



## CS-25 ADVANCE JAVA PROGRAMMING

HANDOUT MATERIAL

BCA 5<sup>TH</sup> SEM.

2018-19

# Servlet Programming

Unit – II

Part – II

Prepared By: Dr. Sharon V. Mohtra  
Assistant Professor,  
Dept. of Computer Applications,  
Christ College,  
Rajkot



## Content

- Servlet Introduction
- Architecture of a Servlet
- Servlet API (javax.servlet and javax.servlet.http)
- Servlet Life Cycle
- Developing and Deploying Servlets
- Handling Servlet Requests and Responses
- Reading Initialization Parameters
- Session Tracking Approaches (URL Rewriting, Hidden Form Fields, Cookies, Session API)
- Servlet Collaboration
- Servlet with JDBC

## I - Introduction to Servlets

- Servlets Implementation
  - doTrace( )
- The Servlet interface
- The Generic Servlet class
- The single thread Model interface
- The Http Servlet class
  - Service( )
  - doGet( )
  - doPost( )
  - doDelete( )
  - doOption( )
  - doPut( )
- Servlet Exceptions
  - The Servlet Exception class
  - The unavailable Exception class
- Servlet Lifecycle

## II - Servlet Request and Response

- The Http Servlet Request interface
  - getAttribute( )
  - setAttribute( )
  - getAttributeNames( )
  - getParameters( )
  - getParameterNames( )
  - getParameterValues( )
  - getRemoteHost( )
  - getRemoteAddr( )
  - getCookies( )
  - getHeaders( )
- The Http servlet Response Interface
  - getQueryString( )
  - getSession( )
  - getWriter( )
  - getcontentType( )
  - addCookie( )
  - encodeURL( )
  - sendRedirect( )
  - setHeader( )
  - setStatus( )

### III - Session

- Session Tracking Approaches
  - URL Rewriting
  - Hidden Form Fields
- Cookies
- Session API
- Session Tracking with Servlet API
- The Http Session interface
  - `getAttribute( )`
  - `getAttributeNames( )`
  - `getCreationTime( )`
  - `getId( )`
  - `getLastAccessedTime( )`
  - `isNew( )`
  - `removeAttribute( )`
  - `setAttribute( )`
  - `setMaxInactiveInterval( )`
  - `invalidate( )`

### IV - Servlet Collaboration

- Request Dispatching with Request
- Dispatcher interface
- `Forward( )`
- `Include( )`
- Servlet Context
- The servlet Context interface
- `getContext( )`
- `getRequestDispatcher( )`
- `getServerInfo( )`
- `getInitParameter( )`
- `getInitParameterNames( )`
- `getAttribute( )`
- `setAttribute( )`
- `removeAttribute( )`



## Overview

- Introduction
- Static vs. Dynamic pages
- Servlets
- Java Server Pages (JSP)



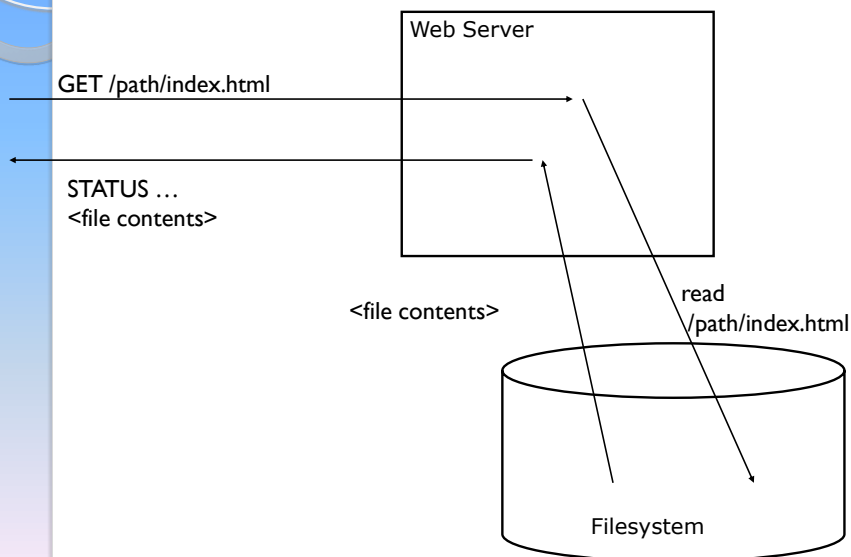
## Introduction

- What you know:
  - Java
  - JDBC
- What you'll know:
  - How to use Java *inside* a webserver!
- Why you might care to learn:
  - Building non-trivial web interfaces and applications
  - In particular: database-backed web applications

## Plain Old Static Pages

- Static webpages
  - Plain files stored in the filesystem
  - Webserver accepts pathnames
  - Returns contents of corresponding file
- Boring!
  - Can't generate customized content—always serve the same static pages...

