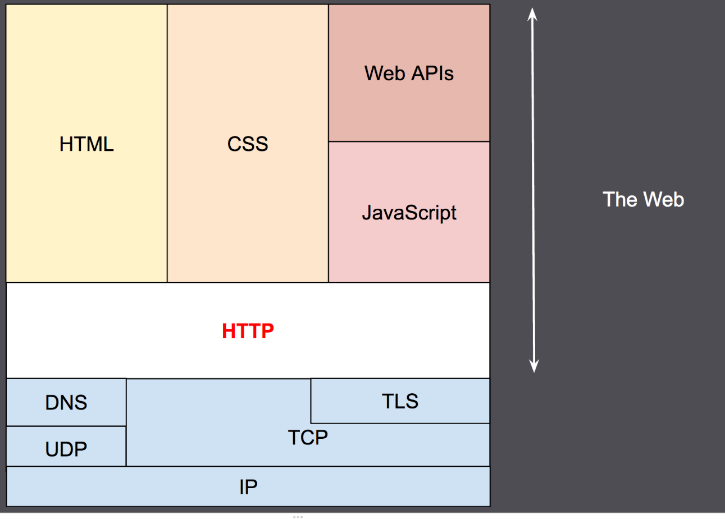
# HTTP

The Hypertext Transfer Protocol (HTTP) is based on the client-server architecture model and a stateless request/response protocol that operates by exchanging messages across a reliable TCP/IP connection.



## HTTP is human-readable, extensible, stateless and connection-oriented

HTTP is a protocol which allows the fetching of resources, such as HTML documents.  It is the foundation of any data exchange on the Web and it is a client-server protocol, which means requests are initiated by the recipient, usually the Web browser. A complete document is reconstructed from the different sub-documents fetched, for instance text, layout description, images, videos, scripts, and more.

### HTTP feature

[HTTP is simple](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#http_is_simple)

HTTP is generally designed to be simple and human readable, even with the added complexity introduced in HTTP/2 by encapsulating HTTP messages into frames. HTTP messages can be read and understood by humans, providing easier testing for developers, and reduced complexity for newcomers.

[HTTP is extensible](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#http_is_extensible)

Introduced in HTTP/1.0, [HTTP headers](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers) make this protocol easy to extend and experiment with. New functionality can even be introduced by a simple agreement between a client and a server about a new header's semantics.

[HTTP is stateless, but not sessionless](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#http_is_stateless_but_not_sessionless)

HTTP is stateless: there is no link between two requests being successively carried out on the same connection. This immediately has the prospect of being problematic for users attempting to interact with certain pages coherently, for example, using e-commerce shopping baskets. But while the core of HTTP itself is stateless, HTTP cookies allow the use of stateful sessions. Using header extensibility, HTTP Cookies are added to the workflow, allowing session creation on each HTTP request to share the same context, or the same state.

[HTTP and connections](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#http_and_connections)

A connection is controlled at the transport layer, and therefore fundamentally out of scope for HTTP. Though HTTP doesn't require the underlying transport protocol to be connection-based; only requiring it to be *reliable*, or not lose messages (so at minimum presenting an error). Among the two most common transport protocols on the Internet, TCP is reliable and UDP isn't. HTTP therefore relies on the TCP standard, which is connection-based.

Before a client and server can exchange an HTTP request/response pair, they must establish a TCP connection, a process which requires several round-trips. The default behavior of HTTP/1.0 is to open a separate TCP connection for each HTTP request/response pair. This is less efficient than sharing a single TCP connection when multiple requests are sent in close succession.

In order to mitigate this flaw, HTTP/1.1 introduced *pipelining* (which proved difficult to implement) and *persistent connections*: the underlying TCP connection can be partially controlled using the [Connection](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Connection) header. HTTP/2 went a step further by multiplexing messages over a single connection, helping keep the connection warm and more efficient.

