The Java Servlet API

The secret's out: Java (tm) isn't just for programming client side applets which run in web browsers, or for writing Internet applications. The simple, flexible *Servlet API* brings the power of Java to your servers, too.

Java is a great platform for writing the server side of your web-based application. The same features which make Java a better platform for writing client applications make it better for writing servers. Your server applications will benefit from its type safety and other rapid development features even more than your client applications did, because multithreading support is built into the Java platform.

Java makes it easy to develop and deploy all parts of a professional, maintainable distributed system application. The **Servlet API** provides you the fastest way to start using Java Server technology in your networked applications. You can start with applications which involve clients and a single server, and gradually create multi-tier enterprise applications which integrate the power and flexibility of Java throughout your existing network ... because Java Servlets run on the software and hardware you've already installed.

Overview of Java Servlets

Servlets are protocol- and platform-independent server side components, written in Java, which dynamically extend Java-enabled servers. They provide a general framework for services built using the request-response paradigm. Their initial use is to provide secure web-based access to data which is presented using HTML web pages, interactively viewing or modifying that data using dynamic web page generation techniques.

Since servlets run inside servers, they do not need a graphical user interface. Otherwise, they are the server side counterpart to *applets*: they are Java application components which are downloaded, on demand, to the part of the system which needs them.

Servlets are most often provided by organizations which provide customized multi-user services to their customer bases. However, servlets are also flexible enough to support standardized services such as serving static web pages through the HTTP (or HTTPS) protocols, and proxying services. And since they are used for dynamic extensibility, they may be used in a plug-in style, supporting facilities such as search engines and semi-custom applications.

Although all servlets are written in Java, their clients may be written in any language. When servlets are used in the middle tiers of distributed application systems, they can in turn be clients to other services, written in any language. For example, servlets can use JDBC to contact an application's highly tuned relational databases. Communicating with other kinds of current or legacy systems may call for alternate software packages, as required by those systems.

Many Ways to Use Servlets

A few of the many applications for servlets include:

- A simple servlet can process data which was POSTed over HTTPS using an HTML FORM, passing data such as a purchase order (with credit card data). This would be part of an order entry and processing system, working with product and inventory databases and perhaps an on-line payment system.
- Since servlets handle multiple requests concurrently, the requests can be synchronized with each other to support collaborative applications such as on-line conferencing.
- One could define a community of active agents, which share work among each other. The code for each agent would be loaded as a servlet, and the agents would pass data to each other.
- One servlet could forward requests other servers. This technique can balance load among several servers which mirror the same content. Or, it could be used to partition a single logical service between several servers, routing requests according to task type or organizational boundaries.

API Availability

The Java Servlet API is a *Standard Java Extension API*. This means that while it is not part of the core Java framework which must always be part of all products bearing the Java brand, it will be made available with such products by their vendors as an add-on package.

JavaSoft has provided a package which may be used to embed servlet support in other web servers, including Apache (and derived servers such as Stronghold), Netscape servers, and Microsoft's IIS.

Servlets were initially supported in the Java Web Server from JavaSoft. Since then, a number of other Java-based web servers have supported the API, or committed to supporting it.

Features of the `java.servlet` API

The servlet API is already supported by most Java based web servers, and implementations are available for other popular web servers. This means that you get the *Java Advantage* when you use the servlet API: not only will your code not have memory leaks and suffer from hard-to-find pointer bugs, but your code runs on platforms from many server vendors.

You don't need to lock yourself into a single-vendor server extension API, using error-prone languages such as C or scripting languages. Instead, you can do all of your server (and client) development in a single development language. No longer does your team need to juggle multiple languages, and cope with the maintainability problems inherent in scripting languages. No longer do your web-based applications need to suffer the performance penalties of CGI. Java is the professional programming tool you need to provide robust server solutions.

This section provides you an overview of the key features of Java servlets, addressing these main topics:

- What do Servlets Look Like?
- Servlet Lifecycle
- Security Features
- HTML-Aware Servlets
- HTTP-Specific Servlets
- Performance Features

What Do Servlets Look Like?

