 5.13: The `Mapper` interface

```
package org.apache.catalina;
public interface Mapper {
```

```

public Container getContainer();
public void setContainer(Container container);
public String getProtocol();
public void setProtocol(String protocol);
public Container map(Request request, boolean update);
}

```

The `getContainer` method returns the container this mapper is associated with and the `setContainer` method is used to associate a container with this mapper. The `getProtocol` method returns the protocol this mapper is responsible for and the `setProtocol` method is used to assign the name of the protocol this mapper is responsible for. The `map` method returns a child container that will process a particular request.

Figure 5.4 presents the class diagram of this application.

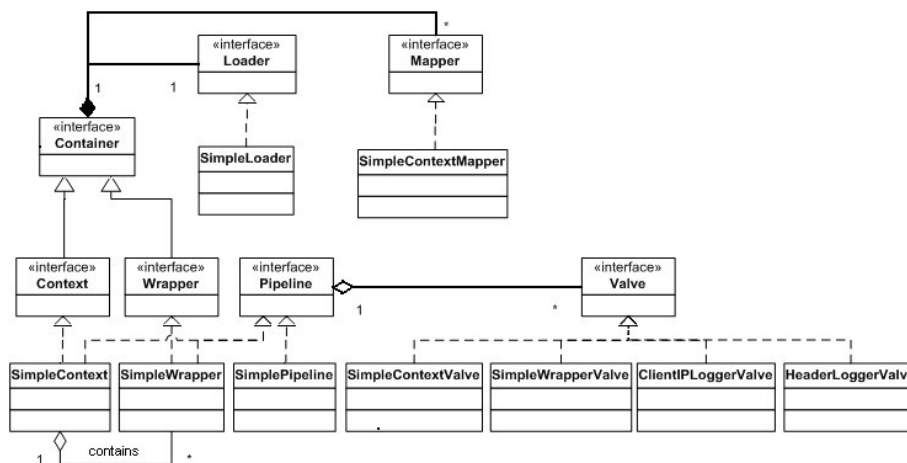


Figure 5.4: The Context application class diagram.

The `SimpleContext` class represents a context. It uses the `SimpleContextMapper` as its mapper and `SimpleContextValve` as its basic valve. Two valves, `ClientIPLoggerValve` and `HeaderLoggerValve`, are added to the context. Two wrappers, each represented by `SimpleWrapper`, are added as child containers of the context. The wrappers use `SimpleWrapperValve` as their basic valve but do not have additional valves.

The Context application uses the same loader and the two valves. However, the loader and valves are associated with the context, not a wrapper. This way,

the loader can be used by both wrappers. The context is assigned as the container for the connector. Therefore, the connector will call the `invoke` method of the context every time it receives an HTTP request. The rest is not hard to figure out if you recall our discussion above:

1. A container has a pipeline. The container's `invoke` method calls the pipeline's `invoke` method.
2. The pipeline's `invoke` method invokes all the valves added to its container and then calls its basic valve's `invoke` method.
3. In a wrapper, the basic valve is responsible to load the associated servlet class and respond to the request.
4. In a context with child containers, the basic valve uses a mapper to find a child container that is responsible for processing the request. If a child container is found, it calls the `invoke` method of the child container. It then goes back to Step 1.

Now let's see the order of processing in the implementation.

The `SimpleContext` class's `invoke` method calls the pipeline's `invoke` method.

```
public void invoke(Request request, Response response)
    throws IOException, ServletException {
    pipeline.invoke(request, response);
}
```

The pipeline is represented by the `SimplePipeline` class. Its `invoke` method is as follows.

```
public void invoke(Request request, Response response)
    throws IOException, ServletException {
    // Invoke the first Valve in this pipeline for this request
    (new SimplePipelineValveContext()).invokeNext(request, response);
}
```

As explained in the "Pipelining Tasks" section above, the code invokes all the valves added to it and then calls the basic valve's `invoke` method. In `SimpleContext`, the `SimpleContextValve` class represents the basic valve. In its `invoke` method, `SimpleContextValve` uses the context's mapper to find a wrapper:

```
// Select the Wrapper to be used for this Request
Wrapper wrapper = null;
try {
    wrapper = (Wrapper) context.map(request, true);
}
```