### [HTTP flow](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#http_flow)

When a client wants to communicate with a server, it performs the following steps:

1. Open a TCP connection: The TCP connection is used to send a request, or several, and receive an answer. The client may open a new connection, reuse an existing connection, or open several TCP connections to the servers.
2. Send an HTTP message: HTTP messages (before HTTP/2) are human-readable. With HTTP/2, these simple messages are encapsulated in frames, making them impossible to read directly, but the principle remains the same. For example:

GET / HTTP/1.1

Host: developer.mozilla.org

Accept-Language: fr

1. Read the response sent by the server, such as:

HTTP/1.1 200 OK

Date: Sat, 09 Oct 2010 14:28:02 GMT

Server: Apache

Last-Modified: Tue, 01 Dec 2009 20:18:22 GMT

ETag: "51142bc1-7449-479b075b2891b"

Accept-Ranges: bytes

Content-Length: 29769

Content-Type: text/html

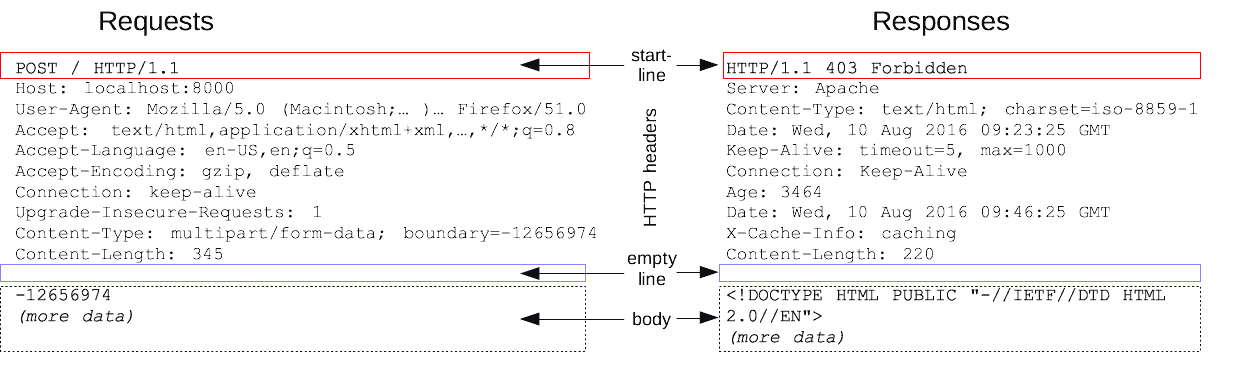
<!DOCTYPE html... (here comes the 29769 bytes of the requested web page)

1. Close or reuse the connection for further requests.

## HTTP message

### HTTP message consists of: request/status line, header (field:value), [body].

* **Start-line** (request/status)
* **Headers**: each header is a “field:value” pair (followed by a CRLF)
* An empty line (indicating the end of the headers)
* **Body** (optional)



#### Start-Line

A start-line will have the following generic syntax:

start-line = Request-Line | Status-Line

Exp:

GET /hello.htm HTTP/1.1 (This is Request-Line sent by the client)

HTTP/1.1 200 OK (This is Status-Line sent by the server)

HTTP status codes are grouped into 5 groups:

1. Informational responses (100–199)
2. Successful responses (200–299)
3. Redirects (300–399)
4. Client errors (400–499)
5. Server errors (500–599)

#### Headers

HTTP headers provide required information about the request or response, or about the object sent in the message body. Each header is of this form:

message-header = field-name ":" [ field-value ]

Exp:

User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.7l zlib/1.2.3

Host: www.example.com

Accept-Language: en, mi

Date: Mon, 27 Jul 2009 12:28:53 GMT

Server: Apache

Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT

ETag: "34aa387-d-1568eb00"

Accept-Ranges: bytes

Content-Length: 51

Vary: Accept-Encoding

Content-Type: text/plain

#### Body

The message body part is optional for an HTTP message but if it is available, then it is used to carry the entity-body associated with the request or response. If entity body is associated, then usually **Content-Type** and **Content-Length** headers lines specify the nature of the body associated.

A message body is the one which carries the actual HTTP request data (including form data and uploaded, etc.) and HTTP response data from the server (including files, images, etc.). Shown below is the simple content of a message body:

<html>

<body>

<h1>Hello, World!</h1>

</body>

</html>

### Examples of HTTP messages

#### Example 1: GET request and response with HTML

HTTP request to fetch **hello.htm** page from the web server running on tutorialspoint.com.