Servlets support the familiar programming model of accepting requests and generating responses. This model is used with a variety of distributed system programming toolsets, ranging from remote procedure calls to the HTTP requests made to web servers.

Servlets implement the *Servlet* interface, usually by extending either the generic or an HTTP-specific implementation. The simplest possible servlet defines a single method, *service*:

```
import java.servlet.*;

public class MyServlet extends GenericServlet {
    public void service (
        ServletRequest      request,
        ServletResponse     response
    ) throws ServletException, IOException
    {
        ...
    }
}
```

The service method is provided with *Request* and *Response* parameters. These encapsulate the data sent by the client, providing access to parameters and allowing servlets to report status including errors. Servlets normally retrieve most of their parameters through an input stream, and send their responses using an output stream:

```
ServletInputStream  in = request.getInputStream ();
ServletOutputStream out = response.getOutputStream ();
```

These input and output streams may be used with data in whatever format is appropriate. For example, an applet and servlet might exchange data using object serialization; HTML, and numerous image formats, may also be appropriate data formats.

Servlet and Environment State

Since servlets are Java objects, they have instance-specific data. This means that in effect servlets are independent applications running within servers, without needing the complexity of additional classes (which are required by some alternative server extension APIs).

Servlets have access to some servlet-specific configuration data at initialization time. This allows different instances of the same servlet class to be initialized with different data, and be managed as differently named servlets. The data provided at initialization time would include an area where each instance would keep its persistent instance-specific state.

Servlets also have the ability to interact with their environment through a `ServletContext` object.

Usage Modes

Servlets may be used in several modes. However, not all server environments support all of these modes:

- The basic mode is at the core of a request/response protocol.
- Servers can chain servlets together to in *filter chains*.
- Servlets may be specialized to support protocols such as HTTP.
- In HTTP based applications, servlets are a complete (and more efficient) replacement for CGI based extensions.
- Also in HTTP based applications, servlets may be used with HTML *server side includes* to dynamically generate part of a web document.

Servlet Lifecycle

Servlets are always dynamically loaded, although servers will usually provide an administrative option to force loading and initializing particular servlets when the server starts up. Servlets are loaded using normal Java class loading facilities, which means that they may be loaded from remote directories (for example, a trusted *https://department/servlets* directory) as easily as from the local filesystem. This allows for increased flexibility in system architecture and easier distribution of services in a network.

Servers also vary in how they know *when* to load servlets. When a request comes in, the server knows how to map it to a servlet, which may first need to be loaded. That mapping will usually be done in one of these common ways:

- Server administrators might specify that some kinds of client requests always map to a particular servlet. For example, one which talks to a particular database.
- Server administrators might specify that part of the client request is the name of the servlet, as found in an administered *servlets* directory. At many sites, that directory would be shared between servers which share the load of processing for the site's clients.
- Some servers may be able to automatically invoke servlets to filter the output of other servlets, based on their administrative configuration. For example, particular types of servlet output may trigger postprocessing by other servlets, perhaps to perform format conversions.
- Properly authorized clients might specify the servlet which is to be invoked, without administrative intervention.

After being loaded, three main methods are involved in the lifecycle of a servlet:

- Servlets are activated by the server through an *init* call. Servlet writers may, if they wish, provide their own implementation of this call, to perform potentially costly (usually, I/O intensive) setup only once, rather than once per request. Examples of such setup include initializing sessions with other network services or getting access to their persistent data (stored in a database or file).
- After initialization, servlets handle many requests. Each client request generates one *service*

upcall. These requests may be concurrent; this allows servlets to coordinate activities among many clients. Class-static state may be used to share data between requests.
- Requests are processed until the servlet is explicitly shut down by the web server, by calling the *destroy* method. The servlet's class may then become eligible for garbage collection.