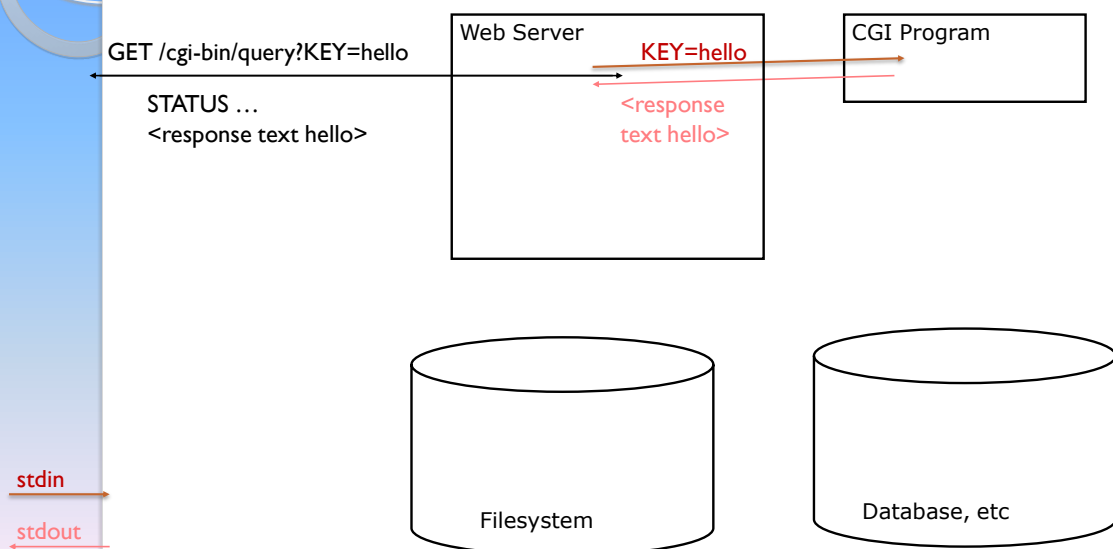
## Static Pages



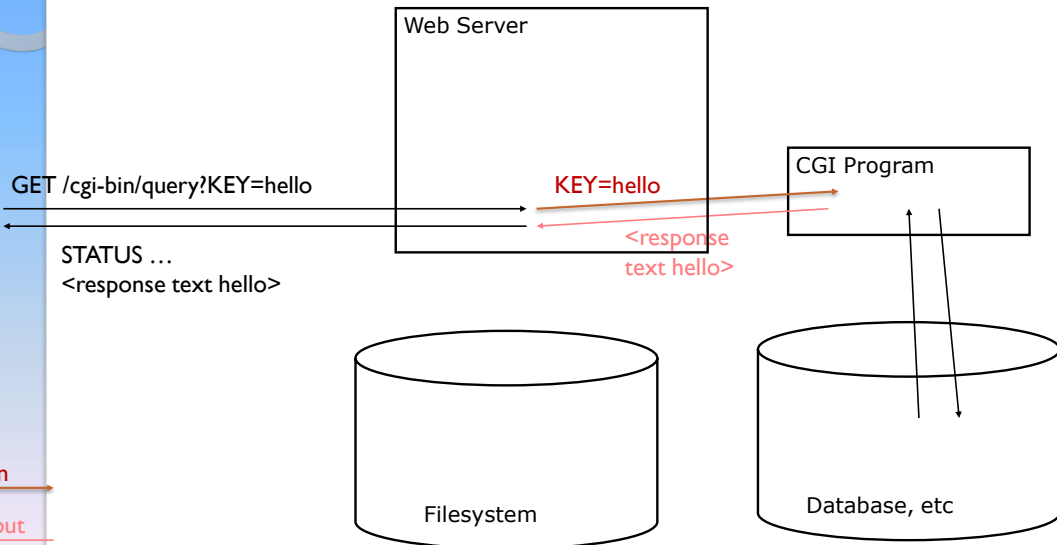
# Dynamic Content

- CGI: Easy “fix”
  - Common Gateway Interface
  - Oldest standard
  - But at least a **standard!**
  - Inefficient
  - No persistent state
- Forward requests to external programs
  - Spawn one process for each new request
  - Communicate via
    - Standard input/output
    - Environment variables
  - Process terminates after request is handled

## CGI



# CGI



## Servlets to the rescue...

- Little Java programs...
  - Contain application-specific code
  - Web server does generic part of request handling
  - Servlets run “in” the web server and do some of the handling
- Highlights
  - Standard!
  - Efficiency (much better than CGI)
  - Security (Java!)
  - Persistence (handle multiple requests)



## Objectives

- Java Servlet Web Components
- Support Environments for Java Servlets
- Servlets with CGI
- Compared to Java Applets
- Functionality of Java Servlets
- Java Servlet API
- Java Servlet Debugging

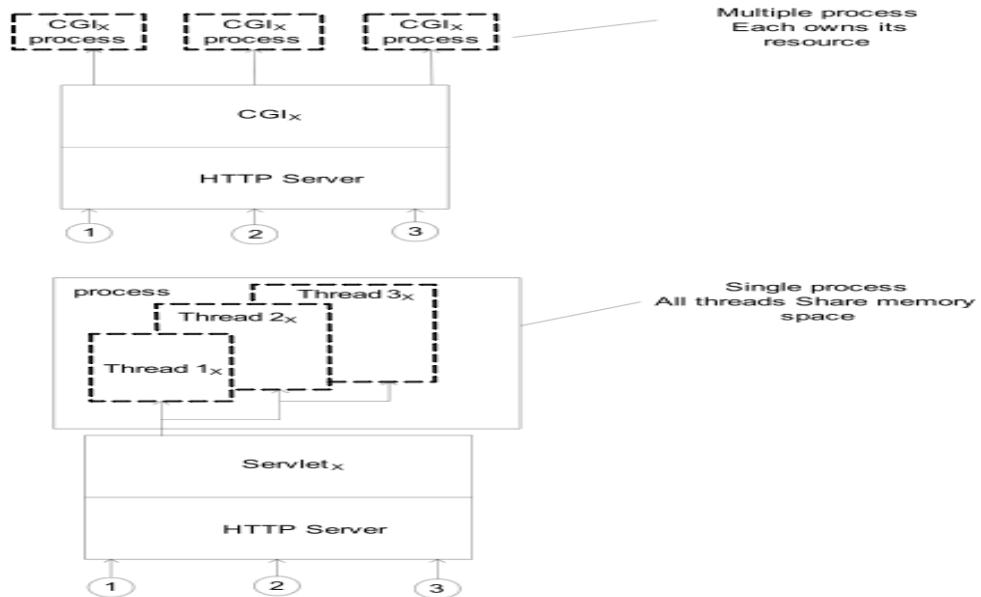


## Introduction to Java Servlets - II

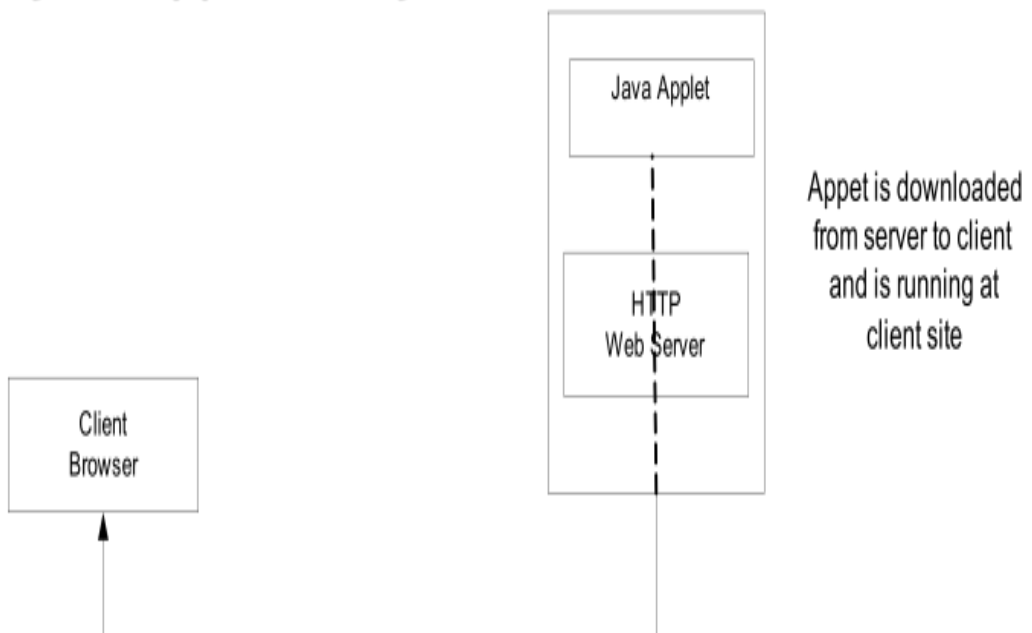
- Java Servlets technology provides an HTTP-based request and response paradigm on web servers.
- Java Servlets can handle generic service requests and respond to the client's requests.
- Java Servlets are used in embedded systems, wireless communication, and any other generic request/response application.
- The Common Gateway Interface (CGI) was a popular technology used to generate dynamic HTTP web contents in the mid-1990s.