**Client request**

GET /hello.htm HTTP/1.1

User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)

Host: www.tutorialspoint.com

Accept-Language: en-us

Accept-Encoding: gzip, deflate

Connection: Keep-Alive

**Server response**

HTTP/1.1 200 OK

Date: Mon, 27 Jul 2009 12:28:53 GMT

Server: Apache/2.2.14 (Win32)

Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT

Content-Length: 88

Content-Type: text/html

Connection: Closed

<html>

<body>

<h1>Hello, World!</h1>

</body>

</html>

#### Example 2: request with forms

Brower can request with a form either by GET or POST method.

Assume your browser is displaying this HTML page, which is a form with 2 textboxes named “name” and “message” and a “submit” button.

<html>

<form method="GET" action="example/message.html">

<p>Name: <input type="text" name="name"></p>

<p>Message: <textarea name="message"></textarea></p>

<p><button type="submit">Send</button></p>

</form>

</html>

You type “Jean” in “name”, “Yes?” (? ss encoded as 3F) in “message”, and click on “submit” button, the browser will generate this HTTP request (notes that method=”GET” in the form):

GET /example/message.html?name=Jean&message=Yes%3F HTTP/1.1

If the HTML form above uses “POST”, the HTTP request will be:

POST /example/message.html HTTP/1.1

Content-length: 24

Content-type: application/x-www-form-urlencoded

name=Jean&message=Yes%3F

#### Example 3: GET request and response with JSON

Clients can request JSON or XML from the server by sending HTTP GET requests with header: **Accept: application/json** or **Accept: application/xml** header. If the client can handle both types of content, it can list them all in the Accept header, separated by a comma. In the server response, the Content-Type header tells the client the type of returned content; for JSON files, it is Content-Type: application/json.  
**Client request pets API:**

GET /animal/pets HTTP/1.1

Host: livingthings.com

Accept: application/json

**Server response** with an uncompressed JSON payload that's similar to the following:

HTTP/1.1 200 OK

Content-Type: application/json

[

{

"id": 1,

"type": "dog",

"price": 249.99

},

{

"id": 2,

"type": "cat",

"price": 124.99

},

{

"id": 3,

"type": "fish",

"price": 0.99

}

]

To receive this output as a compressed payload, your API client can submit a request as follows:

GET /animal/pets HTTP/1.1

Host: livingthings.com

Accept-Encoding:gzip

The client receives the response with a Content-Encoding header and GZIP-encoded payload that are similar to the following:

HTTP/1.1 200 OK

Content-Encoding:gzip

...

���RP�

J�)JV

�:P^IeA\*������+(�L �X�YZ�ku0L0B7!9��C#�&����Y��a���^�X

#### Example 4: POST request and response with HTML and Cookies

HTTP request to post form data to **process.cgi** CGI page on a web server running on tutorialspoint.com. The server returns the passed name after setting them as cookies:

**Client request**

POST /cgi-bin/process.cgi HTTP/1.1

User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)

Host: www.tutorialspoint.com

Content-Type: text/xml; charset=utf-8

Content-Length: 60

Accept-Language: en-us

Accept-Encoding: gzip, deflate

Connection: Keep-Alive

first=Zara&last=Ali

**Server response**

HTTP/1.1 200 OK

Date: Mon, 27 Jul 2009 12:28:53 GMT

Server: Apache/2.2.14 (Win32)

Content-Length: 88

**Set-Cookie: first=Zara,last=Ali;domain=tutorialspoint.com;Expires=Mon, 19-**

**Nov-2010 04:38:14 GMT;Path=/**

Content-Type: text/html

Connection: Closed

<html>

<body>

<h1>Hello Zara Ali</h1>

</body>

</html>

### GET (data in start-line) vs POST (data in body)

Methods are indicated in only request messages (not response messages); and method is at the first position of a request message’s start line. Methods of HTTP consist of: GET, POST, PUT, HEAD, DELETE, PATCH, OPTIONS.

GET and POST are the most popular.

**GET method:** the data is placed to the **start-line** of the HTTP request message. Browsers show the data in the URL box.

GET/RegisterStudent.asp?**user=jhon&pass=java** HTTP/1.1

Host: www.guru99.com

**POST method:** the data is placed in the **body** of the HTTP request message. Browsers don’t show the data in the URL box.

