SIP SERVLET Annotations

Day 2: Sip Servlet.
Author: Daniel Anselmo Bustos

Indice

Some Concepts to begin.

What does your Web Servers do?

A web browser lets a user request a resource. The web server gets the request, finds the resource, and returns something to the user. Sometimes that resource is an HTML page. Sometimes it's a picture. Or a sound file. Or even a PDF document. Don't matter—the client asks for the thing (resource) and the server sends it back. Figure 1.

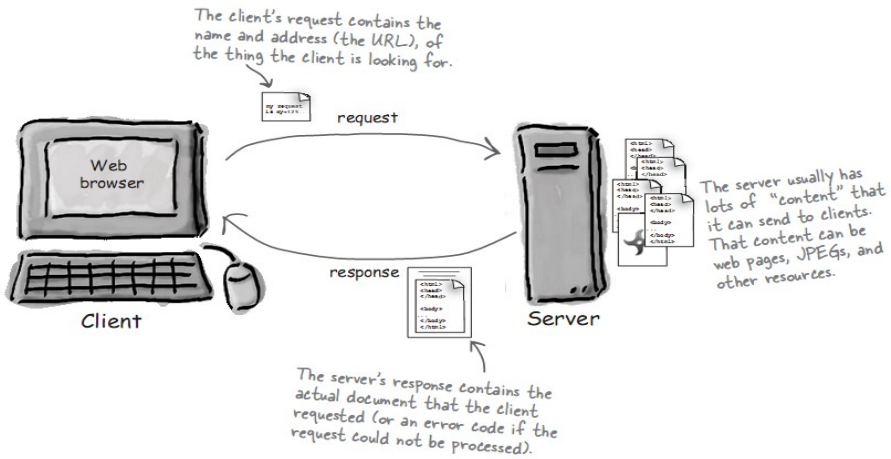


Figure 1: WEB Service Figure

Differences between HTML and HTTP.

HTML : When a server answers a request, the servers usually sends some type of content to the browser so that the browser can display it. Servers often send the browser a set of instructions written in HTML. The HTML tells the browser how to present the content to the user.
All web browsers know what to do with HTML, although sometimes an older browser might not understand part of the page that was written using newer version of HTML.

HTTP: Most of the conversations held on the web between clients and servers are held using the HTTP protocol , which allows for simple request and response conversations. The client sends an HTTP request, and the server answers with an HTTP response.
Bottom line: If you are a web server, you speak HTTP. When a web server sends an HTML page to the client, it sends it using HTTP.

HTML: Tells the browser how to display the content to the user.	HTTP is the protocol clients and servers use on the web to communicate	The server uses HTTP to send HTML to the client.

When the Client send a **Request,** the first things to make is specifies the HTTP command, this is caller METHOD that tells the server the type of actions it wants performed. The first line of the request also specifies the address of a document (URL) and the version of the HTTP protocol.

Example:

GET / intro.html HTTP/1.0

After the client sends the request, the servers processes it and send back a response. The first line of the response is a status line that specifies the version of the HTPP protocol the server is using.

HTTP/1.0 200 OK

The most frequents METHOD in HTTP request are GET and POST

Others methods are: HEAD, PUT, DELETE, TRACE, OPTIONS

Servlet API

Servlets use classes and interfaces from two packages:

javax.servlet and *javax.servlet.http*

The *javax.servlet* contains classes to support generic, protocol-independent servlets. These classes are extended by the classes in the javax.servlet.http to add HTTP specific functionality.

Javax indicate that the servlet API is a standard extension.

Every servlet must implement the *javax.servlet.Servlet* interface. Most servlets implement it by extending one of two special classes: *javax.servlet.GenericServlet* or *javax.servlet.http.HttpServlet,* while an HTTP servlet should subclassHttpServlet, which is itself a subclass of GenericServlet with added HTTP-specific functionality.

The Servlet doesn't have a main method.

The Service Method:

Each time the server dispatches a request to a servlet, it invokes the servlet's service() method.
A generic servlet should override its **service()** method to handle request as appropriate for the servlet. The **service()** methods accepts two parameters: a request object and a response object. The request object tells the servlet about the request, while the response object is used to return a response.

Generally the HTTP servlet doesn't overrides the service() method, instead overrides **doGet()** to handle the GET request and **doPost()** to handle POST request.