If a wrapper is found, its `invoke` method is called.

```
wrapper.invoke(request, response);
```

A wrapper in this application is represented by the `SimpleWrapper` class. Here is the `invoke` method of `SimpleWrapper`, which is exactly the same as the `SimpleContext` class's `invoke` method.

```
public void invoke(Request request, Response response)
    throws IOException, ServletException {
    pipeline.invoke(request, response);
}
```

The pipeline is an instance of `SimplePipeline` whose `invoke` method has been listed above. The wrappers in this application do not have valves except the basic valve, which is an instance of `SimpleWrapperValve`. The wrapper's pipeline calls the `SimpleWrapperValve` class's `invoke` method that allocates a servlet and calls its `service` method, as explained in the section "The Wrapper Application" above.

Note that the wrapper is not associated with a loader, but the context is. Therefore, the `getLoader` method of the `SimpleWrapper` class returns the parent's loader.

The four classes—`SimpleContext`, `SimpleContextValve`, `SimpleContextMapper`, and `Bootstrap2`—have not been explained in the previous section and will be discussed below.

ex05.pyrmont.core.SimpleContextValve

This class represents the basic valve for `SimpleContext`. Its most important method is the `invoke` method, given in Listing 5.14.

Listing 5.14: The `SimpleContextValve` class's `invoke` method

```
public void invoke(Request request, Response response,
    ValveContext valveContext)
    throws IOException, ServletException {
    // Validate the request and response object types
    if (!(request.getRequest() instanceof HttpServletRequest) ||
        !(response.getResponse() instanceof HttpServletResponse)) {
        return;
    }

    // Disallow any direct access to resources under WEB-INF or META-INF
    HttpServletRequest hreq = (HttpServletRequest) request.getRequest();
    String contextPath = hreq.getContextPath();
    String requestURI = ((HttpRequest) request).getDecodedRequestURI();
    String relativeURI =
        requestURI.substring(contextPath.length()).toUpperCase();
```

```

Context context = (Context) getContainer();
// Select the Wrapper to be used for this Request
Wrapper wrapper = null;
try {
    wrapper = (Wrapper) context.map(request, true);
}
catch (IllegalArgumentException e) {
    badRequest(requestURI, (HttpServletResponse)
        response.getResponse());
    return;
}
if (wrapper == null) {
    notFound(requestURI, (HttpServletResponse) response.getResponse());
    return;
}
// Ask this Wrapper to process this Request
response.setContext(context);
wrapper.invoke(request, response);
}

```

ex05.pyrmont.core.SimpleContextMapper

The SimpleContextMapper class, given in Listing 5.15, implements the org.apache.catalina.Mapper interface in Tomcat 4 and is designed to be associated with an instance of SimpleContext.

Listing 5.15: The SimpleContext class

```

package ex05.pyrmont.core;

import javax.servlet.http.HttpServletRequest;
import org.apache.catalina.Container;
import org.apache.catalina.HttpRequest;
import org.apache.catalina.Mapper;
import org.apache.catalina.Request;
import org.apache.catalina.Wrapper;

public class SimpleContextMapper implements Mapper {

    private SimpleContext context = null;
    public Container getContainer() {
        return (context);
    }
    public void setContainer(Container container) {
        if (!(container instanceof SimpleContext))
            throw new IllegalArgumentException
                ("Illegal type of container");
        context = (SimpleContext) container;
    }

    public String getProtocol() {
        return null;
    }
}

```

```

public void setProtocol(String protocol) { }

public Container map(Request request, boolean update) {
    // Identify the context-relative URI to be mapped
    String contextPath =
        ((HttpServletRequest) request.getRequest()).getContextPath();
    String requestURI = ((HttpRequest) request).getDecodedRequestURI();
    String relativeURI = requestURI.substring(contextPath.length());
    // Apply the standard request URI mapping rules from
    // the specification
    Wrapper wrapper = null;
    String servletPath = relativeURI;
    String pathInfo = null;
    String name = context.findServletMapping(relativeURI);
    if (name != null)
        wrapper = (Wrapper) context.findChild(name);
    return (wrapper);
}
}