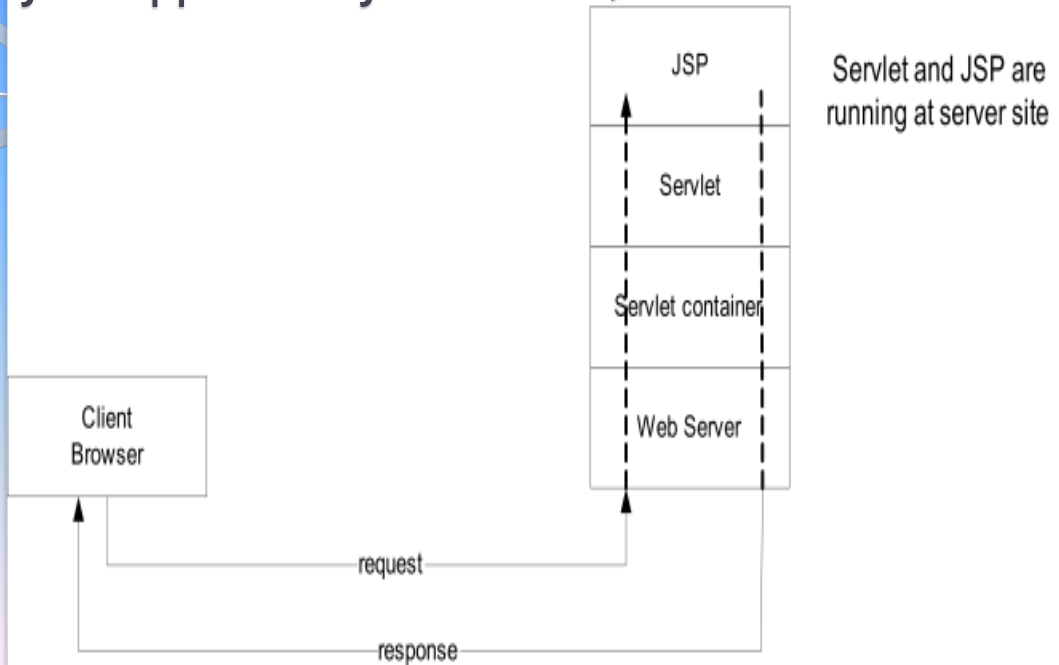
## CGI and Java Servlets Technology



## Java Applet and Java Servlet



## Java Applet and Java Servlet, contd.



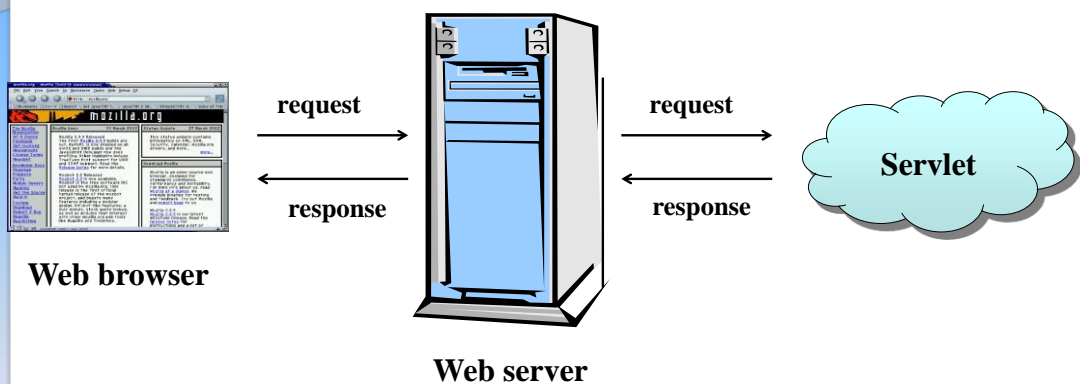
## Introduction to Java Servlets, contd.

- A Servlet component can delegate the requests to its back-end tier such as a database management system, RMI, EAI, or other Enterprise Information System (EIS).
- A Servlet is deployed as a middle tier just like other web components such as JSP components.
- The Servlet components are building block components, which always work together with other components such as JSP components, JavaBean components, Enterprise Java Bean (EJB) components, and web service components.
- A Servlet component is also a distributed component, which can provide services to remote clients and also access remote resources.

## What is a Servlet?

- Servlets are **Java programs** that can be run dynamically from a Web Server
- Servlets are a **server-side** technology
- A Servlet is an intermediating layer between an HTTP request of a client and the Web server

## A Java Servlet View





## An Example

- <http://www.abc.edu:8080/servlet/ServletTest>
- (accessible only from campus)



## What do Servlets do?

- **Read data sent by the user** (e.g., form data)
- **Look up other information about the request in the HTTP request** (e.g. authentication data, cookies, etc.)
- **Generate the result** (may do this by talking to a database, file system, etc.)
- **Format the result in a document** (e.g., make it into HTML)
- **Set the appropriate HTTP response parameters** (e.g. cookies, content-type, etc.)
- **Send the document to the user**



## Java Servlets Advantages

- **Efficiency:** Reduction of the time need for creating new processes and initialization and reduction of memory requirements as well.
- **Convenience:** All needed functionality is provided by the Servlets API.
- **Portability:** Cross-platform, Write Once Run Anywhere (WORA) code.
- **Security:** Built-in security layers.
- **Open source:** Free Servlet development kits available for download.
- **Functionality:** Session tracking, data sharing, JDBC database connections, etc.



## Java Servlets Disadvantages

- Designing in servlet is difficult and slows down the application.
- Writing complex business logic makes the application difficult to understand.
- You need a Java Runtime Environment on the server to run servlets. CGI is a completely language independent protocol, so you can write CGIs in whatever languages you have available (including Java if you want to).



## Finally the advantages over CGI

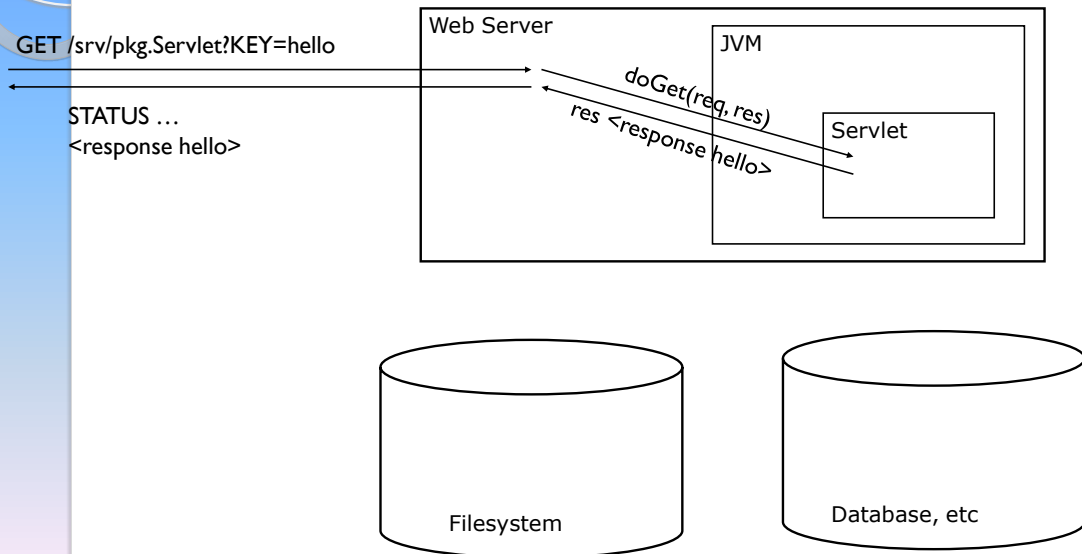
- **Fast performance and ease of use** combined with more power over traditional CGI (Common Gateway Interface).
- Traditional CGI scripts written in Java have a number of disadvantages when it comes to performance:
  - When an **HTTP request** is made, a new process is created for each call of the CGI script. This **overhead of process** creation can be very system-intensive, especially when the script does relatively fast operations. Thus, process creation will take more time than CGI script execution. Java Servlets solve this, as a Servlet is not a separate process. Each request to be handled by a Servlet is handled by a separate Java thread within the web server process.