Handling HEAD Request

There is not **doHead()** method to write. Any servlet that subclasses HttpServlet and JSP is introduce Java implements the doGet() method automatically supports HEAD request.
The **service()** method of the *HttpServlet* identifies HEAD request and treats them specially. It constructs a modified *HttpServletResponse* object and passes it, along with an unchanged request, to the **doGet()** method. The **doGet()** method proceeds as normal, but only the headers it sets are returned to the client.

Container
AS we said below servlet don't have a main() method. They are under the control of another Java application called **Container**.

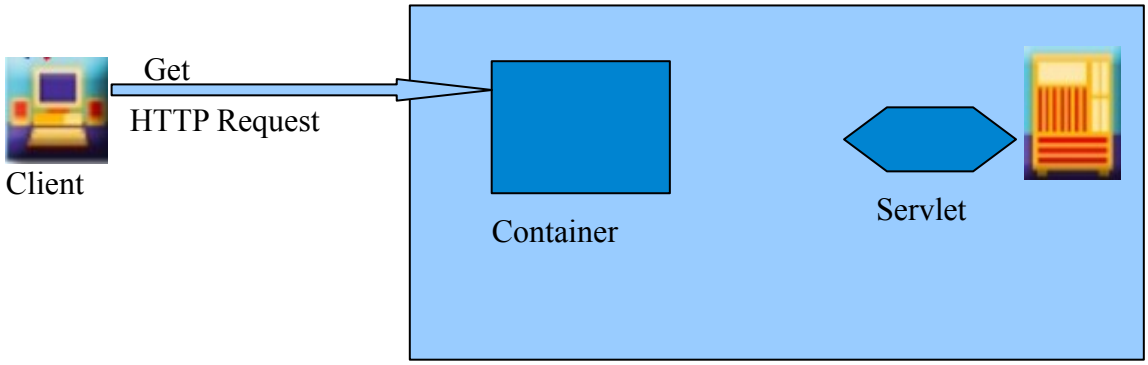
Tomcat, Glassfish are examples of Container. When your web server application (like Apache) gets a request for a servlet, the server hands the request not to the servlet itself, but to the container in which the servlet is deployed. It's the container that gives the servlet the HTTP request and response, and it's the container that calls the servlet's methods [like doPost() or do Get()].

Advantages of user Containers.

- **Communication support**: The container provides an easy way for your servlets to talk to your web server. The containers knows the protocol between the the web server and itself, so that your servlet does not have a worry about an API between, say, the apache web server and your own web application code.
- **Lifecycle Management**: The container controls the life and death of your servlets. It takes care of loading the classes, instantiating and initializing the servlets, invoking the servlet methods, and making servlet instances eligible for garbage collection. With the container in control, you don't have a worry as much about resource management.
- **Multithreading support:** The container automatically creates a new Java thread for every servlet request it receives. When the servlet's done running the HTTP service method for that client's request, the thread completes (i.e. Dies). This doesn't mean you're off the hook for thread safety. But having the server create and manage threads for multiple requests still saves you a lot of work.
- **Declarative Security**: With a container, you get to use an XML deployment descriptor to configure security without having to hard-code it into your servlet (or any other) class code.

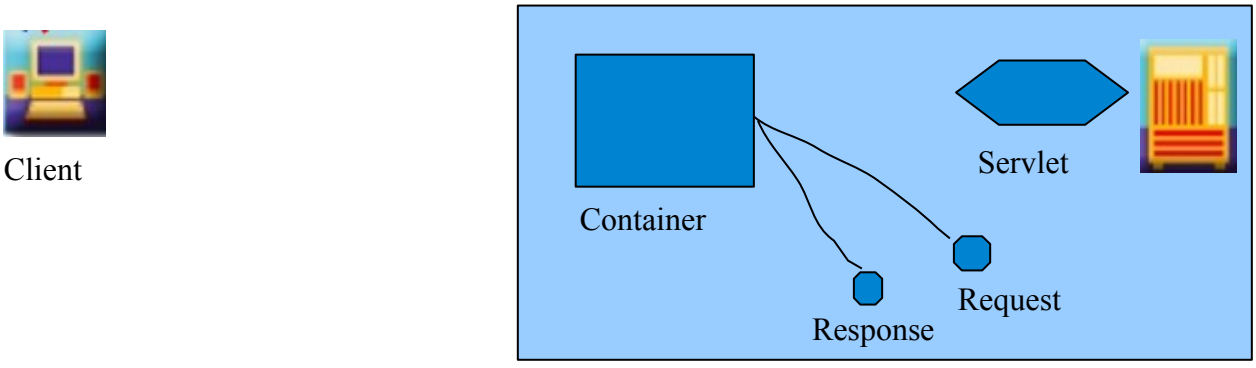
Step by Step How the container handles a request?

