Overview of Servlet Technology

There are three things that a servlet container does to service a request for a servlet:

* Creating a request object and populate it with information that may be used by the invoked servlet.
  + such as parameters, headers, cookies, query string, URI, etc.
  + A request object is an instance of the javax.servlet.ServletRequest interface or the javax.servlet.http.ServletRequest interface.
* Creating a response object that the invoked servlet uses to send the response to the web client.
  + A response object is an instance of the javax.servlet.ServletResponse interface or the javax.servlet.http.ServletResponse interface.
* Invoking the service method of the servlet, passing the request and response objects.
  + Here the servlet reads the values from the request object and writes to the response object.

Tomcat is a complex system, consisting of many different components. Here we explain the internal workings of its free, open source, and most popular servlet container code-named Catalina.

Catalina

Catalina is a very sophisticated piece of software, which was elegantly designed and developed.

Catalina consist of two main modules: the connector and the container

* The connector is there to connect a request with the container.
  + Its job is to construct a request object and a response object for each HTTP request it receives.
  + It then passes processing to the container.
* The container receives the request and response objects from the connector and is responsible for invoking the servlet's service method.

Before container can invoke a servlet's service method,

* It must load the servlet,
* Authenticate the user (if required),
* Update the session for that user, etc. It's not surprising then that a container uses many different modules for processing. For example, the manager module is for processing user sessions, the loader is for loading servlet classess, etc.

**Socket-Programming:**

**Socket**: implemented by browser in servlet technology

* A socket is an endpoint of a network connection.
* A socket enables an application to read from and write to the network.
* Two software applications residing on two different computers can communicate with each other by sending and receiving byte streams over a connection.

To send a message from your application to another application, you need to know

* the IP address as well as
* the port number

of the socket of the other application.

The Socket class represents a "client" socket(browser in servlet technology), i.e. a socket that you construct whenever you want to connect to a remote server application.

* In Java, a socket is represented by the java.net.Socket class.
* Once you create an instance of the Socket class successfully, you can use it to send and receive streams of bytes.
  + To send byte streams, you must first call the **Socket class's** getOutputStream method to obtain a java.io.OutputStream object.
  + To send text to a remote application, you often want to construct a java.io.PrintWriter object **from the** OutputStream object returned.
  + To receive byte streams from the other end of the connection, you call the **Socket class's** getInputStream method that returns a java.io.InputStream.

**public** **class** SocketCls {

**psv** main(String[] args) **throws** UnknownHostException, IOException {

//constructors accepts the host name and the port number of server

Socket socket = **new** Socket("127.0.0.1", 8080);

//sending text (HTTP request) to server

**boolean** autoflush = **true**;

OutputStream os = socket.getOutputStream();

PrintWriter out = **new** PrintWriter( socket.getOutputStream(), autoflush); out.println("GET /index.jsp HTTP/1.1");

// receives the response from the server

BufferedReader in = **new** BufferedReader( **new** InputStreamReader(

socket.getInputStream() ));

**boolean** loop = **true**;

StringBuffer sb = **new** StringBuffer(8096);

**while** (loop) {

**if** ( in.ready() ) {

**int** i=0;

**while** (i!=-1) {

i = in.read();

sb.append((**char**) i);

}

loop = **false**;

}

}

// display the response to the out console

System.***out***.println(sb.toString());

socket.close();

}

}

Note that to get a proper response from the web server, you need to send an HTTP request that complies with the HTTP protocol.

**Server Socket:** In order for your application to be able to stand by all the time i.e work as server you need to use the java.net.ServerSocket class.

The role of a server socket is to wait for connection requests from clients. Once the server socket gets a connection request, it creates a Socket instance to handle the communication with the client.

To create a server socket you need to specify

* The IP address the server socket is listening on
  + Also referred as the binding address.
* Port number the server socket will be listening on.
* Backlog, which is the maximum queue length of incoming connection requests.

One of the constructors of the ServerSocket class :

public ServerSocket(int port, int backLog, InetAddress bindingAddress);

The binding address must be an instance of java.net.InetAddress.

An easy way to construct an InetAddress object is by calling its static method getByName, passing a String containing the host name.

InetAddress.getByName("127.0.0.1");

Once you have a ServerSocket instance, you can tell it to wait for an incoming connection request to the binding address by calling the ServerSocket class's accept method.

This method will only return when there is a connection request and its return value is an instance of the Socket class.

This Socket object can then be used to send and receive byte streams from the client application.

**Chapter 1: Simple Web Server**

We are going to implement here the server part not client,

Client program for this sample web server is browser.

This web server serves static resources found in the directory indicated by the WEB\_ROOT and all subdirectories under it.