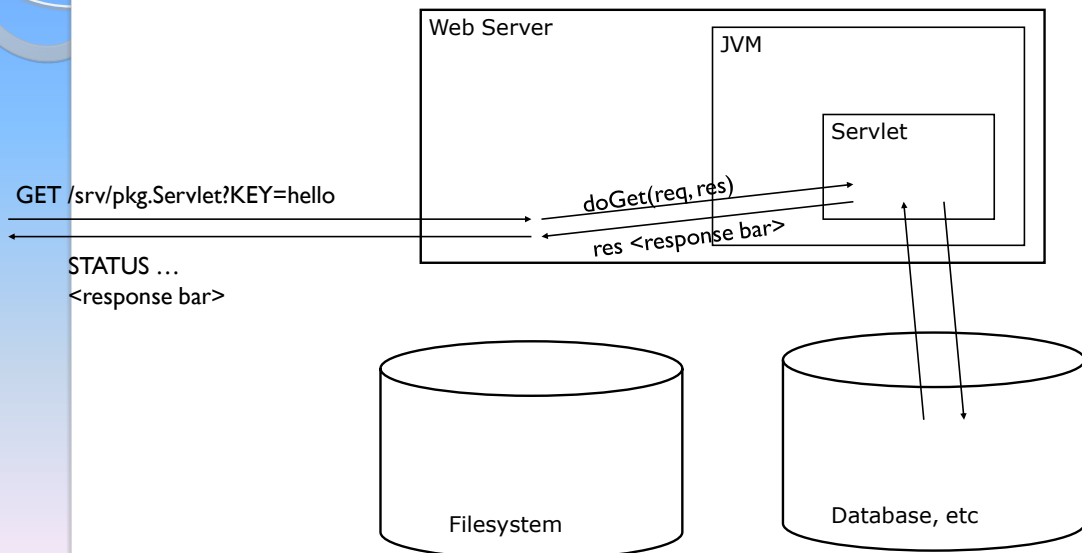
## Finally the advantages over CGI

- Simultaneous CGI request causes the CGI script to be **copied and loaded into memory** as many times as there are requests. However, with Servlets, there are the same amount of threads as requests, but there will only be one copy of the Servlet class created in memory that stays there also between requests.
- Only a single instance answers all requests concurrently. **This reduces memory usage and makes the management of persistent (long time storage) data easy.**
- A Servlet can be run by a Servlet container in a restrictive environment, called a sandbox. Like an applet that runs in the sandbox of the web browser, making a restrictive use of potentially harmful Servlets possible.

## Servlets...grilled!



## Servlets



## Java Servlet Architecture

- A Java Servlet is a typical Java class that extends the abstract class `HttpServlet`.
- The `HttpServlet` class extends another abstract class called `GenericServlet`.
- The `GenericServlet` class implements three interfaces: `javax.servlet.Servlet`, `javax.servlet.ServletConfig`, and `java.io.Serializable`.

## Java Servlet Class Hierarchy





## Class Hierarchy

- class java.lang.Object
  - class java.util.EventObject (implements java.io.Serializable)
    - class javax.servlet.**ServletContextEvent**
      - class javax.servlet.**ServletContextAttributeEvent**
  - class javax.servlet.**GenericServlet** (implements java.io.Serializable, javax.servlet.Servlet, javax.servlet.ServletConfig)
  - class java.io.InputStream
    - class javax.servlet.**ServletInputStream**
  - class java.io.OutputStream
    - class javax.servlet.**ServletOutputStream**
  - class javax.servlet.**ServletRequestWrapper** (implements javax.servlet.ServletRequest)
  - class javax.servlet.**ServletResponseWrapper** (implements javax.servlet.ServletResponse)
  - class java.lang.Throwable (implements java.io.Serializable)
    - class javax.servlet.**ServletException**
      - class javax.servlet.**UnavailableException**

## Interface Hierarchy

- interface java.util.EventListener
  - interface javax.servlet.**ServletContextAttributeListener**
  - interface javax.servlet.**ServletContextListener**
- interface javax.servlet.**Filter**
- interface javax.servlet.**FilterChain**
- interface javax.servlet.**FilterConfig**
- interface javax.servlet.**RequestDispatcher**
- interface javax.servlet.**Servlet**
- interface javax.servlet.**ServletConfig**
- interface javax.servlet.**ServletContext**
- interface javax.servlet.**ServletRequest**
- interface javax.servlet.**ServletResponse**
- interface javax.servlet.**SingleThreadModel**



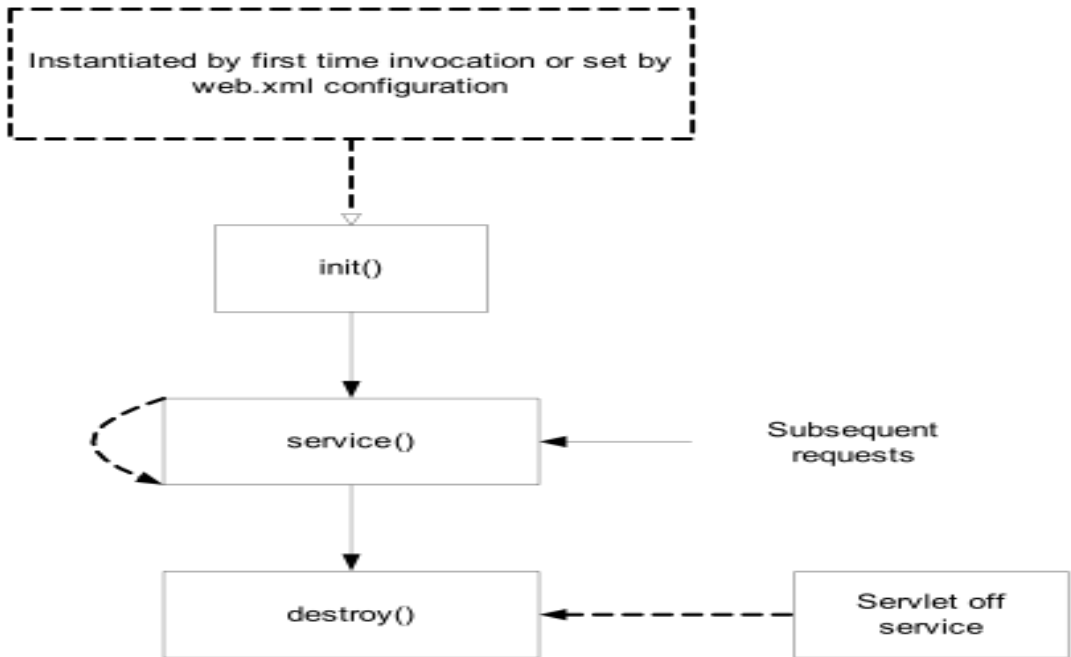
## INTRODUCTION TO SERVLET LIFE CYCLE



### Servlet Lifecycle

- A Servlet has a lifecycle just like a Java applet. The lifecycle is managed by the Servlet container.
- There are three methods in the Servlet interface, which each Servlet class must implement. They are `init()`, `service()`, and `destroy()`.

## The Lifecycle of a Servlet



## What is a **Servlet** Container?

**Servlet** container (also known as servlet engine) is a runtime environment, which implements **servlet** API and manages **life cycle** of **servlet** components.

Container is responsible for instantiating, invoking, and destroying **servlet** components.

One example of container is Apache Tomcat which is an opensource container.



## Servlet Life Cycle

- The life cycle 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.
- If an instance of the servlet does not exist, the Web container
  - Loads the servlet class.
  - Creates an instance of the servlet class.
  - Initializes the servlet instance by calling the **init** method. Initialization is covered in Initializing a Servlet.
- Invokes the service method, passing a request and response object.
- If the container needs to remove the servlet, it finalizes the servlet by calling the servlet's destroy method.



## Methods

- As Servlets are managed components managed by web container, one of the various responsibilities of web container is servlet life cycle management.
- A servlet is managed through a well defined life cycle that defines how it is loaded, instantiated and initialized, handles requests from clients and how it is taken out of service.
- The servlet life cycle methods are defined in the **javax.servlet.Servlet** interface of the Servlet API that all Servlets must implement directly or indirectly by extending `GenericServlet` or `HttpServlet` abstract classes.
- Most of the servlet you develop will implement it by extending `HttpServlet` class.