Step 1 : User clicks a link that has a URL to a servlet instead of a static page.



Step 2: The container "sees" that the request is for a servlet, so the container creates two objects:

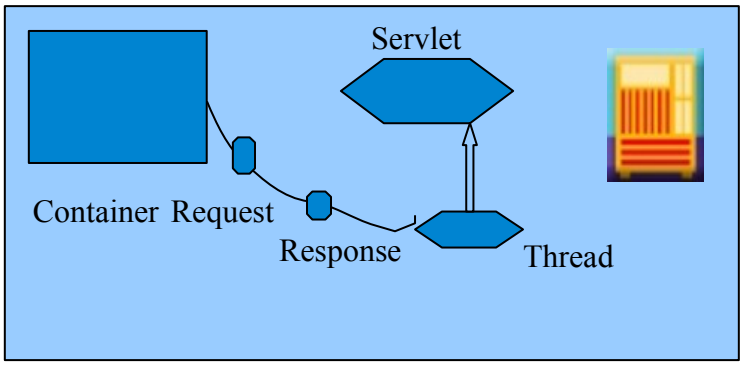
1) HttpServletResponse
2) HttpServletRequest



Step 3: The container finds the correct servlet based on the URL in the request, creates or allocates

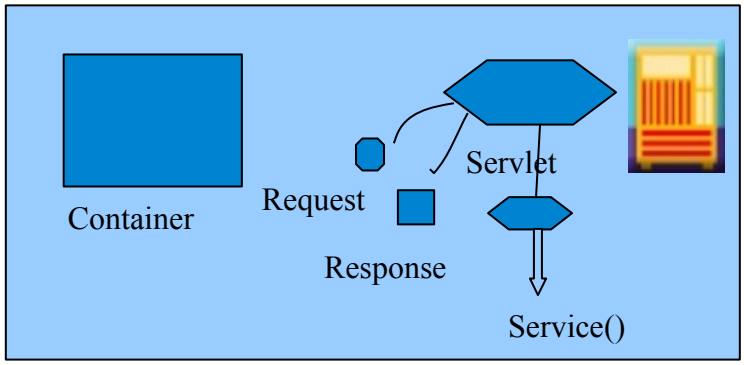
a thread for that request, and passes the request and response objects to the servlet thread.

Client



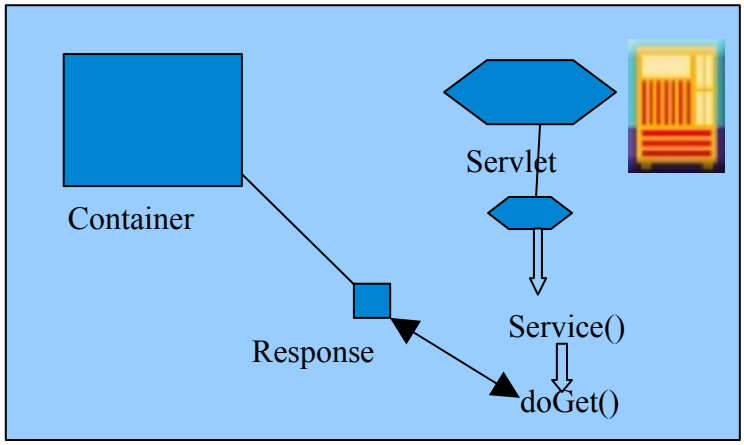
Step 4: The container calls the servlet's service() method. Depending on the type of request, the service() method calls either the doGet() or doPost() method. For this example, we'll assume the request was an HTTP GET.