Running/Testing the Application

- run the HttpServer

- open your browser and type the following in the URL: http://localhost:8080/index.html

- to stop the server, use the URL: http://localhost:8080/SHUTDOWN

**public** **class** HttpServer {

**public** **static** **final** String ***WEB\_ROOT*** =

System.*getProperty*("user.dir") + File.***separator*** + "webroot";

// shutdown command

**private** **static** **final** String ***SHUTDOWN\_COMMAND*** = "/SHUTDOWN";

// the shutdown command received

**private** **boolean** shutdown = **false**;

**public** **static** **void** main(String[] args) {

HttpServer server = **new** HttpServer();

server.await();

}

**public** **void** await() {

ServerSocket serverSocket = **new** ServerSocket(port, 3,

InetAddress.*getByName*("127.0.0.1"));

// Loop waiting for a request

**while** (!shutdown) {

socket = serverSocket.accept();

System.***out***.println("Waiting for client ...");

input = socket.getInputStream();

output = socket.getOutputStream();

// create Request object and parse

Request request = **new** Request(input);

request.parse();

// create Response object

Response response = **new** Response(output);

response.setRequest(request);

response.sendStaticResource();

// Close the socket

socket.close();

shutdown = request.getUri().equals(***SHUTDOWN\_COMMAND***);

}

}

HttpServer is typical Server Socket which waits for client to connect.

* Once there is a connection request(via typing a url in browser) it gets an instance of the Socket class.
* it gets the input and output stream of socket object
* Create the request object and gets uri by calling it's parse method.
  + Uri is relative path of the resource from webroot directory.
  + This uri is used by response object later to get the file provided in URL.
* Create the response object by passing output stream.
* Sets request object to response object.
  + Because response object has to get the uri, earlier evaluated by request object.
* Finally called the sendStaticResource of response object.
  + Which sends the content of the resource to the browser as raw data.
  + This is achieved by putting the file contents into output stream.

**public** **class** Request {

**private** InputStream input;

**private** String uri;

**public** Request(InputStream input) {

**this**.input = input;

}

/\*\*

\* The parse method reads the Request object and stores the requested resource path (URI) provided by URL in uri variable which is later made available to response object.

\*/

**public** **void** parse() {

// Read a set of characters from the socket

StringBuffer request = **new** StringBuffer(2048);

**byte**[] buffer = **new** **byte**[2048];

**int** i = input.read(buffer);

**for** (**int** j=0; j<i; j++) {

request.append((**char**) buffer[j]);

}

uri = parseUri(request.toString());

}

/\*\*

\* The parseUri method searches for the first and the second spaces in

\* the request and obtains the URI from it.

\*

\*/

**private** String parseUri(String requestString) {

**int** index1, index2;

index1 = requestString.indexOf(' ');

**if** (index1 != -1) {

index2 = requestString.indexOf(' ', index1 + 1);

**if** (index2 > index1)

**return** requestString.substring(index1 + 1, index2);

}

**return** **null**;

}

//Used by response object to get file path

**public** String getUri() {

**return** uri;

}

}

Main responsibility of Request object is

* parses the raw data in the HTTP request and
* extracts uri from it
* stores the URI in the uri variable.
  + Later response object invoke getUri method to get the URI of the HTTP request.

To understand how the parse and parseUri methods work, we need to know the structure of an HTTP request.

Here, we are only interested in the first part of the HTTP request - the request line.

A request line begins with a

* method token,
* followed by the request URI then,
* the protocol version, and
* ends with carriage-return linefeed (CRLF) characters.

Elements in a request line are separated by a space character

For Ex, request for the index.html file using the GET method is :

GET /index.html HTTP/1.1

and uri is :

/index.html

**public** **class** Response {

**private** **static** **final** **int** ***BUFFER\_SIZE*** = 1024;

Request request;

OutputStream output;

**public** Response(OutputStream output) {

**this**.output = output;

}

**public** **void** setRequest(Request request) {

**this**.request = request;

}

**public** **void** sendStaticResource() **throws** IOException {

**byte**[] bytes = **new** **byte**[***BUFFER\_SIZE***];

FileInputStream fis = **null**;

File file = **new** File(HttpServer.***WEB\_ROOT***, request.getUri());

**if** (file.exists()) {

fis = **new** FileInputStream(file);

**int** ch = fis.read(bytes, 0, ***BUFFER\_SIZE***);

**while** (ch!=-1) {

output.write(bytes, 0, ch);

ch = fis.read(bytes, 0, ***BUFFER\_SIZE***);

}

}

**else** {

// file not found

String errorMessage = "HTTP/1.1 404 File Not Found\r\n";

output.write(errorMessage.getBytes());

}

}

}

Response code-flow:

* HttpServer object pass a Request object and output stream to the Response object.
* Response object uses Request object to get the path of static resource then
* constructs a java.io.FileInputStream object by passing the File object.
* invokes the read method of the FileInputStream and writes the byte array to the OutputStream output.
* content of the static resource is sent to the browser as raw data.

**Chapter 2 : A Simple Servlet Container**

Servlet overview:

Servlet programming is made possible through the classes and interfaces in two packages:

* javax.servlet and
* javax.servlet.http.