```

The `setContainer` method throws an `IllegalArgumentException` if you pass a container that is not an instance of `SimpleContext`. The `map` method returns a child container (a wrapper) that is responsible for processing the request. The `map` method is passed two arguments, a request object and a boolean. This implementation ignores the second argument. What the method does is retrieve the context path from the request object and uses the context's `findServletMapping` method to obtain a name associated with the path. If a name is found, it uses the context's `findChild` method to get an instance of `Wrapper`.

ex05.pyrmont.core.SimpleContext

The `SimpleContext` class is the context implementation for this application. It is the main container that is assigned to the connector. However, processing of each individual servlet is performed by a wrapper. This application has two servlets, `PrimitiveServlet` and `ModernServlet`, and thus two wrappers. Each wrapper has a name. The name of the wrapper for `PrimitiveServlet` is `Primitive`, and the wrapper for `ModernServlet` is `Modern`. For `SimpleContext` to determine which wrapper to invoke for every request, you must map request URL patterns with wrappers' names. In this application, we have two URL patterns that can be used to invoke the two wrappers. The first pattern is `/Primitive`, which is mapped to the wrapper `Primitive`. The second pattern is `/Modern`, which is mapped to the wrapper `Modern`. Of course, you can use more than one pattern for a given servlet. You just need to add those patterns.

There are a number of methods from the `Container` and `Context` interfaces that `SimpleContext` must implement. Most methods are blank methods, however those methods related to mapping are given real code. These methods are as follows.

- `addServletMapping`. Adds a URL pattern/wrapper name pair. You add every pattern that can be used to invoke the wrapper with the given name.
- `findServletMapping`. Obtains a wrapper name given a URL pattern. This method is used to find which wrapper should be invoked for a particular URL pattern. If the given pattern has not yet been registered using the `addServletMapping`, the method returns `null`.
- `addMapper`. Adds a mapper to the context. The `SimpleContext` class declares the `mapper` and `mappers` variables. `mapper` is for the default mapper and `mappers` contains all mappers for the `SimpleContext` instance. The first mapper added to this context becomes the default mapper.
- `findMapper`. Finds the correct mapper. In `SimpleContext`, this returns the default mapper.
- `map`. Returns the wrapper that is responsible for processing this request.

In addition, `SimpleContext` also provides the implementation of the `addChild`, `findChild`, and `findChildren` methods. The `addChild` method is used to add a wrapper to the context, the `findChild` method is used to find a wrapper given a name, and `findChildren` returns all wrappers in the `SimpleContext` instance.

ex05.pyrmont.startup.Bootstrap2

The `Bootstrap2` class is used to start the application. This class is similar to `Bootstrap1` and is given in Listing 5.16.