Client

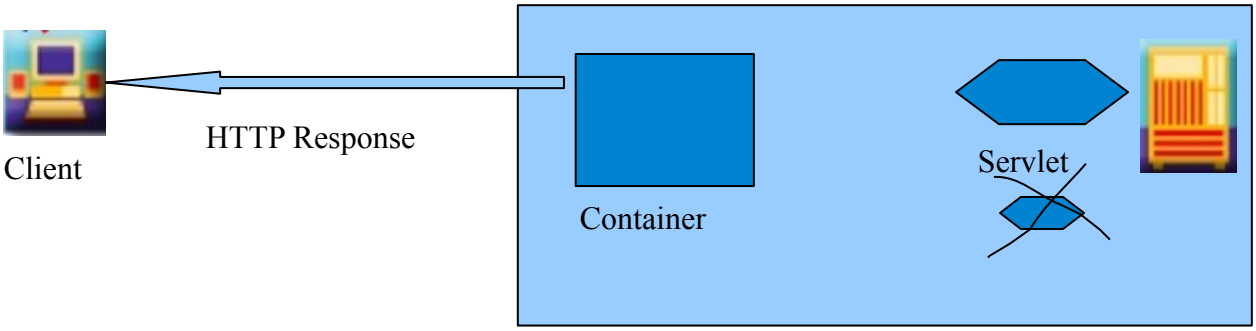


Step 5: The doGet() method generates the dynamic page and stuffs the page into the response object. Remember, the container still has a reference to the response object!

Client



Step 6: The thread completes, the container converts the response object into an HTTP response, sends it back to the client, then deletes the request and response objects.



A Servlet container also contains and manages servlet through their lifecycle.

How to the container map the servlet?
The user does something in the browser (click a link, hits the submit, enter the URL, etc.) and that something is supposed to send the request to a specific servlet.

Resuming.....

The servlet container takes the HTTP request; parses its request URI, the headers, and the body; and stores all of that data inside an object that implements the *javax.servlet.ServletRequest* interface. The response object encapsulates the response back to the client. The container then calls a method of the servlets class, passing the request and response objects. The Servlet processes the request and sends a response back to the client.

The Deployment Descriptor (DD)

When you deploy your servlet into your web Container, you'll create a fairly simple XML document called the Deployment Descriptor (DD) to tell the Container how to run your servlets and JSPs. Although you'll use the DD for more than just mapping names, you'll use two XML elements to map URLs to servlets—one to map the client-known public URL name to your own internal name, and the other to map your own internal name to a fully-qualified class name.

Key points

- one DD per web application
- <servlet-name> ties the <servlet> element to the <servlet-mapping> element.
- <servlet-class> is a Java class
- a <url-pattern> is the name the client uses for the request.

<servlet> : maps internal name to fully qualified class name
<servlet-mapping> : maps internal name to public URL name

Besides mapping URLs to actual servlets, you can use the DD to customize other aspects of your web application including security roles, error pages, tag libraries, initial configuration information.

Examples:web.xml entry (in <web-app...>...</web-app>)

Give name to servlet

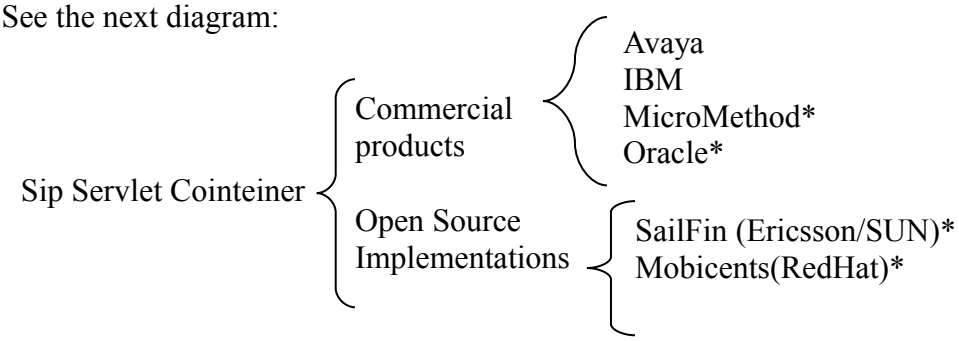
<servlet>
<servlet-name>MyName</servlet-name>
<servlet class>myPackageMyServlet</servlet class> <servlet-class>myPackage.MyServlet</servlet-class>
</servlet>
– Give address (URL mapping) to servlet
<servlet-mapping> <servlet-mapping>
<servlet-name>MyName</servlet-name>
<url-pattern>/MyAddress</url-pattern>
</servlet-mapping>
Resultant URL
– http://hostname/webappPrefix/MyAddress

And now.....