Of those classes and interfaces, the javax.servlet.Servlet interface is of the utmost importance.

All servlets must implement this interface or extend a class that does.

The Servlet interface has five methods whose signatures are as follows.

* public void **init**(ServletConfig config) throws ServletException
* public void **service**(ServletRequest request, ServletResponse response) throws ServletException, java.io.IOException
* public void **destroy**()
* public ServletConfig **getServletConfig**()
* public java.lang.String **getServletInfo**()

Of the five methods in Servlet, the init, service, and destroy methods are the servlet's life cycle methods.

Init:

* The init method is called by the servlet container after the servlet class has been instantiated.
* The servlet container calls this method exactly once to indicate to the servlet that the servlet is being placed into service.
* The init method must complete successfully before the servlet can receive any requests.
* A servlet programmer can override this method to write initialization code that needs to run only once.
  + such as loading a database driver, initializing values, and so on.

service method

* The servlet container calls the service method of a servlet whenever there is a request for the servlet.
* The servlet container passes two objects to service method
  + javax.servlet.ServletRequest object which contains the client's HTTP request information and
  + javax.servlet.ServletResponse object which object encapsulates the servlet's response.

destroy method

* The servlet container calls the destroy method before removing a servlet instance from service.
* This normally happens when the servlet container is shut down or the servlet container needs some free memory.
* This method is called only after all threads within the servlet's service method have exited or after a timeout period has passed.
* After the servlet container has called the destroy method, it will not call the service method again on the same servlet.
* The destroy method gives the servlet an opportunity to clean up any resources that are being held, such as memory, file handles, and threads, and make sure that any persistent state is synchronized with the servlet's current state in memory.

let's examine servlet programming from a servlet container's perspective.

A fully-functional servlet container does the following for each HTTP request for a servlet:

* When the servlet is called for the first time, load the servlet class and call the servlet's init method (once only)
* For each request, construct an instance of javax.servlet.ServletRequest and an instance of javax.servlet.ServletResponse.
* Invoke the servlet's service method, passing the ServletRequest and ServletResponse objects.
* When the servlet class is shut down, call the servlet's destroy method and unload the servlet class.

This chapter explains how you can develop your own servlet container by presenting two applications.

**Application 1**

The first servlet container (Application1) runs very simple servlets and does not call the servlets' init and destroy methods. Instead, it does the following:

* Wait for HTTP requests.
* Construct a ServletRequest object and a ServletResponse object.
* If the request is for a static resource, invoke the process method of the StaticResourceProcessor instance, passing the ServletRequest and ServletResponse objects.
* If the request is for a servlet, load the servlet class and invoke the service method of the servlet, passing the ServletRequest and ServletResponse objects.

Note : In this servlet container, the servlet class is loaded every time the servlet is requested.

The first application consists of six classes:

* HttpServer : similar to the HttpServer class of previous chapter with one change
  + HttpServer1 class can serve both static resources and servlets.
* Request: it implement the javax.servlet.ServletRequest interface so provide implementations for all methods in the interface.
  + This is because request object to be passed to the servlet's service method.
* Response : implements javax.servlet.ServletResponse, so provide implementations for all the methods in the interface.
  + because response object to be passed to the servlet's service method.
* StaticResourceProcessor: used to serve requests for static resources which is achieved by response.sendStaticResource();
* ServletProcessor1 : process HTTP requests for servlets.
  + This is explained in details.
* Constants: WEB-ROOT reference is extracted from HttpServer and kept here.

Complete code - C:\Project\Tomcat\src\B\_SimpleServletContainer\Appl1

**public** **class** HttpServer1 {

**private** **static** **final** String ***SHUTDOWN\_COMMAND*** = "/SHUTDOWN";

// the shutdown command received

**private** **boolean** shutdown = **false**;

**public** **static** **void** main(String[] args) {

HttpServer1 server = **new** HttpServer1();

server.await();

}

**public** **void** await() {

ServerSocket serverSocket = **new** ServerSocket(8080, 1,

InetAddress.*getByName*("127.0.0.1"));

// Loop waiting for a request

**while** (!shutdown) {

Socket socket = serverSocket.accept();

InputStream input = socket.getInputStream();

OutputStream output = socket.getOutputStream();

// create Request object and parse

Request request = **new** Request(input);

request.parse();

// create Response object

Response response = **new** Response(output);

response.setRequest(request);

// check if this is a request for a servlet or a static resource

// a request for a servlet begins with "/servlet/"

**if** (request.getUri().startsWith("/servlet/")) {

ServletProcessor1 processor = **new** ServletProcessor1();

processor.process(request, response);

}

**else** {

StaticResourceProcessor processor = **new**

StaticResourceProcessor();

processor.process(request, response);

}

// Close the socket

socket.close();

//check if the previous URI is a shutdown command

shutdown = request.getUri().equals(***SHUTDOWN\_COMMAND***);

}

}

}

**public** **class** ServletProcessor1 {

**public** **void** process(Request request, Response response) {

String uri = request.getUri();