## Init and Destroy:

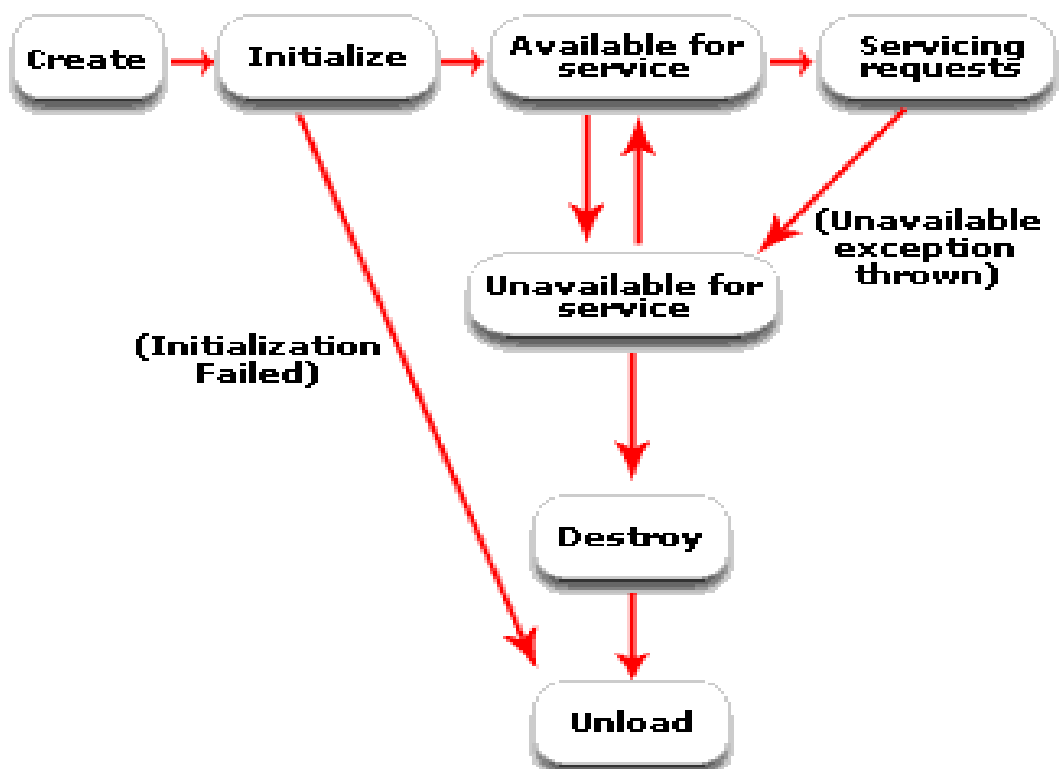
- Just like applets, servlets can define `init()` and `destroy()` methods. A servlet's `init(ServletConfig)` method is called by the server immediately after the server constructs the servlet's instance. Depending on the server and its configuration, this can be at any of these times:
- When the server starts
- When the servlet is first requested, just before the `service()` method is invoked
- At the request of the server administrator
- In any case, `init()` is guaranteed to be called before the servlet handles its first request.


## **The servlet life cycle methods defined in Servlet interface: `init()`, `service()` and `destroy()`**

- The life cycle starts when container instantiates the object of servlet class and calls the `init()` method, and ends with the container calling the `destroy()` method.
- The syntaxes of this methods are shown below.
- 1) `public void init(ServletConfig config) throws ServletException`
- or
- `public void init()`
- 2) `public void service(HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException`
- 
- 3) `public void destroy()`

## Life Cycle Steps


- The servlet life cycle consists of four steps:
  - instantiation,
  - initialization,
  - request handling and
  - end of service.





The life cycle of a servlet can be categorized into four parts:

- **Loading and Instantiation:** The servlet container loads the servlet during startup or when the first request is made. The loading of the servlet depends on the attribute `<load-on-startup>` of web.xml file. If the attribute `<load-on-startup>` has a positive value then the servlet is load with loading of the container otherwise it loads when the first request comes for service. After loading of the servlet, the container creates the instances of the servlet.
- **Initialization:** After creating the instances, the servlet container calls the `init()` method and passes the servlet initialization parameters to the `init()` method. The `init()` must be called by the servlet container before the servlet can service any request. The initialization parameters persist untill the servlet is destroyed. The `init()` method is called only once throughout the life cycle of the servlet. The servlet will be available for service if it is loaded successfully otherwise the servlet container unloads the servlet.

- 
- **Servicing the Request:** After successfully completing the initialization process, the servlet will be available for service. Servlet creates seperate threads for each request. The servlet container calls the `service()` method for servicing any request. The `service()` method determines the kind of request and calls the appropriate method (`doGet()` or `doPost()`) for handling the request and sends response to the client using the methods of the response object.
  - **Destroying the Servlet:** If the servlet is no longer needed for servicing any request, the servlet container calls the `destroy()` method . Like the `init()` method this method is also called only once throughout the life cycle of the servlet. Calling the `destroy()` method indicates to the servlet container not to sent the any request for service and the servlet releases all the resources associated with it. Java Virtual Machine claims for the memory associated with the resources for garbage collection.

## Do's and Don'ts

- **init**  
Executed once when the servlet is first loaded.  
*Not called for each request.*
- **Service**  
Called in a new thread by server for each request.  
Dispatches to doGet, doPost, etc.  
Do not override this method!
- **doGet, doPost, do.....**  
Handles GET, POST, etc. requests.  
Override these to provide desired behavior.
- **destroy**  
Called when server deletes servlet instance.  
*Not called after each request.*

## Why You Should Not Override service

- **The service method does other things besides just calling doGet**  
You can add support for other services later by adding doPut, doTrace, etc.  
You can add support for modification dates by adding a getLastModified method  
The service method gives you automatic support for:
  - HEAD requests
  - OPTIONS requests
  - TRACE requests
- **Alternative: have doPost call doGet**





## The Servlet Interface

All Known Implementing Classes: **GenericServlet**

- This interface is for developing servlets.
- A servlet is a body of Java code that is loaded into and runs inside a servlet engine, such as a web server.
- It receives and responds to requests from clients.
- For example, a client may need information from a database; a servlet can be written that receives the request, gets and processes the data as needed by the client, and then returns it to the client.



## Implementing interface

- All servlets implement this interface.
- Servlet writers typically do this by subclassing either `GenericServlet`, which implements the `Servlet` interface, or by subclassing `GenericServlet`'s descendent, `HttpServlet`.
- Developers need to directly implement this interface only if their servlets cannot (or choose not to) inherit from `GenericServlet` or `HttpServlet`. For example, RMI or CORBA objects that act as servlets will directly implement this interface.



## Methods

- The Servlet interface defines methods to initialize a servlet, to receive and respond to client requests, and to destroy a servlet and its resources. These are known as life-cycle methods, and are called by the network service in the following manner:
- Servlet is created then **initialized**.
- Zero or more **service** calls from clients are handled
- Servlet is **destroyed** then garbage collected and finalized



## Methods

- **destroy()** Cleans up whatever resources are being held (e.g., memory, file handles, threads) and makes sure that any persistent state is synchronized with the servlet's current in-memory state.
- **getServletConfig()** Returns a servlet config object, which contains any initialization parameters and startup configuration for this servlet.
- **getServletInfo()** Returns a string containing information about the servlet, such as its author, version, and copyright.
- **init(ServletConfig)** Initializes the servlet.
- **service(ServletRequest, ServletResponse)** Carries out a single request from the client.



## The Generic Servlet class

(public abstract class **GenericServlet**  
extends java.lang.Object implements  
Servlet, ServletConfig, java.io.Serializable)

- Defines a generic, protocol-independent servlet. To write an HTTP servlet for use on the Web, extend `HttpServlet` instead.
- It implements the `Servlet` and `Servlet-Config` interfaces.
- `GenericServlet` may be directly extended by a servlet, although it's more common to extend a protocol-specific subclass such as `HttpServlet`.




## GenericServlet.....

- It makes writing servlets easier.
- It provides simple versions of the lifecycle methods `init` and `destroy` and of the methods in the `Servlet-Config` interface.
- It also implements the `log` method, declared in the `ServletContext` interface.
- To write a generic servlet, you need only override the abstract service method.
- **Constructor Summary :**
  - **GenericServlet()** : Does nothing.