Security Features

Servlets have access to information about their clients. When used with secure protocols such as SSL, peer identities can be determined quite reliably. Servlets relying on HTTP also have access to HTTP-specific authentication data.

Servlets have the Java advantage: memory access violations and strong typing violations are not possible, so that faulty servlets will not crash servers the way that is common in most C language server extension environments.

Unlike any other current server extension API, Java Servlets provide strong security policy support. This is because all Java environments provide a **Security Manager** which can be used to control whether actions such as network or file access are to be permitted. By default, all servlets are *untrusted*, and are not allowed to perform operations such as accessing network services or local files.

However, servlets "built in to" the server, or servlets which have been digitally signed as they were put into Java Archive (JAR) files, may be *trusted* and granted more permissions by the security manager. A digital signature on executable code indicates that the organization which signed the code "vouches for it" in some sense. Such signatures can't support accountability by themselves, but they do indicate a degree of assurance that may be placed on use of that code. For example, a particular signature from a MIS organization might be required on all code which is granted general access to network services within a corporate intranet. That signature might only be used on code which is strongly believed not to violate particular security policies.

Extension APIs in other languages, such as C or scripting languages, can't support such fine grained access controls even if they do allow digital signatures for their code. This means that 100% Pure Java extensions are fundamentally more secure than current competitive solutions, including in particular Active-X. (The security of code signing used with Active-X has been seriously overstated.)

There are immediate commercial applications for such improved security technologies. Today, most Internet Service Providers (ISPs) are not able to accept server extensions from their clients. This is because they have no way to defend themselves, or their other clients, against attacks building on extensions which use native C code or CGI facilities. However, extensions built with 100% Pure Java Servlets can be prevented from maliciously altering data. Combined with the use of digitally signed code, ISPs can have the assurance that they are not assuming undue risk when they extend their web servers with servlets provided by their customers.

HTML-Aware Servlets

Many servlets will directly generate HTML formatted text, since it is easy to do with standard internationalized Java formatted output classes such as `java.io.PrintWriter`. There is no need to use scripting languages to dynamically modify or generate HTML pages.

Other Java HTML generation approaches may of course be used. For example, some multi-language sites (which serve pages in many languages, such as English and Japanese) may prefer to maintain language-specific libraries of localized HTML template files, and fill out those templates using localized message catalogs. Other sites may have developed HTML generation packages which are particularly well attuned to other specific needs for dynamic web page generation; for example, ones closely integrated with other application toolsets.

In web servers which provide complete servlet support, servlets may also be invoked by servers to help preprocess web pages, using *Server Side Include* functionality. The desire for this kind of preprocessing is indicated to web servers by special HTML syntax such as this, which is used in *shtml* files:

```
<SERVLET NAME=ServletName>
<PARAM NAME=param1 VALUE=val1>
<PARAM NAME=param2 VALUE=val2>
    If you see this text, it means that the web server
    providing this page does not support the SERVLET tag.
    Ask your Internet Service Provider to upgrade!
</SERVLET>
```

This style usage of the SERVLET tag indicates that a pre-configured servlet should be loaded and initialized (if that has not already been done), then invoked with a particular set of parameters. The output of that servlet is included directly in the HTML-formatted response. (Another invocation style allows passing initialization arguments to the servlet, and specifying its CLASS and CODEBASE directly.)

The SERVLET tag could be used to insert formatted data such as the the output of a web or database search, user-targetted advertising, individualized views of an on-line magazine, and more.

Using these two facilities, HTML-aware servlets can generate arbitrary dynamic web pages. Typical servlets will accept input parameters from many sources, such as:

- The input stream of a request, perhaps from an applet;
- In the URI of the request;
- From some other servlet or network service;
- By using parameters passed from an HTML form.

Those parameters will be used to generate HTML-formatted responses. The servlet will often consult one or more databases, or other data with which the servlet has been configured, when deciding the exact data to return with the response.