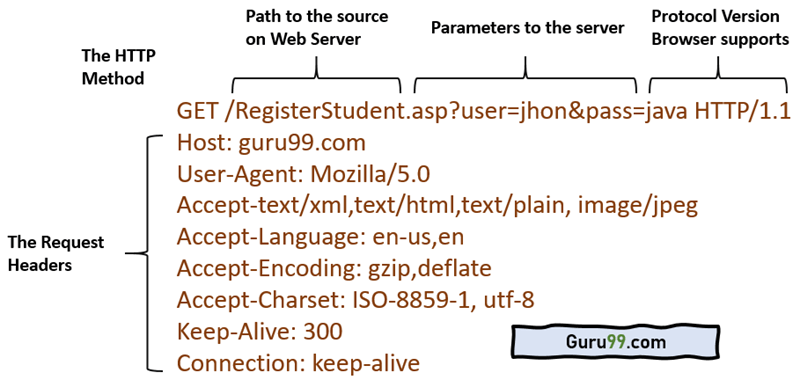
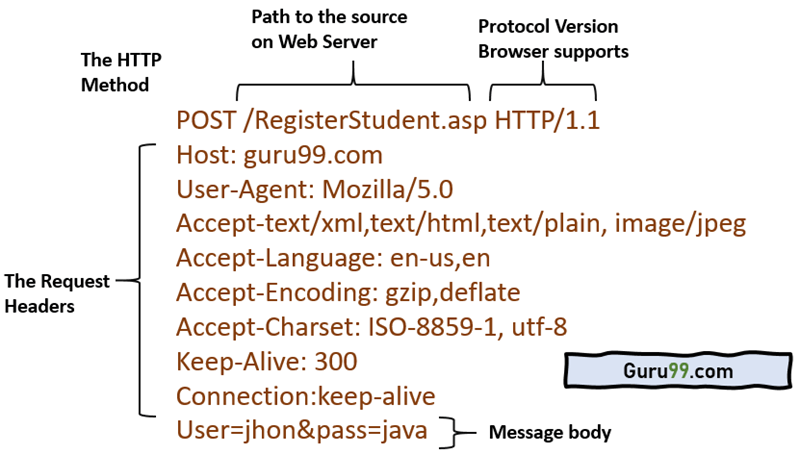
POST/RegisterStudent.asp HTTP/1.1

Host: www.guru99.com

Content-Type: application/x-www-form-urlencoded

Content-Length: 27

**user=jhon&pass=java**

## HTTP headers

How to view headers

While the body, holds the data message (HTML, JSON) in the Response, or form fields in the Request, the headers let the client and the server to pass essential information about each other.

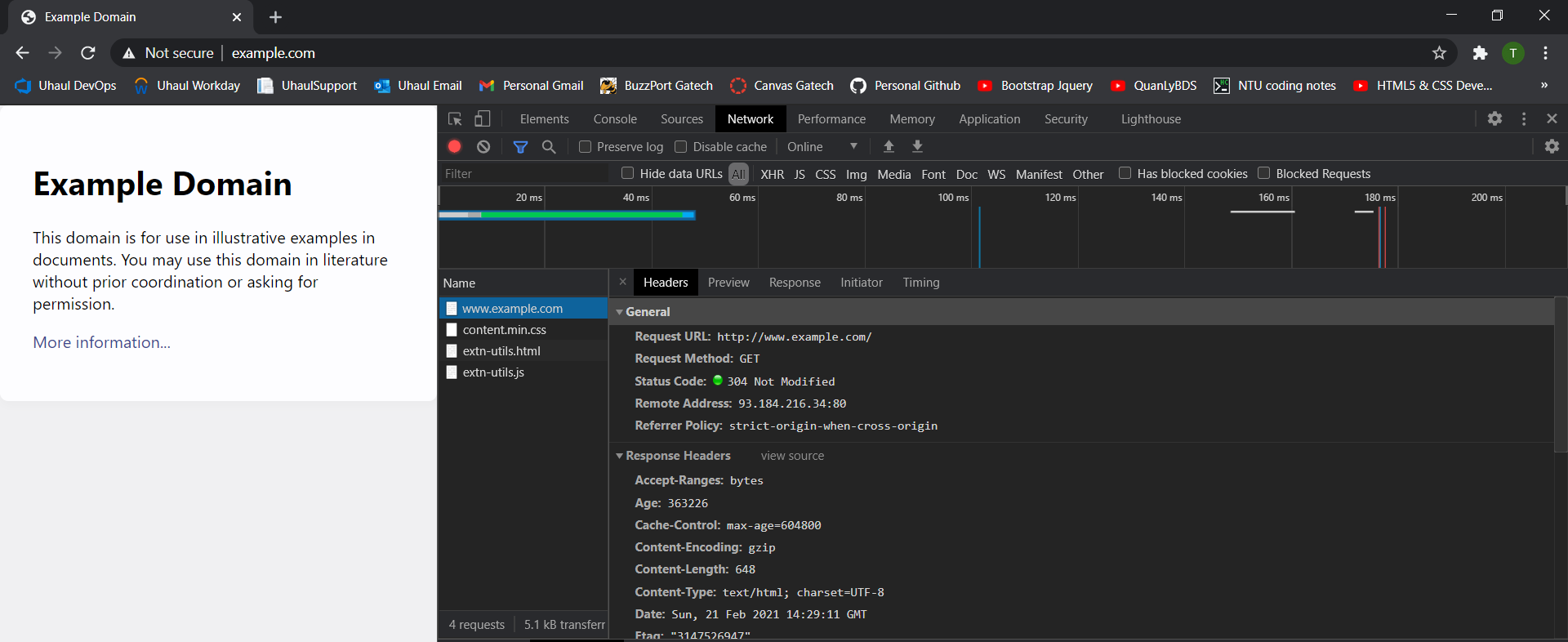
Headers can be grouped into four categories by their context:

* **General headers** contain information that is relevant for both Request and Response, but no information about the data in a body
* **Request headers** hold information about the client and requested resource
* **Response headers** include server details, like time, location, configuration
* **Entity header** informs browser about the type and body of the resource

Google Chrome’s inspect tool to view headers:

* Go to the webpage [www.example.com](http://www.example.com/), right click > Inspect > Network tab
* Select the document to inspect headers.

Google Chrome also classifies headers as 4 groups mentioned above.



### General headers

* **Request URL**: The address of the Request and Response
* **Request Method**: A method that is used for the operation, like GET, POST, PUT or DELETE
* **Status Code**: One of the most critical information that tells the status of the request/response.

Status codes are grouped:  
*1xx - Informational; the request is processing  
2xx - Success; received, accepted, created  
3xx - Redirect; actions needed, moved to a new location  
4xx - Client Error; bad request, unauthorized or not found  
5xx - Server Error; server failed to fulfill the request, internal server error*

* **Remote Address**: The IP address of the server

### Request Headers

* **Accept**: Informs the server, what data types can be accepted, describes the content format.

For example:  
*audio/ogg indicates an audio file  
image/png - an image file  
text/html - HTML file  
application/json - data in the JSON format*

* **Accept-Encoding**: An algorithm, such as compression that is used on the recourse sent back.

For example: gzip, deflate

* **Accept-Language**: Hints the server about the expected language.

For example: en-US,en

* **Connection**: Controls how long connection should stay open. For example: keep-alive
* **Host**: The domain name of the server. For example: example.com
* **User-Agent**: Lets server to identify the characteristics of the application, OS, vendor, and versions of the requesting agent.   
  For example: Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_15\_4)
* **Cookie**: 'cookie-list'  
  Contains stored piece of information, previously sent by the server. For example: Cookie: name=value; name2=value2; name3=value3
* **Authorization**: 'type' 'credentials'  
  Includes credentials to authenticate a user with a server. The two most used types are Basic, for base64-encoded credentials, and Bearer for access tokens.
* **Referer**: 'url'  
  Contains the address of the previous page, from which the user was linked to the current page.

### Response headers

* **Age**: 270773  
  Time in seconds how long the object was in the proxy cache
* **Cache-Control**: max-age=604800  
  Set the instruction for caching. Other setting types: no-cache, no-store, no-transform
* **Date**: Sun, 12 Apr 2020 16:49:25 GMT  
  The time when the message was created
* **Expires**: Sun, 19 Apr 2020 16:49:25 GMT  
  Sets the date when the relevant content will no longer be new/fresh
* **Server**: ECS (nyb/1D2C)  
  Specifies the software used by the server at the time of the sent Response
* **X-Cache**: HIT  
  It means that the request was sent not from the origin servers, but from an exclusive network (CDN), designed to cache content, so the user could get Response faster.
* **Set-Cookie**: 'cookie-name=cookie-value'  
  Sent cookies from the server to the user-agent. May include other cookie settings, such as expiration date, max-age, domain, security. For example: Set-Cookie: id=qwerty123; Expires=Wed, 13 Apr 2020 07:00:00 GMT.