## Method Summary

- void **destroy()**  
Called by the servlet container to indicate to a servlet that the servlet is being taken out of service.
- java.lang.String **getInitParameter** (java.lang.String name)  
Returns a String containing the value of the named initialization parameter, or null if the parameter does not exist.
- java.util.Enumeration **getInitParameterNames()**  
Returns the names of the servlet's initialization parameters as an Enumeration of String objects, or an empty Enumeration if the servlet has no initialization parameters.

- 
- ServletConfig **getServletConfig()**  
Returns this servlet's ServletConfig object.
  - ServletContext **getServletContext()**  
Returns a reference to the Servlet-Context in which this servlet is running.
  - java.lang.String **getServletInfo()**  
Returns information about the servlet, such as author, version, and copyright.
  - java.lang.String **getServletName()**  
Returns the name of this servlet instance.
  - void **init()**  
A convenience method which can be overridden so that there's no need to call super.init(config).

- void **init**(ServletConfig config)  
Called by the servlet container to indicate to a servlet that the servlet is being placed into service.
- void **log**(java.lang.String msg)  
Writes the specified message to a servlet log file, prepended by the servlet's name.
- void **log**(java.lang.String message, java.lang.Throwable t)  
Writes an explanatory message and a stack trace for a given Throwable exception to the servlet log file, prepended by the servlet's name.
- abstract void **service**(ServletRequest req, ServletResponse res)  
Called by the servlet container to allow the servlet to respond to a request.

javax.servlet.http


**Class HttpServlet** (public abstract class **HttpServlet** extends **GenericServlet** implements **java.io.Serializable**)


- Provides an abstract class to be subclassed to create an HTTP servlet suitable for a Web site. A subclass of HttpServlet must override at least one method, usually one of these:
  - doGet, if the servlet supports HTTP GET requests
  - doPost, for HTTP POST requests
  - doPut, for HTTP PUT requests
  - doDelete, for HTTP DELETE requests
  - init and destroy, to manage resources that are held for the life of the servlet
  - getServletInfo, which the servlet uses to provide information about itself

- There's almost no reason to override the service method. service handles standard HTTP requests by dispatching them to the handler methods for each HTTP request type (the doXXX methods listed above).
- Likewise, there's almost no reason to override the doOptions and doTrace methods.
- Servlets typically run on multithreaded servers, so be aware that a servlet must handle concurrent requests and be careful to synchronize access to shared resources. Shared resources include in-memory data such as instance or class variables and external objects such as files, database connections, and network connections.
- **Constructor Summary**
  - **HttpServlet()** : Does nothing, because this is an abstract class.

## Method Summary

- protected void **doDelete**(HttpServletRequest req, HttpServletResponse resp)  
Called by the server (via the service method) to allow a servlet to handle a DELETE request.
- protected void **doGet**(HttpServletRequest req, HttpServletResponse resp)  
Called by the server (via the service method) to allow a servlet to handle a GET request.
- protected void **doHead**(HttpServletRequest req, HttpServletResponse resp)  
Receives an HTTP HEAD request from the protected service method and handles the request.

- 
- protected void **doOptions**(HttpServletRequest req, HttpServletResponse resp)  
Called by the server (via the service method) to allow a servlet to handle a OPTIONS request.
  - protected void **doPost**(HttpServletRequest req, HttpServletResponse resp)  
Called by the server (via the service method) to allow a servlet to handle a POST request.
  - protected void **doPut**(HttpServletRequest req, HttpServletResponse resp)  
Called by the server (via the service method) to allow a servlet to handle a PUT request.

- 
- protected void **doTrace**(HttpServletRequest req, HttpServletResponse resp)  
Called by the server (via the service method) to allow a servlet to handle a TRACE request.
  - protected long **getLastModified**(HttpServletRequest req)  
Returns the time the HttpServletRequest object was last modified, in milliseconds since midnight January 1, 1970 GMT.
  - protected void **service**(HttpServletRequest req, HttpServletResponse resp)  
Receives standard HTTP requests from the public service method and dispatches them to the doXXX methods defined in this class.
  - void **service**(ServletRequest req, ServletResponse res)  
Dispatches client requests to the protected service method.



## The SingleThreadModel interface

**Deprecated.** As of Java Servlet API 2.4, with no direct replacement.

- Ensures that servlets handle only one request at a time.
- This interface has **no methods**.
- If a servlet implements this interface, you are *guaranteed* that no two threads will execute concurrently (at the same time) in the servlet's service method.
- The servlet container can make this guarantee by synchronizing access to a single instance of the servlet, or by maintaining a pool of servlet instances and dispatching each new request to a free servlet.



## SingleThreadModel.....

- SingleThreadModel does not solve all thread safety issues.
- For example, session attributes and static variables can still be accessed by multiple requests on multiple threads at the same time, even when SingleThreadModel servlets are used.
- It is recommended that a developer take other means to resolve those issues instead of implementing this interface, such as avoiding the usage of an instance variable or synchronizing the block of the code accessing those resources.
- This interface is deprecated in Servlet API version 2.4.





## ° SERVLET EXCEPTIONS



**The Servlet Exception class** (public class **ServletException**  
extends **Exception**) : subclass - **UnavailableException**

- Defines a general exception a servlet can throw when it encounters difficulty.
- **Constructor Summary**
- **ServletException()**  
Constructs a new servlet exception.
- **ServletException(String message)**  
Constructs a new servlet exception with the specified message.
- **ServletException(String message, Throwable rootCause)**  
Constructs a new servlet exception when the servlet needs to throw an exception and include a message about the "root cause" exception that interfered with its normal operation, including a description message.
- **ServletException(Throwable rootCause)**  
Constructs a new servlet exception when the servlet needs to throw an exception and include a message about the "root cause" exception that interfered with its normal operation.
- **Method: Throwable getRootCause()**  
Returns the exception that caused this servlet exception.



## The `UnavailableException` class (public

class `UnavailableException` extends `ServletException`)

- Defines an exception that a servlet or filter throws to indicate that it is permanently or temporarily unavailable.
- When a servlet or filter is **permanently** unavailable, something is wrong with it, and it cannot handle requests until some action is taken.
- For example, a servlet might be configured incorrectly, or a filter's state may be corrupted. The component should log both the error and the corrective action that is needed.



## `UnavailableException`....

- A servlet or filter is **temporarily** unavailable if it cannot handle requests momentarily due to some system-wide problem. For example, a third-tier server might not be accessible, or there may be insufficient memory or disk storage to handle requests. A system administrator may need to take corrective action.
- Servlet containers can safely treat **both types** of unavailable exceptions in the **same way**. However, treating temporary unavailability effectively makes the servlet container more robust. Specifically, the servlet container might block requests to the servlet or filter for a period of time suggested by the exception, rather than rejecting them until the servlet container restarts.

## Constructor Summary

- **UnavailableException**(int seconds, Servlet servlet, String msg)  
**Deprecated.** As of Java Servlet API 2.2, use *UnavailableException(String, int)* instead.
- **UnavailableException**(Servlet servlet, String msg)  
**Deprecated.** As of Java Servlet API 2.2, use *UnavailableException(String)* instead.
- **UnavailableException**(String msg)  
Constructs a new exception with a descriptive message indicating that the servlet is permanently unavailable.
- **UnavailableException**(String msg, int seconds)  
Constructs a new exception with a descriptive message indicating that the servlet is temporarily unavailable and giving an estimate of how long it will be unavailable.

## Method Summary

- Servlet **getServlet()**  
**Deprecated.** As of Java Servlet API 2.2, with no replacement. Returns the servlet that is reporting its unavailability.
- int **getUnavailableSeconds()**  
Returns the number of seconds the servlet expects to be temporarily unavailable.
- boolean **isPermanent()**  
Returns a boolean indicating whether the servlet is permanently unavailable.