String servletName = uri.substring(uri.lastIndexOf("/") + 1);

URLClassLoader loader = **null**;

// create a URLClassLoader

URL[] urls = **new** URL[1];

URLStreamHandler streamHandler = **null**;

File classPath = **new** File(Constants.***WEB\_ROOT***);

// the forming of repository is taken from the createClassLoader method in

// org.apache.catalina.startup.ClassLoaderFactory

String repository = (**new** URL("file", **null**, classPath.getCanonicalPath() + File.***separator***)).toString() ;

// the code for forming the URL is taken from the addRepository method in

// org.apache.catalina.loader.StandardClassLoader class.

urls[0] = **new** URL(**null**, repository, streamHandler);

loader = **new** URLClassLoader(urls);

Class myClass = loader.loadClass(servletName);

Servlet servlet = (Servlet) myClass.newInstance();

servlet.service((ServletRequest) request, (ServletResponse) response);

}

}

**Servlet Class Loader**

Process method of ServletProcessor class loads the servlet class. To do this,

* create a instance of class loader java.net.URLClassLoader class
  + URLClassLoader is an indirect child class of the java.lang.ClassLoader class
  + use loadClass method of URLClassLoader to load a servlet class.
* Tell this class loader the location to look for the class to be loaded.
  + public URLClassLoader(URL[] urls); where urls is an array of java.net.URL objects pointing to the locations on which searches will be conducted when loading a class.
  + Any URL that ends with a / is assumed to refer to a directory.
  + Otherwise, the URL is assumed to refer to a JAR file, which will be downloaded and opened as needed.
  + For this servlet container, the class loader is directed to look in the directory pointed by Constants.WEB\_ROOT

In our application, there is only **one** location that the class loader must look, i.e. the webroot directory. Therefore, we start by creating an array of a **single** URL.

The URL class provides a number of constructors for this application, we used the same constructor used in another class in Tomcat.

The constructor has the following signature.

public URL(URL context, java.lang.String spec, URLStreamHandler hander) throws MalformedURLException

You can use this constructor by passing a specification for the second argument and null for both the first and the third arguments.

However, there is another constructor that accepts three arguments:

public URL(java.lang.String protocol, java.lang.String host, java.lang.String file) throws MalformedURLException

So to avoid ambuigity we need to tell the compiler the type of the third argument, like this.

URLStreamHandler streamHandler = null;

new URL(null, aString, streamHandler);

For the second argument, you pass a String containing the repository (the directory where servlet classes can be found),

String repository = (new URL("file", null, classPath.getCanonicalPath() + File.separator)).toString() ;

Finally, constructs the appropriate URLClassLoader instance by passing just onr url i.i. urls[0]

urls[0] = new URL(null, repository, streamHandler);

loader = new URLClassLoader(urls);

Now load a servlet class using the loadClass method:

myClass = loader.loadClass(servletName);

Next, the process method creates an instance of the servlet class loaded, downcasts it to javax.servlet.Servlet, and invokes the servlet's service method:

servlet = (Servlet) myClass.newInstance();

servlet.service((ServletRequest) request, (ServletResponse) response);

Servlet used in this ex: You can use **PrimitiveServlet** to test this container.

The service method invoked by process method is supposed to be of PrimitiveServlet. PrimitiveServlet class file can be found in the webroot directory.

**Running the Application**

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

java -classpath ./lib/servlet.jar;./ ex02.pyrmont.HttpServer1

To test the application, type the following in your URL or Address box of your browser:

http://localhost:8080/index.html or http://localhost:8080/servlet/PrimitiveServlet

When invoking PrimitiveServlet, you will see the following text in your browser:

Hello. Roses are red.

**Application 2**

There is a serious security threat in the first application. In the ServletProcessor1 class's process method because of exposing request and response class to Servlate (user).

**try** {

servlet = (Servlet) myClass.newInstance();

//upcasting is optional

servlet.service((ServletRequest) request, (ServletResponse) response);

}

This compromises security.

Servlet programmers can downcast the ServletRequest/ServletResponse instances back to Request/Response respectively and call their public methods -parse() sendStaticResource().

**public** **void** service(ServletRequest request, ServletResponse response){

Request request1 = (Request)request;

request1.parse();

}

You cannot make the parse private because it will be called from other classes.

However, these two methods are not supposed to be available from inside a servlet.

Solutions:

* Make both Request and Response classes have default access modifier, so that they cannot be used from outside the package.
* However, there is a more elegant solution: by using facade classes.
  + This is discussed here.

The security is protected by adding two façade classes: RequestFacade and ResponseFacade.

RequestFacade implements the ServletRequest interface and is instantiated by passing a Request instance that it assigns to a ServletRequest object reference in its constructor.

Implementation of each method in the ServletRequest interface invokes the corresponding method of the Request object.

Again, the ServletRequest object itself is private and cannot be accessed from outside the class.