### Entity headers

* **Content-Encoding**: gzip  
  Specifies the compression algorithm used for the response body
* **Content-Length**: 648  
  The size of the recourse in bytes
* **Content-Type**: text/html; charset=UTF-8  
  The resource type received. The current type is an HTML document.

## HTTP cookies

An HTTP cookie (web cookie, browser cookie) is a small piece of data that a server sends to the user's web browser. The browser may store it and send it back with later requests to the same server. Typically, it's used to tell if two requests came from the same browser — keeping a user logged-in, for example. It remembers stateful information for the [stateless](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#http_is_stateless_but_not_sessionless) HTTP protocol.

Cookies are mainly used for three purposes:

**Session management:** Logins, shopping carts, game scores, or anything else the server should remember

**Personalization:** User preferences, themes, and other settings

**Tracking:** Recording and analyzing user behavior

Cookies are sent with every request, so they can worsen performance (especially for mobile data connections). Cookies were once used for general client-side storage, and now it’s recommended to use modern storage APIs, like [Web Storage API](https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API) (localStorage and sessionStorage) and [IndexedDB](https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API).

### Creating cookies

After receiving an HTTP request, a server can send one or more [Set-Cookie](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie) headers with the response. The cookie is usually stored by the browser, and then the browser sends request to the same server with the cookie inside a [Cookie](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cookie) HTTP header. An expiration date or duration can be specified, after which the browser stop sending the cookie. Additional restrictions to a specific domain and path can be set, limiting where the cookie is sent.

#### Headers: “Set-Cookie” (in Response) and “Cookie” (in Request)

The [Set-Cookie](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie) HTTP response header sends cookies from the server to the user agent. A simple cookie is set like this:

Set-Cookie: <cookie-name>=<cookie-value>

This shows the server sending headers to tell the client to store a pair of cookies:

HTTP/2.0 200 OK

Content-Type: text/html

Set-Cookie: yummy\_cookie=choco

Set-Cookie: tasty\_cookie=strawberry

[page content]

Then, with every subsequent request to the server, the browser sends back all previously stored cookies to the server using the [Cookie](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cookie) header.

GET /sample\_page.html HTTP/2.0

Host: www.example.org

Cookie: yummy\_cookie=choco; tasty\_cookie=strawberry

#### Define the lifetime of a cookie

The lifetime of a cookie can be defined in two ways:

* **Session** cookies are deleted when the current session ends. The browser defines when the "current session" ends, and some browsers use session restoring when restarting, which can cause session cookies to last indefinitely long.
* **Permanent** cookies are deleted at a date specified by the Expires attribute, or after a period of time specified by the Max-Age attribute.

For example:

Set-Cookie: id=a3fWa; Expires=Thu, 31 Oct 2021 07:28:00 GMT;

Note: When an Expires date is set, the time and date set is relative to the client the cookie is being set on, not the server.

If your site authenticates users, it should regenerate and resend session cookies, even ones that already exist, whenever the user authenticates. This technique helps prevent [session fixation attacks](https://developer.mozilla.org/en-US/docs/Web/Security/Types_of_attacks#session_fixation), where a third party can reuse a user's session.

#### Restrict access to cookies

There are a couple of ways to ensure that cookies are sent securely and are not accessed by unintended parties or scripts: the Secure attribute and the HttpOnly attribute.

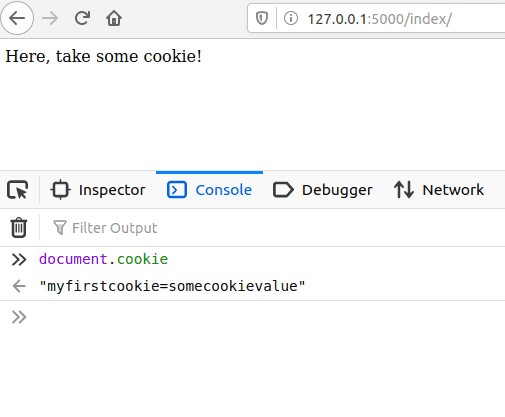
A cookie with the Secure attribute is sent to the server only with an encrypted request over the HTTPS protocol, never with unsecured HTTP (except on localhost), and therefore can't easily be accessed by a [man-in-the-middle](https://developer.mozilla.org/en-US/docs/Glossary/MitM) attacker. Insecure sites (with http: in the URL) can't set cookies with the Secure attribute. However, do not assume that Secure prevents all access to sensitive information in cookies; for example, it can be read and modified by someone with access to the client's hard disk (or JavaScript if the HttpOnly attribute is not set).