## ◦ SERVLET REQUEST & RESPONSE



### Servlet Request & Response

- The Java Servlet is a server-side web component that takes a HTTP request from a client, handles it, talks to a database, talks to a JavaBean component, and responds with a HTTP response or dispatches the request to other Servlets or JSP components.
- Servlets can dynamically produce text-based HTML markup contents and binary contents as well contents based on the client's request.
- **HttpServletRequest** and **HttpServletResponse** are the two interfaces that provide the Servlet with full access to all information from the request and the response sent back to the client.
- When the `doGet()` or `doPost()` is called, the Servlet container passes in the `HttpServletRequest` and `HttpServletResponse` objects.

javax.servlet.http

## Interface **HttpServletRequest**

- Extends the `ServletRequest` interface to provide request information for HTTP servlets.
- The servlet container creates an `HttpServletRequest` object and passes it as an argument to the servlet's service methods (`doGet`, `doPost`, etc).
- **Field Summary**
- static String **BASIC\_AUTH**  
String identifier for Basic authentication.
- static String **CLIENT\_CERT\_AUTH**  
String identifier for Client Certificate authentication.
- static String **DIGEST\_AUTH**  
String identifier for Digest authentication.
- static String **FORM\_AUTH**  
String identifier for Form authentication.

Summary

- String **getAuthType()**  
Returns the name of the authentication scheme used to protect the servlet.
- String **getContextPath()**  
Returns the portion of the request URI that indicates the context of the request.
- `Cookie[]` **getCookies()**  
Returns an array containing all of the `Cookie` objects the client sent with this request.
- long **getDateHeader(String name)**  
Returns the value of the specified request header as a long value that represents a `Date` object.
- String **getHeader(String name)**  
Returns the value of the specified request header as a `String`.
- Enumeration **getHeaderNames()**  
Returns an enumeration of all the header names this request contains.
- Enumeration **getHeaders(String name)**  
Returns all the values of the specified request header as an Enumeration of `String` objects.
- int **getIntHeader(String name)**  
Returns the value of the specified request header as an int.

- String **getMethod()**  
Returns the name of the HTTP method with which this request was made, for example, GET, POST, or PUT.
- String **getPathInfo()**  
Returns any extra path information associated with the URL the client sent when it made this request.
- String **getPathTranslated()**  
Returns any extra path information after the servlet name but before the query string, and translates it to a real path.
- String **getQueryString()**  
Returns the query string that is contained in the request URL after the path.
- String **getRemoteUser()**  
Returns the login of the user making this request, if the user has been authenticated, or null if the user has not been authenticated.
- String **getRequestedSessionId()**  
Returns the session ID specified by the client.
- String **getRequestURI()**  
Returns the part of this request's URL from the protocol name up to the query string in the first line of the HTTP request.
- StringBuffer **getRequestURL()**  
Reconstructs the URL the client used to make the request.

- String **getServletPath()**  
Returns the part of this request's URL that calls the servlet.
- HttpSession **getSession()**  
Returns the current session associated with this request, or if the request does not have a session, creates one.
- HttpSession **getSession(boolean create)**  
Returns the current HttpSession associated with this request or, if there is no current session and create is true, returns a new session.
- Principal **getUserPrincipal()**  
Returns a java.security.Principal object containing the name of the current authenticated user.
- boolean **isRequestedSessionIdFromCookie()**  
Checks whether the requested session ID came in as a cookie.
- boolean **isRequestedSessionIdFromURL()**  
Checks whether the requested session ID came in as part of the request URL.
- boolean **isRequestedSessionIdValid()**  
Checks whether the requested session ID is still valid.
- boolean **isUserInRole(String role)**  
Returns a boolean indicating whether the authenticated user is included in the specified logical "role".

javax.servlet.http

## Interface **HttpServletResponse**

(SubClasses: `HttpServletResponseWrapper`)

- Extends the `ServletResponse` interface to provide HTTP-specific functionality in sending a response. For example, it has methods to access HTTP headers and cookies.
- The servlet container creates an `HttpServletResponse` object and passes it as an argument to the servlet's service methods (`doGet`, `doPost`, etc).

## Method Summary

- void **addCookie**(`Cookie cookie`)  
Adds the specified cookie to the response.
- void **addDateHeader**(`String name`, `long date`)  
Adds a response header with the given name and date-value.
- void **addHeader**(`String name`, `String value`)  
Adds a response header with the given name and value.
- void **addIntHeader**(`String name`, `int value`)  
Adds a response header with the given name and integer value.
- boolean **containsHeader**(`String name`)  
Returns a boolean indicating whether the named response header has already been set.
- String **encodeRedirectURL**(`String url`)  
Encodes the specified URL for use in the `sendRedirect` method or, if encoding is not needed, returns the URL unchanged.
- String **encodeURL**(`String url`)  
Encodes the specified URL by including the session ID in it, or, if encoding is not needed, returns the URL unchanged.

- void **sendError**(int sc)  
Sends an error response to the client using the specified status code and clearing the buffer.
- void **sendError**(int sc, String msg)  
Sends an error response to the client using the specified status.
- void **sendRedirect**(String location)  
Sends a temporary redirect response to the client using the specified redirect location URL.
- void **setDateHeader**(String name, long date)  
Sets a response header with the given name and date value.
- void **setHeader**(String name, String value)  
Sets a response header with the given name and value.
- void **setIntHeader**(String name, int value)  
Sets a response header with the given name and integer value.
- void **setStatus**(int sc)  
Sets the status code for this response.

## Servlet Example

```
package pkg;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Servlet extends HttpServlet {
    public void doGet(HttpServletRequest req,
                      HttpServletResponse res)
        throws IOException, ServletException {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println("<html><head><title>Sample Servlet");
        out.println("</title></head><body>");
        out.println("<h1>Hello World at " +
                    req.getRequestURI() + " !</h1>");
        out.println("<p>Key is " + req.getParameter("KEY"));
        out.println("</p></body></html>");
    }
}
```





## SESSION TRACKING APPROACHES



### Why Session Tracking?

- In particular, when you are doing on-line shopping, it is a real annoyance that the Web server can't easily remember previous transactions.
- This makes applications like shopping carts very problematic: when you add an entry to your cart, how does the server know what's already in your cart?
- Even if servers did retain contextual information, you'd still have problems with e-commerce.
- When you move from the page where you specify what you want to buy (hosted on the regular Web server) to the page that takes your credit card number and shipping address (hosted on the secure server that uses SSL), how does the server remember what you were buying?



## What is a Session?

- A *session* is defined as a series of related browser requests that come from the same client during a certain time period.
- Session tracking enables you to track a user's progress over multiple servlets or HTML pages, which, by nature, are stateless.
- Session tracking ties together a series of browser requests—think of these requests as pages—that may have some meaning as a whole, such as a shopping cart application.



## Session Tracking....

- HTTP is a "stateless" protocol which means each time a client retrieves a Web page, the client opens a separate connection to the Web server and the server automatically does not keep any record of previous client request.
- Still there are following three ways to maintain session between web client and web server:



## Session Tracking Approaches

- Cookies
- URL Rewriting
- Hidden Form Fields
- Session API



## Cookies

- A webserver can assign a unique session ID as a cookie to each web client and for subsequent requests from the client they can be recognized using the recieved cookie.
- This may not be an effective way because many time browser does not support a cookie, so it is not recommended to use this procedure to maintain the sessions.
- Cookies can be referenced by Java Servlet using the cookie API.
- It is composed of two parts: Cookie name and cookie value.
- Ex., `Cookie mycookie = new Cookie("userid","123")`

## Hidden Form Fields:

- A web server can send a hidden HTML form field along with a unique session ID as follows:
- `<input type="hidden" name="sessionid" value="12345">`  
This entry means that, when the form is submitted, the specified name and value are automatically included in the GET or POST data. Each time when web browser sends request back, then session\_id value can be used to keep the track of different web browsers.
- This could be an effective way of keeping track of the session but clicking on a regular (`<A HREF...>`) hypertext link does not result in a form submission, so hidden form fields also cannot support general session tracking.

