Oracle Docs :

servlet is a Java programming language class used to extend the capabilities of servers that host applications accessed by means of a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers. For such applications, Java Servlet technology defines HTTP-specific servlet classes.

The javax.servlet and javax.servlet.http packages provide interfaces and classes for writing servlets. All servlets must implement the Servletinterface, which defines lifecycle methods. When implementing a generic service, you can use or extend the GenericServlet class provided with the Java Servlet API. The HttpServlet class provides methods, such as doGet and doPost, for handling HTTP-specific services.

**Table 15-1 Servlet Lifecycle Events**

|  |  |  |
| --- | --- | --- |
| **Object** | **Event** | **Listener Interface and Event Class** |
| Web context | Initialization and destruction | javax.servlet.ServletContextListener and ServletContextEvent |
| Web context | Attribute added, removed, or replaced | javax.servlet.ServletContextAttributeListener andServletContextAttributeEvent |
| Session | Creation, invalidation, activation, passivation, and timeout | javax.servlet.http.HttpSessionListener, javax.servlet.http.HttpSessionActivationListener, and HttpSessionEvent |
| Session | Attribute added, removed, or replaced | javax.servlet.http.HttpSessionAttributeListener and HttpSessionBindingEvent |
| Request | A servlet request has started being processed by web components | javax.servlet.ServletRequestListener and ServletRequestEvent |
| Request | Attribute added, removed, or replaced | javax.servlet.ServletRequestAttributeListener and ServletRequestAttributeEvent |

### Handling Servlet Lifecycle Events

You can monitor and react to events in a servlet’s lifecycle by defining listener objects whose methods get invoked when lifecycle events occur. To use these listener objects, you must define and specify the listener class.

Use the @WebListener annotation to define a listener to get events for various operations on the particular web application context. Classes annotated with @WebListener must implement one of the following interfaces:

javax.servlet.ServletContextListener

javax.servlet.ServletContextAttributeListener

javax.servlet.ServletRequestListener

javax.servlet.ServletRequestAttributeListener

javax.servlet..http.HttpSessionListener

javax.servlet..http.HttpSessionAttributeListener

The lifecycle of a servlet is controlled by the container in which the servlet has been deployed. When a request is mapped to a servlet, the container performs the following steps.

1. If an instance of the servlet does not exist, the web container
   1. Loads the servlet class.
   2. Creates an instance of the servlet class.
   3. Initializes the servlet instance by calling the init method. Initialization is covered in [Creating and Initializing a Servlet](https://docs.oracle.com/javaee/6/tutorial/doc/bnafu.html).
2. Invokes the service method, passing request and response objects. Service methods are discussed in [Writing Service Methods](https://docs.oracle.com/javaee/6/tutorial/doc/bnafv.html).

If it needs to remove the servlet, the container finalizes the servlet by calling the servlet’s destroy method. For more information, see [Finalizing a Servle](https://docs.oracle.com/javaee/6/tutorial/doc/bnags.html)

### Handling Servlet Errors

Any number of exceptions can occur when a servlet executes. When an exception occurs, the web container generates a default page containing the following message:

A Servlet Exception Has Occurred

But you can also specify that the container should return a specific error page for a given exception.

### Using Scope Objects

Collaborating web components share information by means of objects that are maintained as attributes of four scope objects. You access these attributes by using the getAttribute and setAttribute methods of the class representing the scope. [Table 15-2](https://docs.oracle.com/javaee/6/tutorial/doc/bnafo.html#bnafq) lists the scope objects.

**Table 15-2 Scope Objects**

|  |  |  |
| --- | --- | --- |
| **Scope Object** | **Class** | **Accessible from** |
| Web context | javax.servlet.ServletContext | Web components within a web context. See [Accessing the Web Context](https://docs.oracle.com/javaee/6/tutorial/doc/bnagl.html). |
| Session | javax.servlet.http.HttpSession | Web components handling a request that belongs to the session. See [Maintaining Client State](https://docs.oracle.com/javaee/6/tutorial/doc/bnagm.html). |
| Request | Subtype of javax.servlet.ServletRequest | Web components handling the request. |
| Page | javax.servlet.jsp.JspContext | The JSP page that creates the object. |

## Writing Service Methods

The service provided by a servlet is implemented in the service method of a GenericServlet, in the do*Method* methods (where *Method* can take the value Get, Delete, Options, Post, Put, or Trace) of an HttpServlet object, or in any other protocol-specific methods defined by a class that implements the Servlet interface. The term **service method** is used for any method in a servlet class that provides a service to a client.