The SIP SERVLET!!!!!!!

Based in the JSR 289: SIP Servlet Specification v1.1

See the next diagram:



* Claimed JSR 289 Compliance

A SIP servlet is a java-based application component which is managed by a SIP servlet container and which performs SIP signaling like other java based components, servlet are platform independent Java classes that are compiled to platform neutral bytecode that can be loaded dynamically into and run by Java enabled SIP application server.

SIP Servlets are very similar to HTTP Servlets, and HTTP servlet developers will quickly adapt to the programming model. The service level defined by both HTTP and SIP Servlets is very similar, and you can easily design applications that support both HTTP and SIP.

Differences from HTTP Servlets

SIP was to some extent derived from HTTP and so the two protocols have much in common. Both are request-response protocols and messages have very similar structure and formats. However, in terms of providing services, there are important differences between the two protocols:

- HTTP services (including HTTP servlet applications) are almost exclusively hosted on HTTP origin servers, that is, on the Web server generating the final response to requests (as opposed to a proxy server). In contrast, an important function of SIP applications is intelligent request routing and the ability to act as a proxy is crucial in this context.

- HTTP is not a peer-to-peer protocol as is SIP and web applications never originate requests. SIP applications, on the other hand, need the ability to initiate requests of their own. An application that accepts an incoming call may have to terminate it by subsequently sending a BYE request towards the caller, a wakeup-call application may have to send the initial INVITE establishing a dialog, and a presence server application needs to be able to initiate NOTIFY requests. A back-to-back user agent (B2BUA) is a type of network based application that achieves a level of control over calls not attainable through proxying, and that requires client functionality, also. These examples demonstrate why client-side functionality is necessarily part of a SIP service infrastructure and explains the presence of such functionality in the SIP Servlet API.

According with the last point, the SIP Servlet API MUST support the following capabilities:

- generate multiple response (for example, one or more 1xx followed by a final response). See the figure 2.
- proxying requests, possibly to multiple destinations. Figure 3
- initiate requests
- receive responses as well as requests Figure 4

Figure 2: Multiple Response

Generate multiple response

The figure 2 shows an example of a response to the INVITE request. In this example, the server sends back three responses 100, 180, and 200 to the single INVITE request. To implement such sequence, in SIP Servlet, only a request is specified in a doXxx method, and an application generates and returns necessary responses in an overridden method. [2]
Currently, SIP Servlet defines the following doXxx methods:

```
protected void doInvite(SipServletRequest req);
protected void doAck(SipServletRequest req);
protected void doOptions(SipServletRequest req);
protected void doBye(SipServletRequest req);
```

```
protected void doCancel(SipServletRequest req);
protected void doSubscribe(SipServletRequest req);
protected void doNotify(SipServletRequest req);
protected void doMessage(SipServletRequest req);
protected void doInfo(SipServletRequest req);
protected void doPrack(SipServletRequest req);
```

Proxying requests, possibly to multiple destinations

Another function that is different from the HTTP protocol is “forking.” Forking is a process of proxying one request to multiple servers simultaneously (or sequentially) and used when multiple terminals (operators) are associated with one telephone number (such as in a call center).

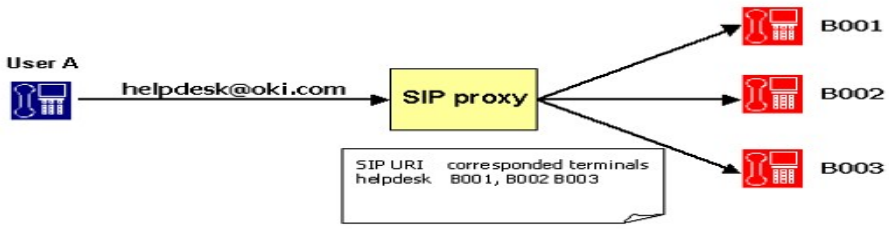


Figure 3: Proxy

Receive responses as well as requests

One of the major features of SIP is that roles of a client and server are not fixed. In HTTP, Web browsers always send HTTP requests and receive HTTP responses: They never receive HTTP requests and send HTTP responses. In SIP, however, each terminal needs to have functions of both a client and server.