## URL Rewriting

- You can append some extra data on the end of each URL that identifies the session, and the server can associate that session identifier with data it has stored about that session.
- For example, with **`http://anysite.com/file.htm;sessionid=12345`**, the session identifier is attached as `sessionid=12345` which can be accessed at the web server to identify the client.
- URL rewriting is a better way to maintain sessions and works for the browsers when they don't support cookies but here drawback is that you would have generate every URL dynamically to assign a session ID though page is simple static HTML page.



## The HttpSession Interface

- Apart from the above mentioned three ways, servlet provides HttpSession Interface which provides a way to identify a user across more than one page request or visit to a Web site and to store information about that user.
- The servlet container uses this interface to create a session between an HTTP client and an HTTP server. The session persists for a specified time period, across more than one connection or page request from the user.




## HttpSession

- `HttpSession session = request.getSession();`
- You need to call `request.getSession()` before you send any document content to the client.
- Here is a summary of the important methods available through HttpSession object:



## Method Summary

- **public Object getAttribute(String name)**  
This method returns the object bound with the specified name in this session, or null if no object is bound under the name.
- **public Enumeration getAttributeNames()**  
This method returns an Enumeration of String objects containing the names of all the objects bound to this session.
- **public long getCreationTime()**  
This method returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.
- **public String getId()**  
This method returns a string containing the unique identifier assigned to this session.
- **public long getLastAccessedTime()**  
This method returns the last time the client sent a request associated with this session, as the number of milliseconds since midnight January 1, 1970 GMT.

- 
- **public int getMaxInactiveInterval()**  
This method returns the maximum time interval, in seconds, that the servlet container will keep this session open between client accesses.
  - **public void invalidate()**  
This method invalidates this session and unbinds any objects bound to it.
  - **public boolean isNew()**  
This method returns true if the client does not yet know about the session or if the client chooses not to join the session.
  - **public void removeAttribute(String name)**  
This method removes the object bound with the specified name from this session.
  - **public void setAttribute(String name, Object value)**  
This method binds an object to this session, using the name specified.
  - **public void setMaxInactiveInterval(int interval)**  
This method specifies the time, in seconds, between client requests before the servlet container will invalidate this session.



## Servlet Collaboration

- Sometimes servlets have to cooperate, usually by sharing some information. We call communication of this sort servlet collaboration.
- Collaborating servlets can pass the shared information directly from one servlet to another through method invocations, as shown earlier. This approach requires each servlet to know the other servlets with which it is collaborating--an unnecessary burden. There are several better techniques.



`javax.servlet`

## Interface **RequestDispatcher**

- Defines an object that receives requests from the client and sends them to any resource (such as a servlet, HTML file, or JSP file) on the server. The servlet container creates the `RequestDispatcher` object, which is used as a wrapper around a server resource located at a particular path or given by a particular name.
- This interface is intended to wrap servlets, but a servlet container can create `RequestDispatcher` objects to wrap any type of resource.

## Method Summary

- void **forward**(ServletRequest request, ServletResponse response)  
Forwards a request from a servlet to another resource (servlet, JSP file, or HTML file) on the server.
- void **include**(ServletRequest request, ServletResponse response)  
Includes the content of a resource (servlet, JSP page, HTML file) in the response.

javax.servlet

## Interface ServletContext


- Defines a set of methods that a servlet uses to communicate with its servlet container, for example, to get the MIME type of a file, dispatch requests, or write to a log file.
- There is one context per "web application" per Java Virtual Machine. (A "web application" is a collection of servlets and content installed under a specific subset of the server's URL namespace such as /catalog and possibly installed via a .war file.)
- In the case of a web application marked "distributed" in its deployment descriptor, there will be one context instance for each virtual machine. In this situation, the context cannot be used as a location to share global information (because the information won't be truly global). Use an external resource like a database instead.
- The ServletContext object is contained within the ServletConfig object, which the Web server provides the servlet when the servlet is initialized.





## Method Summary

- ServletContext **getContext**(String uripath)  
Returns a ServletContext object that corresponds to a specified URL on the server.
- RequestDispatcher **getRequestDispatcher**(String path)  
Returns a RequestDispatcher object that acts as a wrapper for the resource located at the given path.
- String **getServerInfo**()  
Returns the name and version of the servlet container on which the servlet is running.
- String **getInitParameter**(String name)  
Returns a String containing the value of the named context-wide initialization parameter, or null if the parameter does not exist.

- 
- Enumeration **getInitParameterNames**()  
Returns the names of the context's initialization parameters as an Enumeration of String objects, or an empty Enumeration if the context has no initialization parameters.
  - Object **getAttribute**(String name)  
Returns the servlet container attribute with the given name, or null if there is no attribute by that name.
  - void **setAttribute**(String name, Object object)  
Binds an object to a given attribute name in this servlet context.
  - void **removeAttribute**(String name)  
Removes the attribute with the given name from the servlet context.

## How to run a Servlet???

- Set path of **servlet-api in compiler's classpath.**
- Compile servlet class.
- Prepare/Update Web Application Deployment Descriptor file **web.xml.**
- Place the class file of servlet and web.xml at a proper place in directory structure followed by a specific web server directory for your web application.
- Start the web server.
- Run servlet in the web browser.

## Setting path of servlet api:

- My Computer-Properties-Advanced-Environment Variables-Path/Classpath variable to be edited.
- You can find servlet.jar (in Tomcat 4.0 or below versions) or servlet-api.jar (in Tomcat 5.0 and higher versions) at \$CATALINA\_HOME (Apache/Tomcat/) /common/lib directory.
- Place the fully qualified path of this servlet api along with its name in that Classpath variable.



## Compile the servlet:

- As servlet is after all a java class only, the compilation process remains the same here as with any simple java class.
- On the command prompt:
  - `c:/>javac ServletClass.java`
- Hence you will get a class file with the same filename as of the java file and at the same location of that java file.



## Dealing with web.xml:

- Here you are specifying description of all the servlets you have build for your web application.

```
<?xml version="1.0">
```

```
<web-app>
```

```
...
```

```
<servlet>
```

```
<servlet-name>Demo</servlet-name>
```

```
<servlet-class>DemoServlet</servlet-class>
```

```
</servlet>
```

```
...
```

```
</web-app>
```



## Dealing with web.xml:

- Here, servlet-name and servlet-class can be same or different as per the choice of user.
- Servlet-name is how would you like to identify your servlet-class.
- It is also possible to specify a mapping of URIs to servlets with the `<servlet-mapping>` element in web.xml.
- In most case, modifying the web.xml file requires either the Web Application or servlet container to be restarted before any changes take effect.



## Placing servlet class file and web.xml at proper place:

- web.xml file is to be placed in the WEB-INF folder of your web application.
  - webapps/DemoApplication/WEB-INF/web.xml
- Class file of your servlet is to be placed in classes folder i.e. residing inside the WEB-INF folder of your web application.
  - webapps/ DemoApplication/ WEB-INF/ classes/DemoServlet.class



## Starting and Stopping Web Server:

- In the web server directory, you can find files in the .exe or .bat form to start and stop the services of web server.
- For Tomcat, you can find these files in \$CATALINA\_HOME/bin directory.
- You can start the web server by executing the startup file over there and then can execute your web application in the browser.
- After closing the web application, you can stop the web browser by relevant shutdown file in the same bin directory.



## Run servlet in the web browser:

- You need to specify a URL in the address bar of your web browser to invoke your web application.

- The general format is as given below:

`http://<servername>/<webappname>/servlet/<servletname>`

E.g.

`http://localhost:8080/DemoApp/servlet/Demo`