**public** **class** RequestFacade **implements** ServletRequest {

**private** ServletRequest request = **null**;

**public** RequestFacade(Request request) {

**this**.request = request;

}

/\* implementation of the ServletRequest\*/

**public** Object getAttribute(String attribute) {

**return** request.getAttribute(attribute);

}

}

Now, Instead of upcasting the Request object to ServletRequest and passing it to the service method, we construct a RequestFacade object and pass it to the service method.

**public** **class** ServletProcessor{

**public** **void** process(Request request, Response response){

Servlet servlet = **null**;

RequestFacade requestFacade = **new** RequestFacade(request);

ResponseFacade responseFacade = **new** ResponseFacade(response);

servlet = (Servlet) myClass.newInstance();

servlet.service((ServletRequest) requestFacade,(ServletResponse)

responseFacade);

}

}

Servlet programmers can still downcast the ServletRequest instance back to RequestFacade, however they can only access the methods available in the ServletRequest interface.

Now, the parseUri method is safe.

* As it is not the part of ServletRequest interface so not implemented by RequestFacade.
* Again request object cannot be obtained as ServletRequest object which refers to the request object, is private

**Chapter 3 :** **Connector**

**The StringManager Class**

This class handles the **internationalization** of error messages in different modules in this application and in Catalina itself.

* In Tomcat error messages are useful for both system administrators and servlet programmers.
  + For servlet programmers, Tomcat sends a particular error message inside every javax.servlet.ServletException thrown so that the programmer knows what has gone wrong with his/her servlet.
* Tomcat uses properties file to store error messages so that editing them is easy.
  + Storing all error messages in one big properties file is a maintenance nightmare, so Tomcat allocates a properties file for **each** package.
  + Each properties file is handled by an instance of the org.apache.catalina.util.StringManager class.
* When Tomcat is run, there will be many instances of StringManager.
  + Each of which reads a properties file specific to a package.
* StringManager is a singleton class.

When a class in a package needs to look up an error message in that package's properties file, it will first obtain an instance of StringManager.

Each instance is stored in a Hashtable with package names as its keys.

**private** **static** Hashtable managers = **new** Hashtable();

**public** **synchronized** **static** StringManager getManager(String packageName) {

StringManager mgr = (StringManager)managers.get(packageName);

**if** (mgr == **null**) {

mgr = **new** StringManager(packageName);

managers.put(packageName, mgr);

}

**return** mgr;

}

To get an error message, use the StringManager class's getString, passing an error code.

**public** String getString(String key)

**Application Overview:**

This chapter's application consists of three modules:

* Startup : The startup module consists only of one class, Bootstrap, which starts the application.
* Connector: The connector module has classes that can be grouped into five categories:
  + The connector and its supporting class (HttpConnector and HttpProcessor).
  + The class representing HTTP requests (HttpRequest) and its supporting classes.
  + The class representing HTTP responses (HttpResponse) and its supporting classes.
  + Façade classes (HttpRequestFacade and HttpResponseFacade).
  + The Constant class.
* Core : The core module consists of two classes:
  + ServletProcessor and
  + StaticResourceProcessor.

Starting the Application : start the application from Bootstrap class, which instantiates the HttpConnector (connector) class and calls its start method.

**public** **final** **class** Bootstrap {

**public** **static** **void** main(String[] args) {

HttpConnector connector = **new** HttpConnector();

connector.start();

}

}

Stopping the Application:

Connector: HttpConnector class represents a connector responsible for creating a server socket that waits for incoming HTTP requests.

**public** **class** HttpConnector **implements** Runnable {

**boolean** stopped;

**public** **void** run() {

serverSocket = **new** ServerSocket(8080, 1,

InetAddress.*getByName*(127.0.0.1"));

**while** (!stopped) {

Socket socket = serverSocket.accept();

// Hand this socket off to an HttpProcessor

HttpProcessor processor = **new** HttpProcessor(**this**);

processor.process(socket);

}

}

**public** **void** start() {

Thread thread = **new** Thread(**this**);

thread.start();

}

}

The HttpConnector class implements java.lang.Runnable so that it can be dedicated a thread of its own. The run method contains a while loop that does the following:

* Waits for HTTP requests
* Creates an instance of HttpProcessor for each request.
* Calls the process method of the HttpProcessor, passing the socket.

The HttpProcessor class's process method receives the socket from an incoming HTTP request.

For each incoming HTTP request, it does the following:

* Create an HttpRequest object.
* Create an HttpResponse object.
* Parse the HTTP request's first line and headers and populate the HttpRequest object.
* Pass the HttpRequest and HttpResponse objects to either a ServletProcessor or a StaticResourceProcessor.