A cookie with the HttpOnly attribute is inaccessible to the JavaScript [Document.cookie](https://developer.mozilla.org/en-US/docs/Web/API/Document/cookie) API; it is sent only to the server. For example, cookies that persist server-side sessions don't need to be available to JavaScript, and should have the HttpOnly attribute. This precaution helps mitigate cross-site scripting ([XSS](https://developer.mozilla.org/en-US/docs/Web/Security/Types_of_attacks#cross-site_scripting_(xss))) attacks. Here is an example:

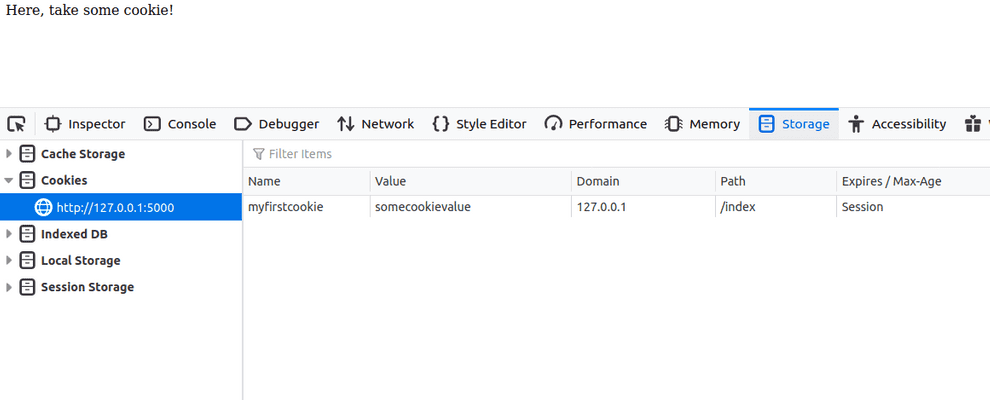
Set-Cookie: id=a3fWa; Expires=Thu, 21 Oct 2021 07:28:00 GMT; Secure; HttpOnly

### How to see cookies?

Assume you visit <http://127.0.0.1:5000/index/>, the backend sets a cookie in the browser. To see this cookie you can either call document.cookie from the browser's console:



Or you can check the **Storage** tab in the developer tools. Click on **Cookies**, and you should see the cookie there:



On a command line you can use also **curl** to see what cookies the backend sets:

curl -I http://127.0.0.1:5000/index/

To save cookies to a file for later use:

curl -I http://127.0.0.1:5000/index/ --cookie-jar mycookies

To display cookies on stdout:

curl -I http://127.0.0.1:5000/index/ --cookie-jar -

Note that cookies without the HttpOnly attribute are accessible on document.cookie from JavaScript in the browser. On the other hand a cookie marked as HttpOnly cannot be accessed from JavaScript.

To mark a cookie as HttpOnly pass the attribute in the cookie:

Set-Cookie: myfirstcookie=somecookievalue; HttpOnly

Now the cookie will still appear in the Cookie Storage tab, but document.cookie will return an empty string.

### More on cookies

#### Define where cookies are sent

The Domain and Path attributes define the scope of the cookie: what URLs the cookies should be sent to.

##### Domain attribute

The Domain attribute specifies which hosts are allowed to receive the cookie. If unspecified, it defaults to the same [host](https://developer.mozilla.org/en-US/docs/Glossary/Host) that set the cookie, excluding subdomains. If Domain is specified, then subdomains are always included. Therefore, specifying Domain is less restrictive than omitting it. However, it can be helpful when subdomains need to share information about a user.

For example, if Domain=mozilla.org is set, then cookies are available on subdomains like developer.mozilla.org.

##### Path attribute

The Path attribute indicates a URL path that must exist in the requested URL in order to send the Cookie header. The %x2F ("/") character is considered a directory separator, and subdirectories match as well.