For example, both of two SIP phones must call to the other and disconnect the call.

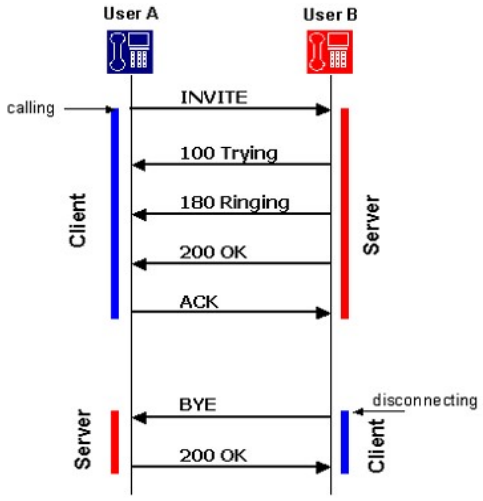


Figure 4: Receive responses and request

The Figures 4 indicates that a calling or disconnecting terminal acts as a client. In SIP, roles of a client and server can be changed in one dialog. This client function is called UAC (User Agent Client) and server function is called UAS (User Agent Server), and the terminal is called UA (User Agent). SIP Servlet defines methods to receive responses as well as requests.

Asynchronicity

Another important difference is the fact that the SIP Servlet API event model is asynchronous rather than synchronous as in the HTTP API. This means that applications are not obliged to respond to incoming requests in the upcall reporting the event. They may initiate some other action, return control to the container, and then respond to the request at some later point in time. The container relies on timeout of the application instance as a whole in order to guarantee that resources are always recovered eventually, even if an application fails to respond to the request in a timely manner.

Asynchronicity simplifies the programming of event driven services and allows an application such as a B2BUA not to hog threads while waiting for a potentially long-lived transaction to complete.

The SERVLET Interfaces

The Servlet interface is the central abstraction of the Servlet API and hence of the SIP Servlet API. All servlets implement this interface either directly or, more commonly, by extending a class that implements the interface. The generic Servlet API defines a class GenericServlet that is an implementation of the Servlet interface. The SIP Servlet API defines a class SipServlet that extends the GenericServlet interface and performs dispatching based on the type of message received. For most purposes, developers will extend SipServlet to implement their servlets.

The LifeCycle

The lifecycle is illustrated in Figure 5, in section Container , we looked at the Container's overall role in a servlet's life—it creates the request and response objects, creates or allocates a new thread for the servlet, and calls the servlet's service() method, passing the request and response references as arguments.

The servlet lifecycle is simple; there's only one main state—initialized. If the servlet isn't initialized, then it's either being initialized [running its constructor or init()method], being destroyed [running its destroy() method], or it simply does not exist.

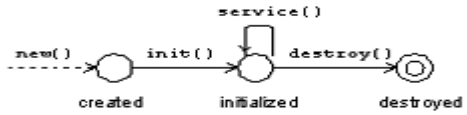


Figure 5: Servlet Lifecycle

The servlet starts life when the Container finds the servlet class file. This virtually always happens when the Container starts up. When the Container starts, it looks for deployed web apps and then starts searching for servlet class files.

The Servlet init() method is called by the container to allow the Servlet to initialize one time setup tasks that are not specific to any application instance. The Servlet is not usable until the init() method succesfully returns. A SIP Servlet in addition to receiving requests can also initiate requests (acting as a UAC) or create timers using the TimerService interface. Until the initialization of the

Servlet is complete the SIP Servlet SHOULD NOT perform any of the signaling related tasks including sending SIP messages or setting timers. A listener for SIP Servlet lifecycle if present MUST be invoked by the container. The listener defines a single method indicating that the initialization of the Servlet is now complete and it can now receive messages as well as perform any other tasks.

Processing SIP Messages

The basic Servlet interface defines a service method for handling client requests. This method is called for each message that the servlet container routes to an instance of a servlet. The handling of concurrent messages in a servlet application generally requires the developer to design servlets that can deal with multiple threads executing within the service method at a particular time. Generally, the servlet container handles concurrent requests to the same servlet by concurrent execution of the service method on different threads.
SIP servlets are invoked to handle both incoming requests and responses. In either case, the message is delivered through the service method of the Servlet interface:

```
void service(ServletRequest req, ServletResponse res)
  throws ServletException, java.io.IOException;
```

The relationship between SIP Servlet and HTTP Servlet Applications.