The general pattern for a service method is to extract information from the request, access external resources, and then populate the response, based on that information. For HTTP servlets, the correct procedure for populating the response is to do the following:

1. Retrieve an output stream from the response.
2. Fill in the response headers.
3. Write any body content to the output stream.

Response headers must always be set before the response has been committed. The web container will ignore any attempt to set or add headers after the response has been committed. The next two sections describe how to get information from requests and generate responses.

### Getting Information from Requests

A request contains data passed between a client and the servlet. All requests implement the ServletRequest interface. This interface defines methods for accessing the following information:

* Parameters, which are typically used to convey information between clients and servlets
* Object-valued attributes, which are typically used to pass information between the web container and a servlet or between collaborating servlets
* Information about the protocol used to communicate the request and about the client and server involved in the request
* Information relevant to localization

You can also retrieve an input stream from the request and manually parse the data. To read character data, use the BufferedReader object returned by the request’s getReader method. To read binary data, use the ServletInputStream returned by getInputStream.

HTTP servlets are passed an HTTP request object, HttpServletRequest, which contains the request URL, HTTP headers, query string, and so on. An HTTP request URL contains the following parts:

http://[*host*]:[*port*][*request-path*]?[*query-string*]

The request path is further composed of the following elements:

* **Context path**: A concatenation of a forward slash (/) with the context root of the servlet’s web application.
* **Servlet path**: The path section that corresponds to the component alias that activated this request. This path starts with a forward slash (/).
* **Path info**: The part of the request path that is not part of the context path or the servlet path.

You can use the getContextPath, getServletPath, and getPathInfo methods of the HttpServletRequest interface to access this information. Except for URL encoding differences between the request URI and the path parts, the request URI is always comprised of the context path plus the servlet path plus the path info.

Query strings are composed of a set of parameters and values. Individual parameters are retrieved from a request by using the getParameter method. There are two ways to generate query strings.

* A query string can explicitly appear in a web page.
* A query string is appended to a URL when a form with a GET HTTP method is submitted.

### Constructing Responses

A response contains data passed between a server and the client. All responses implement the ServletResponse interface. This interface defines methods that allow you to

* Retrieve an output stream to use to send data to the client. To send character data, use the PrintWriter returned by the response’s getWritermethod. To send binary data in a Multipurpose Internet Mail Extensions (MIME) body response, use the ServletOutputStream returned by getOutputStream. To mix binary and text data, as in a multipart response, use a ServletOutputStream and manage the character sections manually.
* Indicate the content type (for example, text/html) being returned by the response with the setContentType(String) method. This method must be called before the response is committed. A registry of content type names is kept by the Internet Assigned Numbers Authority (IANA) at <http://www.iana.org/assignments/media-types/>.
* Indicate whether to buffer output with the setBufferSize(int) method. By default, any content written to the output stream is immediately sent to the client. Buffering allows content to be written before anything is sent back to the client, thus providing the servlet with more time to set appropriate status codes and headers or forward to another web resource. The method must be called before any content is written or before the response is committed.
* Set localization information, such as locale and character encoding. See [Chapter 17, Internationalizing and Localizing Web Applications](https://docs.oracle.com/javaee/6/tutorial/doc/bnaxu.html) for details.

HTTP response objects, javax.servlet.http.HttpServletResponse, have fields representing HTTP headers, such as the following:

* Status codes, which are used to indicate the reason a request is not satisfied or that a request has been redirected.
* Cookies, which are used to store application-specific information at the client. Sometimes, cookies are used to maintain an identifier for tracking a user’s session (see [Session Tracking](https://docs.oracle.com/javaee/6/tutorial/doc/bnagm.html#bnagr)).

Java Servlets are programs that run on a Web or Application server and act as a middle layer between a requests coming from a Web browser or other HTTP client and databases or applications on the HTTP server.

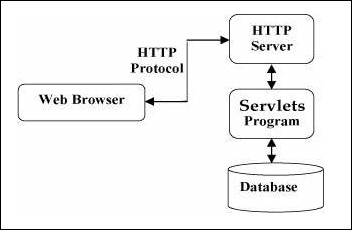
Using Servlets, you can collect input from users through web page forms, present records from a database or another source, and create web pages dynamically.

Java Servlets often serve the same purpose as programs implemented using the Common Gateway Interface (CGI). But Servlets offer several advantages in comparison with the CGI.