**public** **void** process(Socket socket) {

SocketInputStream input = **new**

SocketInputStream(socket.getInputStream(), 2048);

OutputStream output = socket.getOutputStream();

// create HttpRequest and HttpResponse object and parse

HttpRequest request = **new** HttpRequest(input);

HttpResponse response = **new** HttpResponse(output);

response.setRequest(request);

response.setHeader("Server", "Pyrmont Servlet Container");

parseRequest(input, output);

parseHeaders(input);

// request for a servlet or a static resource

**if** (request.getRequestURI().startsWith("/servlet/")) {

ServletProcessor processor = **new** ServletProcessor();

processor.process(request, response);

}

// request for a static resource

**else** {

StaticResourceProcessor processor = **new** StaticResourceProcessor();

processor.process(request, response);

}

socket.close();

}

}

Important points:

* The process method starts by obtaining the input stream using the SocketInputStream class that extends java.io.InputStream.
* HttpRequest instance is created by passing socket's InputStream in constructor.
* HttpResponse instance accepts socket's OutputStream.
* Assigns the HttpRequest to the HttpResponse.
* HttpProcessor class uses the org.apache.catalina.util.StringManager class for sending error messages.

Creating an HttpRequest Object

The HttpRequest class implements javax.servlet.http.HttpServletRequest. Accompanying it is a façade class called HttpRequestFacade

the main challenge in this class is to parse the HTTP request and populate the HttpRequest object.

For headers and cookies, the HttpRequest class provides the addHeader and addCookie methods

that are called from the parseHeaders method of HttpProcessor.

Parameters are parsed when they are needed, using the HttpRequest class's parseParameters method.

HTTP request parsing is a complex task. and divided into the following subsections:

* Reading the socket's input stream
* Parsing the request line
* Parsing headers
* Parsing cookies
* Obtaining parameters

Reading the Socket's Input Stream

SocketInputStream class, a copy of org.apache.catalina.connector.http.SocketInputStream. This class provides methods for obtaining not only the request line, but also the request headers.

the reason for having a SocketInputStream is for its two important methods: readRequestLine and readHeader

**Parsing the Request Line**

The process method of HttpProcessor calls the private parseRequest method to parse the request line, i.e. the first line of an HTTP request.

request line: GET /myApp/ModernServlet?userName=tarzan&password=pwd HTTP/1.1

* Method : GET
* URI : /myApp/ModernServlet
* query string (Optional): userName=tarzan&password=pwd
  + The query string can contain zero or more parameters.
* Protocol : HTTP/1.1

In servlet/JSP programming, the parameter name jsessionid is used to carry a session identifier.

if the browser's support for cookies is being turned off then programmer can opt to embed the session identifiers in query strings.

The parseRequest method parses the request line to obtain several values and assigns these values to the HttpRequest object.

via request variable, as request points to an instance of HttpRequest.

**private** **void** parseRequest(~~SocketInputStream~~ input, OutputStream output)

**throws** IOException, ServletException {

// Parse the incoming request line

input.readRequestLine(requestLine);

String method = **new** String(requestLine.method, 0, requestLine.methodEnd);

String uri = **null**;

String protocol = **new** String(requestLine.protocol, 0, requestLine.protocolEnd);

// Validate the incoming request line

// Parse any query parameters out of the request URI

**int** question = requestLine.indexOf("?");

**if** (question >= 0) {

request.setQueryString(**new** String(requestLine.uri, question + 1,

requestLine.uriEnd - question - 1));

uri = **new** String(requestLine.uri, 0, question);

}

**else** {

request.setQueryString(**null**);

uri = **new** String(requestLine.uri, 0, requestLine.uriEnd);

}

// Checking for an absolute URI (with the HTTP protocol)

// Ex:- http://www.brainysoftware.com/index.html?name=Tarzan

**if** (!uri.startsWith("/")) {

**int** pos = uri.indexOf("://");

// Parsing out protocol and host name

**if** (pos != -1) {

//skipping ://.www and finding next index of '/'

pos = uri.indexOf('/', pos + 3);

**if** (pos == -1) {

uri = "";

}

**else** {

uri = uri.substring(pos);

}

}

}

// Parse any requested session ID out of the request URI

String match = ";jsessionid=";

**int** semicolon = uri.indexOf(match);

**if** (semicolon >= 0) {

String rest = uri.substring(semicolon + match.length());

**int** semicolon2 = rest.indexOf(';');

**if** (semicolon2 >= 0) {

request.setRequestedSessionId(rest.substring(0, semicolon2));

rest = rest.substring(semicolon2);

}

**else** {

request.setRequestedSessionId(rest);

rest = "";

}

request.setRequestedSessionURL(**true**);

uri = uri.substring(0, semicolon) + rest;

}

**else** {

request.setRequestedSessionId(**null**);

request.setRequestedSessionURL(**false**);

}

// Normalize URI (using String operations at the moment)

String normalizedUri = normalize(uri);

// Set the corresponding request properties

((HttpRequest) request).setMethod(method);

request.setProtocol(protocol);

**if** (normalizedUri != **null**) {

((HttpRequest) request).setRequestURI(normalizedUri);

}

**else** {

((HttpRequest) request).setRequestURI(uri);