The directory structure used for SIP servlet applications is very similar to the one defined for HTTP servlets, in particular, the deployment descriptor, class file, libraries etc. of SIP servlet applications exits underneath the WEB-INF/ directory. This allows converged HTTP and SIP servlet applications to be packaged in a single archive file and deployed as a unit without having any part of the SIP application be interpreted as content to be published by the web server.

Elements of a SIP Application.

A SIP application may consist of the following items:

- Servlets
- Utility Classes
- Static resources and content (text and speech announcements, configuration files, etc)
- Descriptive meta information which ties all the above elements together.

Directory Structure

A SIP application exists as a structured hierarchy of directories. As mentioned above, for compatibility with the HTTP servlet archive file format, the root of this hierarchy serves as the document root for files intented to be published from a web server. This feature is only actually used for combined HTTP and SIP servlet applications.
A special directory exists within the application hierarchy named “WEB-INF”. This directory contains all things related to the application that are not in the document root of the application. For SIP only applications, everything typically resides under WEB-INF. The WEB-INF node is not part of the public document tree of the application. No file contained in the WEB-INF directory may be served directly to a client by the container. However, the contents of the WEB-INF directory are

visible to servlet code using the *getResource()* and *getResourceAsStream()* method calls on the ServletContext.

The contents of the WEB-INF directory are:

- The /WEB-INF/sip.xml deployment descriptor.
- The /WEB-INF/classes/* directory for a servlet and utility classes. The classes in this directory are available to the application classloader.
- The /WEB-INF/lib/*.jar area for Java Archive files.

Application Names

In order to identify SIP servlet applications, a new mandatory element, <app-name>, is defined under the <sip-app> element in the sip.xml deployment descriptor file of each application. The names of Sip Servlet applications must be unique within a container instance or across clustered containers under common administrative control for application selection to function properly. It is recommended that application developers follow the Java class naming convention when naming applications, e.g. "org.sipservlet.app.voicemail", to avoid naming conflicts with other developers. The container is responsible for detecting and handling naming conflict, when a new application is deployed. Though how the application is identified within the container is something internal to it and is like a symbol, it is strongly recommended that containers make use of the deployment context while identifying the applications.
Example of SIP.xml file

```
<sip-app>
  <app-name>org.sipservlet.app.voicemail</app-name>
  ...
  <servlet>
    <servlet-name>depositServlet</servlet-name>
    <servlet-class>org.sipservlet.app.voicemail.DepositServlet</servlet-class>
    ...
  </servlet>

  <servlet>
    <servlet-name>retrievalServlet</servlet-name>
    <servlet-class>org.sipservlet.app.voicemail.RetrievalServlet</servlet-class>
    ...
  </servlet>
  ...
</sip-app>
```

Example Application Directory Structure

SIP Servlet applications

/WEB-INF/sip.xml
/WEB-INF/wakeup.wav
/WEB-INF/lib/foo.jar
/WEB-INF/classes/WakeupServlet.class

HTTP Servlet applications

/wakeup.html
/register.html
/WEB-INF/sip.xml
/WEB-INF/web.xml
/WEB-INF/wakeup.wav
/WEB-INF/lib/foo.jar
/WEB-INF/classes/RegisterWakeupCall.class
/WEB-INF/classes/WakeupServlet.class

SERVLET CONTEXT

The ServletContext defines a servlet's view of the SIP application within which the servlet is running. [3]

SIP Factory:
The SipFactory interface is used by servlets to create instances of various interfaces:

- requests: the createRequest methods create instances of the SipServletRequest interface and is used by UAC applications when creating new requests that do not belong to existing SipSessions.
- address objects: ability to create URI, SipURI, Address and Parameterable instances.
- application sessions: ability to create new application sessions.

All servlet containers MUST make an instance of the javax.servlet.sip.SipFactory interface available to servlets via the context attribute of the same name, javax.servlet.sip.SipFactory.

SIP annotations.