Listing 5.16: The `Bootstrap2` class

```
package ex05.pyrmont.startup;
import ex05.pyrmont.core.SimpleContext;
import ex05.pyrmont.core.SimpleContextMapper;
import ex05.pyrmont.core.SimpleLoader;
import ex05.pyrmont.core.SimpleWrapper;
import ex05.pyrmont.valves.ClientIPLoggerValve;
import ex05.pyrmont.valves.HeaderLoggerValve;
import org.apache.catalina.Connector;
import org.apache.catalina.Context;
import org.apache.catalina.Loader;
import org.apache.catalina.Mapper;
```

```

import org.apache.catalina.Pipeline;
import org.apache.catalina.Valve;
import org.apache.catalina.Wrapper;
import org.apache.catalina.connector.http.HttpConnector;

public final class Bootstrap2 {
    public static void main(String[] args) {
        HttpConnector connector = new HttpConnector();
        Wrapper wrapper1 = new SimpleWrapper();
        wrapper1.setName("Primitive");
        wrapper1.setServletClass("PrimitiveServlet");
        Wrapper wrapper2 = new SimpleWrapper();
        wrapper2.setName("Modern");
        wrapper2.setServletClass("ModernServlet");

        Context context = new SimpleContext();
        context.addChild(wrapper1);
        context.addChild(wrapper2);

        Valve valve1 = new HeaderLoggerValve();
        Valve valve2 = new ClientIPLoggerValve();

        ((Pipeline) context).addValve(valve1);
        ((Pipeline) context).addValve(valve2);

        Mapper mapper = new SimpleContextMapper();
        mapper.setProtocol("http");
        context.addMapper(mapper);
        Loader loader = new SimpleLoader();
        context.setLoader(loader);
        // context.addServletMapping(pattern, name);
        context.addServletMapping("/Primitive", "Primitive");
        context.addServletMapping("/Modern", "Modern");
        connector.setContainer(context);
        try {
            connector.initialize();
            connector.start();

            // make the application wait until we press a key.
            System.in.read();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

The main method starts by instantiating the Tomcat default connector and two wrappers, `wrapper1` and `wrapper2`. These wrappers are given names `Primitive` and `Modern`. The servlet classes for `Primitive` and `Modern` are `PrimitiveServlet` and `ModernServlet`, respectively.

```

HttpConnector connector = new HttpConnector();
Wrapper wrapper1 = new SimpleWrapper();
wrapper1.setName("Primitive");

```

```

wrapper1.setServletClass("PrimitiveServlet");
Wrapper wrapper2 = new SimpleWrapper();
wrapper2.setName("Modern");
wrapper2.setServletClass("ModernServlet");

```

Then, the main method creates an instance of `SimpleContext` and adds `wrapper1` and `wrapper2` as child containers of `SimpleContext`. It also instantiates the two valves, `ClientIPLoggerValve` and `HeaderLoggerValve`, and adds them to `SimpleContext`.

```

Context context = new SimpleContext();
context.addChild(wrapper1);
context.addChild(wrapper2);

Valve valve1 = new HeaderLoggerValve();
Valve valve2 = new ClientIPLoggerValve();

((Pipeline) context).addValve(valve1);
((Pipeline) context).addValve(valve2);

```

Next, it constructs a mapper object from the `SimpleMapper` class and adds it to `SimpleContext`. This mapper is responsible for finding child containers in the context that will process HTTP requests.

```

Mapper mapper = new SimpleContextMapper();
mapper.setProtocol("http");
context.addMapper(mapper);

```

To load a servlet class, you need a loader. Here you use the `SimpleLoader` class, just like in the first application. However, instead of adding it to both wrappers, the loader is added to the context. The wrappers will find the loader using its `getLoader` method because the context is their parent.

```

Loader loader = new SimpleLoader();
context.setLoader(loader);

```

Now, it's time to add servlet mappings. You add two patterns for the two wrappers.

```

// context.addServletMapping(pattern, name);
context.addServletMapping("/Primitive", "Primitive");
context.addServletMapping("/Modern", "Modern");

```

Finally, assign the context as the container for the connector and initialize and start the connector.

```

connector.setContainer(context);
try {
    connector.initialize();
    connector.start();
}

```

Running the Application

To run the application in Windows, from the working directory, type the following:

```
java -classpath ./lib/servlet.jar;./ ex05.pyrmont.startup.Bootstrap2
```

In Linux, you use a colon to separate between libraries.

```
java -classpath ./lib/servlet.jar:./ ex05.pyrmont.startup.Bootstrap2
```

To invoke `PrimitiveServlet`, you use the following URL in your browser.

```
http://localhost:8080/Primitive
```

To invoke `ModernServlet`, you use the following URL.

```
http://localhost:8080/Modern
```

Summary

The container is the second main module after the connector. The container uses many other modules, such as Loader, Logger, Manager, etc. There are four types of containers: Engine, Host, Context, and Wrapper. A Catalina deployment does not need all four containers to be present. The two applications in this chapter show that a deployment can have one single wrapper or a context with a few wrappers.