}

**if** (normalizedUri == **null**) {

**throw** **new** ServletException("Invalid URI: " + uri + "'");

}

}

**Parsing Headers** : An HTTP header is represented by the HttpHeader class.

* You can construct an HttpHeader instance by using its class's no-argument constructor.
* Once you have an HttpHeader instance, you can pass it to the readHeader method of SocketInputStream.
  + If there is a header to read, the readHeader method will populate the HttpHeader object accordingly.
  + If there is no more header to read, both nameEnd and valueEnd fields of the HttpHeader instance will be zero.
* To obtain the header name and value, use the following:
  + String name = new String(header.name, 0, header.nameEnd);
  + String value = new String(header.value, 0, header.valueEnd);

Code-Flow

* The parseHeaders method contains a while loop that keeps reading headers from the SocketInputStream until there is no more header. The loop starts by constructing an HttpHeader instance and passing it to the SocketInputStream class's readHeader:
* you can test whether or not there is a next header to be read from the input stream by testing the nameEnd and valueEnd fields of the HttpHeader instance:
* If there is a next header, the header name and value can then be retrieved:
* Once you get the header name and value, you add it to the headers HashMap in the HttpRequest object
  + request.addHeader(name, value);
* Some headers also require the setting of some properties. For instance,
  + the value of the content-length header is to be returned when the servlet calls the getContentLength method of javax.servlet.ServletRequest, and
  + the cookie header contains cookies to be added to the cookie collection.

**Parsing Cookies**

Cookies are sent by a browser as an HTTP request header.

* Such a header has the name "cookie" and the value is the cookie name/value pair(s).
* cookie header containing two cookies: userName and password.
  + Cookie: userName=budi; password=pwd;

Cookie parsing is done using the parseCookieHeader method of the org.apache.catalina.util.RequestUtil class.

This method accepts the cookie header and returns an array of javax.servlet.http.Cookie.

The number of elements in the array is the same as the number of cookie name/value pairs in the header.

**Obtaining Parameters**

* If the user requested the servlet using the GET method, all parameters are on the query string.
* If the POST method is used, you may find some in the request body too.
* All the name/value pairs are stored in a HashMap.
* Servlet programmers can obtain the parameters as a Map (by calling getParameterMap of HttpServletRequest) and the parameter name/value.

Catch: Servlet programmers are not allowed to change parameter values. Therefore, a special HashMap is used: org.apache.catalina.util.ParameterMap which is similar to unmodifiable(immutable) Map.

The ParameterMap class extends java.util.HashMap and employs a boolean called locked. The name/value pairs can only be added, updated or removed if locked is false. Otherwise, an IllegalStateException is thrown. Reading the values, however, can be done any time.

**public** **final** **class** ParameterMap **extends** HashMap {

**public** ParameterMap() {

**super** ();

}

**private** **boolean** locked = **false**;

**public** **boolean** isLocked() {

**return** (**this**.locked);

}

**public** **void** setLocked(**boolean** locked) {

**this**.locked = locked;

}

**public** Object put(Object key, Object value) {

**if** (locked)

**throw** **new** IllegalStateException (sm.getString("parameterMap.locked"));

**return** (**super**.put(key, value));

}

}

You don't parse the query string or HTTP request body to get parameters until the servlet needs to read one or all of them by calling the getParameter, getParameterMap, getParameterNames, or getParameterValues methods of javax.servlet.http.HttpServletRequest.

The parameters only needs to be parsed once because if the parameters are to be found in the request body, parameter parsing causes the SocketInputStream to reach the end of its byte stream.

The HttpRequest class employs a boolean called parsed to indicate whether or not parsing has been done.

parseParameters method Steps:

* method starts by checking the parsed boolean, which is true if parsing has been done before.
* the parseParameters method creates a ParameterMap called results and points it to parameters.
* It creates a new ParameterMap if parameters is null.
* opens the parameterMap's lock to enable writing to it.
* checks the encoding and assigns a default encoding if the encoding is null.
* parseParameters method tries the query string. Parsing parameters is done using the parseParameters method of org.apache.Catalina.util.RequestUtil.
* tries to see if the HTTP request body contains parameters. This happens if the user sends the request using the POST method, the content length is greater than zero, and the content type is application
* Finally, the parseParameters method locks the ParameterMap back, sets parsed to true and assigns results to parameters.

**Chapter 4: Tomcat Connector**

**HTTP 1.1 New Features**

Followings are three new features of HTTP 1.1. Understanding them is crucial to understanding how the default connector processes HTTP requests.

1. **Persistent Connections**

Prior to HTTP 1.1(1.0, 0.9), whenever a browser connected to a web server, **the connection was closed** by the server right after the requested resource was sent.

However, an Internet page can contain other resources, such as image files, applets, etc.

If the page and all resources it references are downloaded using **different connections**, the process will be very slow.

As establishing and tearing down HTTP connections are expensive operations.

That's why HTTP 1.1 introduced persistent connections.