SIPServlet1.1 defines four annotations that may be used in SIP applications. Using these annotations simplifies SIP application development by making the sip.xml deployment descriptor optional.
The figure 6, shows the four annotations:

Annotation	Description
`@SipServlet`	Marks the class as a SIP servlet.
`@SipListener`	Marks the class as an implementation class of one of the SIP listeners.
`@SipApplication`	An application-level class to define a collection of SIP servlets.
`@SipApplicationKey`	Associates an incoming request and SIP session with a particular `SipApplicationSession`.

Figure 6: SIP annotation defined by SIP servlet 1.1

Using the @SipServlet Annotation
The javax.servlet.sip.annotation. SipServlet class-level annotation is used to mark the class as a SIP servlet.

EXAMPLE1–1 Example of the @SipServletAnnotation (taked from [3])

```
@SipServlet
public class MyServlet extends SipServlet {
...
}
```

Using the @SipListener Annotation
The javax.servlet.sip.annotation.SipListener class-level annotation is used to mark the class as an implementation class of one of the SIP event listener interfaces.

Using the @SipApplication Annotation
The javax.servlet.sip.annotation.SipApplication application-level annotation is used to define a collection of SIP servlets and SIP listeners with a common configuration.
@SipApplication is annotated at the package level, and all SIP servlets or listeners within the package are part of the defined SIP application unless the SIP servlet or listener explicitly sets the application Name element in the @SipServlet or @SipListener annotation, respectively.
@SipApplication should be annotated either in a package-info.java filein a package hierarchy, or before the package definition in a particular source file.

Example of @SipApplication Annotation in a package-info.java File

```
@SipApplication(name="MySipApplication")
packagecom.example.sip;
```

Using the @SipApplicationKey Annotation
The javax.servlet.sip.annotation.SipApplicatonKey method-level annotation associates an incoming request with a particular SIpApplicationSession instance.
The method annotated by @SipApplicationKey must:

- Be public.
- Be static.
- Return a String.
- Define a single argument of type SipServletRequest.
- Not modify the passed-in SipServletRequest object.

SIP Listeners

SIP application listeners are Java servlet application listeners that listen for SIP-specific events. SIP applications implement the SIP event listener interfaces by marking the implementation class with a javax.servlet.sip.annotation.SipListener annotation.

Example of @SipListener

```
@SipListener
public class MyListener implements SipServletListener{
...
}
```

Sip servlet classes may also implement the SIP event listener interfaces.

Example of SIP Listener in SIP Servlet Class

@SipListener
@SipServlet
public class MySipServlet extends SipServlet implements SipServletListener{
...
}

SIP Servlet Listeners

The following SIP servlet listeners, in package javax.servlet.sip, are available to SIP servlet developers:

Listener	Description
`SipServletListener`	Implementations of `SipServletListener` receive notifications on initialization of `SipServlet` instances. See the SIP Servlet 1.1 Javadocs for more information.

SIP Application Session Listeners
The following SIP application listeners, in package javax.servlet.sip, are available to SIP servlet developers:

Listener	Description
`SipApplicationSessionListener`	Implementations of `SipApplicationSessionListener` receive notifications when `SipApplicationSession` instances have been created, destroyed, timed out, or are ready to be invalidated. See the SIP Servlet 1.1 Javadocs for more information.
`SipApplicationSessionAttributeListener`	Implementations of `SipApplicationSessionAttributeListener` receive notifications when attributes are added, removed, or modified in `SipApplicationSession` instances. See the SIP Servlet 1.1 Javadocs for more information.
`SipApplicationSessionBindingListener`	Session attributes that implement `SipApplicationSessionBindingListener` receive notifications when they are bound or unbound to `SipApplicationSession` instances. See the SIP Servlet 1.1 Javadocs for more information.
`SipApplicationSessionActivationListener`	Implementations of `SipApplicationSessionActivationListener` receive notifications when `SipApplicationSession` instances are activated or passivated. See the SIP Servlet 1.1 Javadocs for more information.

Examples of SIP SERVLET.

Click to Dial File:click to dial
Game Sip Servlet File: Game.zip

For more information, please contact me: dansebus@gmail.com

Bibliography

[1] Java Servlet Programming. Jason Hunter with William Crawford. First Edition O'Reilly

[2] Oracle Communications Converged Application Server Developing SIP Applications, Release 4.0

[3] SIP Servlet specification Version 1.1 JSR 289 Expert Group.