HTTP-Specific Servlets

Servlets which are being used with the HTTP protocol may support any HTTP method, including GET, POST, HEAD, and more. They may redirect requests to other locations, and send HTTP-specific error messages. They can get access to parameters which were passed through standard HTML forms, including the HTTP method to be performed and the URI which identifies the destination of the request:

```
String      method = request.getMethod ();  // e.g. POST
String      uri = request.getRequestURI ();

String      name = request.getParameter ("name");
```

```
String      phone = request.getParameter ("phone");
String      card = request.getParameter ("creditcard");
```

For HTTP servlets, request and response data are always provided as MIME format data. The servlet specifies the data type, and then writes data encoded in that format. This allows servlets to consult arbitrary sources of input data, and then return data in the form appropriate to the particular request. Examples include HTML, graphics formats such as JPG or MPEG, and data formats used by specialized applications.

HTTP Servlets are superior to CGI programs in most applications because of better performance, flexibility, portability, and security. Rather than using CGI or a C language plug-in, write your next server extension using the Servlet API!

Performance Features

One of the biggest performance features of servlets is that they do not require creation of a new process for each request. In most environments, many servlets run in parallel within the same process as the server. When used in such environments with HTTP, servlets provide compelling performance advantages over both the CGI approach and the Fast-CGI approach.

This is because servlets only require light weight thread context switches. Even Fast-CGI involves heavy weight process context switching on each request, and regular CGI requires even heavier weight process startup and initialization code on each request. Since in most environments servlets can handle many client requests each time they are initialized, the costs of that initialization are spread over many methods. All the client requests to that service have the opportunity to share data and communications resources, benefiting more strongly from system caches.

With many implementations of the Java Virtual Machine (JVM), Java Servlet programs automatically take advantage of additional processors, helping you scale your applications up from entry-level servers all the way up to mainframe class multiprocessors. This helps provide better throughput and response time to your clients. Because 100% Pure Java programs don’t care what operating system they use, you have the power to choose whatever system vendor best addresses your requirements in any given application.

Using Java Servlets in Web-Based Applications

Servlets can be key components in many large applications leveraging Java and other Internet technologies.

Three Tier Applications

Many organizations find that they need to use multi-tier applications. Most client and single server (two tier) models give way to ones which includes many servers, where servers exchange data between each other.

The first tier of such an application could use any number of Java-enabled browsers, including ones running on Network Computers (NCs) as well as ones built on more traditional platforms such as personal computers or workstations. Complex user interface tasks would be handled by Java applets downloaded from the second tier servers; simpler tasks could be handled using standard HTML forms.

The second tier of such a system would consist of servlets which encapsulate the particular business rules and logic of the application at hand, perhaps arranging to deliver a new shipment of parts. Such rules could include application-specific access controls for sensitive (but unclassified) corporate data.

The third tier of such systems consists of data repositories. This tier might be accessed using relational database interfaces such as JDBC, or other interfaces supported by legacy data.

Web Publishing System

Many organizations have large collections of data to manage and publish. The form of that data tends to be flexible; for example, one might have collections of both historical and realtime weather data, and want to present that data in easily understood formats in response to current applications for that data.

Such a system might use several relations in a database accessed through JDBC, holding historical data or data which had not yet been processed. Java-equipped remote recording stations would post data (such as temperature, wind, and rainfall; a frame of image data; or a stream of MPEG data), to a collecting site. At that collecting site, the servlet processing the data might store some data for the last two weeks in full, or discard some of it immediately.

Those collecting sites would get queries from other sites (perhaps individual browsers, or another collecting site) to return that data in arbitrary formats. Servlets would process saved data and return reports in the appropriate format ... ranging from web pages holding the current data, to historical tables and graphs, up to automated forecasts. Still other servlets could perform administrative tasks, such as archiving and deleting data; or pulling data from staging areas as part of an automated data distribution system.

This same publishing model could be used for other kinds of on-line application data, such as catalogs or databases holding the on-line documentation for a large engineering project.

Top
java-server-feedback@java.sun.com