With a persistent connection, when a page is downloaded, the server **waits** for the web client **to request all resources referenced** by the page, Instead of closing the connection straight away.

This way, the page and referenced resources can be downloaded using the **same** connection.

This saves a lot of work and time for the web server, client, and the network.

The persistent connection is the default connection of HTTP 1.1.

Also, to make it explicit, a browser can send the request header connection with the value keep-alive.

connection: keep-alive

1. **Chunked Encoding**

The consequence of establishing a persistent connection is that the server can send byte streams from multiple resources, and the client can send multiple requests using the same connection.

As a result, the sender must send the **content length header** of each request or response so that the recipient would know how to interpret the bytes. Often the sender does not know how many bytes it will send.

For example, a servlet container can start sending the response when the first few bytes become available and not wait until all of them ready.

So there must be a way to tell the recipient how to interpret the byte stream in the case that the content-length header cannot be known earlier.

Even without having to send multiple requests or many responses, a server or a client does not necessarily know how much data it will send.

In HTTP 1.0 - a server could just leave out the content-length header and keep writing to the connection. When it was finished, it would simply close the connection.

In this case, the client would keep reading until it got a -1 as an indication that the end of file had been reached.

HTTP 1.1 employs a special header called transfer-encoding to indicate that the byte stream will be sent in chunks.

For every chunk, the length (in hexadecimal) followed by CR/LF is sent prior to the data.

End of transaction is marked with a zero length chunk.

Suppose you want to send the 38 bytes[***I'm as helpless as a kitten up a tree***.] in 2 chunks, the first with the length of 29 and the second 9.

**1D**\r\n

I'm as helpless as a kitten u

**9**\r\n

p a tree.

**0**\r\n

1. **Use of the 100 (Continue) Status**

It would be a waste if the client sent the long body just to find out the server turned it down.

HTTP 1.1 clients send the **Expect: 100-continue** header to the server before sending the request body and wait for acknowledgement from the server.

Upon receipt of the **Expect: 100-continue** header, the server responds with the following **100-continue** header if it is willing to or can process the request, followed by two pairs of CRLF characters.

HTTP/1.1 100 Continue

**Chapter 6: Lifecycle**

Catalina consists of many components.

When Catalina is started, these components need to be started as well.

When Catalina is stopped, these components must also be given a chance to do a clean-up.

For example, when the container is stopped,

* It must invoke the destroy method of all loaded servlets and
* The session manager must save the session objects to secondary storage.

A consistent mechanism for starting and stopping components is achieved by implementing the org.apache.catalina.**Lifecycle** interface.

A component implementing the Lifecycle interface can also trigger one or many of the following events: BEFORE\_START\_EVENT,

An event is represented by the org.apache.catalina.**LifecycleEvent** class.

if a Catalina component can trigger events, there must be event listeners that you can write to respond to those events. A listener is represented by the org.apache.catalina.**LifecycleListener** interface.

The Lifecycle Interface

The design of Catalina allows a component to contain other components. For example, a container can contain components such as a loader, a manager, etc. A parent component is responsible for starting and stopping its child components. The design of Catalina is such that all components but one are put "in custody" of a parent component so that a bootstrap class needs only start one single component. This single start/stop mechanism is made possible through the Lifecycle interface.

**Chapter -7 : Logger**

Scope:

* The first section covers the org.apache.catalina.Logger interface, the interface that all loggers must implement.
* The second section explains the loggers in Tomcat and
* Details of the application that uses Tomcat's loggers.

Tomcat provides various loggers in the org.apache.catalina.logger package.

**Logger interface**

A logger must implement the **org.apache.catalina.Logger** interface.

Five verbosity levels are defined as public Static variables:

FATAL, ERROR, WARNING, INFORMATION, and DEBUG.

This interface provides a number of log methods that the implementing class can choose to invoke.

* The two log methods accept a verbosity level.
  + If the number passed is lower than the verbosity level set for the class's instance, the message is logged. Otherwise, the message is ignored.
* The Logger interface has the getContainer and setContainer methods to associate a Logger instance with a container.
* It also provides the addPropertyChangeListener and removePropertyChangeListener methods to add and remove a PropertyChangeListener.

**Tomcat's Loggers**

Tomcat provides three loggers whose classes are FileLogger, SystemErrLogger, and SystemOutLogger.

These classes extend the org.apache.catalina.logger.LoggerBase class.

the LoggerBase class is an abstract class that implements the org.apache.catalina.Logger interface except the log(String msg) method.

In Tomcat 5, it also implements Lifecycle interface.

public abstract void log(String msg);

This method overload is the one that does the logging in the child classes. All the other log method overloads call this overload. Because each child class logs messages to a different destination, this method overload is left blank in the LoggerBase class.

There are child class of LoggerBase which provides the implementation of the log(String message) method overload.

* The SystemOutLogger Class
* The SystemErrLogger Class
* The FileLogger Class

**Chapter -8 : Loader**