* Performance is significantly better.
* Servlets execute within the address space of a Web server. It is not necessary to create a separate process to handle each client request.
* Servlets are platform-independent because they are written in Java.
* Java security manager on the server enforces a set of restrictions to protect the resources on a server machine. So servlets are trusted.
* The full functionality of the Java class libraries is available to a servlet. It can communicate with applets, databases, or other software via the sockets and RMI mechanisms that you have seen already.

Servlets Architecture

The following diagram shows the position of Servlets in a Web Application.



Servlets Tasks

Servlets perform the following major tasks −

* Read the explicit data sent by the clients (browsers). This includes an HTML form on a Web page or it could also come from an applet or a custom HTTP client program.
* Read the implicit HTTP request data sent by the clients (browsers). This includes cookies, media types and compression schemes the browser understands, and so forth.
* Process the data and generate the results. This process may require talking to a database, executing an RMI or CORBA call, invoking a Web service, or computing the response directly.
* Send the explicit data (i.e., the document) to the clients (browsers). This document can be sent in a variety of formats, including text (HTML or XML), binary (GIF images), Excel, etc.
* Send the implicit HTTP response to the clients (browsers). This includes telling the browsers or other clients what type of document is being returned (e.g., HTML), setting cookies and caching parameters, and other such tasks.

Servlets Packages

Java Servlets are Java classes run by a web server that has an interpreter that supports the Java Servlet specification.

Servlets can be created using the **javax.servlet** and **javax.servlet.http**packages, which are a standard part of the Java's enterprise edition, an expanded version of the Java class library that supports large-scale development projects.

These classes implement the Java Servlet and JSP specifications. At the time of writing this tutorial, the versions are Java Servlet 2.5 and JSP 2.1.

Java servlets have been created and compiled just like any other Java class. After you install the servlet packages and add them to your computer's Classpath, you can compile servlets with the JDK's Java compiler or any other current compiler.

A servlet life cycle can be defined as the entire process from its creation till the destruction. The following are the paths followed by a servlet.

* The servlet is initialized by calling the **init()** method.
* The servlet calls **service()** method to process a client's request.
* The servlet is terminated by calling the **destroy()** method.
* Finally, servlet is garbage collected by the garbage collector of the JVM.

Now let us discuss the life cycle methods in detail.

The init() Method

The init method is called only once. It is called only when the servlet is created, and not called for any user requests afterwards. So, it is used for one-time initializations, just as with the init method of applets.

The servlet is normally created when a user first invokes a URL corresponding to the servlet, but you can also specify that the servlet be loaded when the server is first started.

When a user invokes a servlet, a single instance of each servlet gets created, with each user request resulting in a new thread that is handed off to doGet or doPost as appropriate. The init() method simply creates or loads some data that will be used throughout the life of the servlet.

The init method definition looks like this −

public void init() throws ServletException {

// Initialization code...

}

The service() Method

The service() method is the main method to perform the actual task. The servlet container (i.e. web server) calls the service() method to handle requests coming from the client( browsers) and to write the formatted response back to the client.

Each time the server receives a request for a servlet, the server spawns a new thread and calls service. The service() method checks the HTTP request type (GET, POST, PUT, DELETE, etc.) and calls doGet, doPost, doPut, doDelete, etc. methods as appropriate.

Here is the signature of this method −

public void service(ServletRequest request, ServletResponse response)

throws ServletException, IOException {

}

The service () method is called by the container and service method invokes doGe, doPost, doPut, doDelete, etc. methods as appropriate. So you have nothing to do with service() method but you override either doGet() or doPost() depending on what type of request you receive from the client.

The doGet() and doPost() are most frequently used methods with in each service request. Here is the signature of these two methods.

The doGet() Method

A GET request results from a normal request for a URL or from an HTML form that has no METHOD specified and it should be handled by doGet() method.

public void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

// Servlet code

}

The doPost() Method

A POST request results from an HTML form that specifically lists POST as the METHOD and it should be handled by doPost() method.

public void doPost(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

// Servlet code

}

The destroy() Method

The destroy() method is called only once at the end of the life cycle of a servlet. This method gives your servlet a chance to close database connections, halt background threads, write cookie lists or hit counts to disk, and perform other such cleanup activities.

After the destroy() method is called, the servlet object is marked for garbage collection. The destroy method definition looks like this −

public void destroy() {

// Finalization code...

}

Architecture Diagram

The following figure depicts a typical servlet life-cycle scenario.

* First the HTTP requests coming to the server are delegated to the servlet container.
* The servlet container loads the servlet before invoking the service() method.
* Then the servlet container handles multiple requests by spawning multiple threads, each thread executing the service() method of a single instance of the servlet.