For example, if Path=/docs is set, these paths match:

* /docs
* /docs/Web/
* /docs/Web/HTTP

##### SameSite attribute and Cookie prefix

See <https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies>

# Component-base development and Open architecture

## Component-base development

Hardware engineers create circuits by mounting pre-fabricated chips on circuit boards. Component-based development extends this idea to software engineering: software engineers should be able to create applications by deploying and connecting pre-fabricated software components into containers that provide infrastructure services such as discovery, lifecycle management, and communication. Ideally, deploying and connecting components could be done using some graphical drag-and-drop interface. There should be no need to understand how the components are implemented.

Examples:

·       Several companies provide components for enterprise resource planning (logistics, human resources, billing, etc.) and a container for these components. Customers can buy the components they need and can afford. Examples include SAP and Netsuite.

·       Servlets and Java Beans are examples of component architectures for J2EE web servers (e.g., Tomcat) and application servers (e.g., JBoss).

·       Visual Basic was the original component container. Users could drag calendar, calculator, input, and output components into the container, graphically connect them and compile. This idea later evolved into[OLE containers and components](https://en.wikipedia.org/wiki/Object_Linking_and_Embedding).

·       [Service-Oriented Architectures](https://en.wikipedia.org/wiki/Service-oriented_architecture) are closely related to Container-Component architectures. Web service architectures are an example of SOA.

Definitions

Internally, a component is a collection of encapsulated functions and data.

Externally, a component is a set of realized interfaces together with a set of required interfaces.

An interface is a set of related operations, attributes, and receptions (i.e., signal receptors or handlers).

## Open architecture

Two competitors A and B have similar products. Company A uses a proprietary architecture. This forces their customers to purchase add-on components from them or companies explicitly licensed by them. A collects the profits and controls the price. Company B uses an open architecture. They publish the interfaces implemented by their product and invite random third-party developers to build add-on components and sell them directly to A's customers. Although company B does not directly profit from the sales of these components, a large, competitive market develops. B's customers can choose from a large selection of competitively-priced components. In turn, this drives sales of B's product.

Examples:

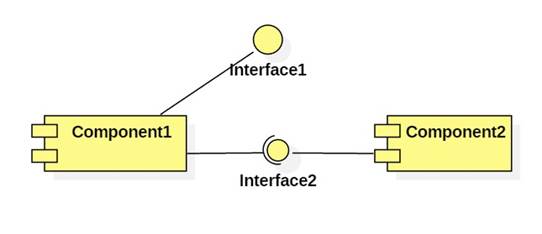
·       Companies A and B are veiled references to Apple and IBM. IBM published details of the PC motherboard, thus creating a market for expansion cards.

·       [Eclipse](https://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Farch.htm)championed the container-component architecture.

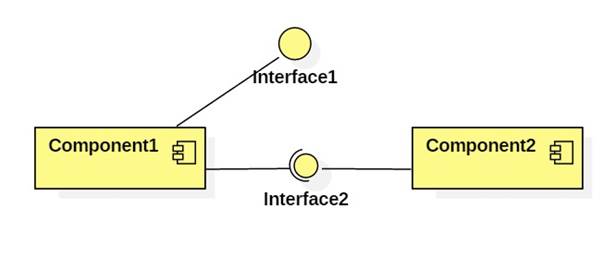
·       Most browsers employ some form of container-component architecture.

## UML Notation for container-component

In the following diagram Component1 realizes (or implements) Interface1 and requires Interface2, which is realized by Component2.



Modern UML favors using class icons with a component decoration or stereotype:



# What is Java EE (Jakarta EE) and Spring

## Jakarta EE

Since 2019, Java EE’s name has been Jakarta EE.

Java EE is indeed an **abstract** specification. Anybody is open to develop and provide a working implementation of the specification. The concrete implementations are the so-called application servers, like [WildFly](http://wildfly.org/), [TomEE](http://tomee.apache.org/), [GlassFish](http://glassfish.org/), [Liberty](http://www-03.ibm.com/software/products/en/appserv-was-liberty-core), [WebLogic](http://www.oracle.com/technetwork/middleware/weblogic/overview/index.html), etc. There are also servlet containers which implement only the JSP/Servlet part of the huge Java EE API, such as [Tomcat](http://tomcat.apache.org/), [Jetty](http://www.eclipse.org/jetty/), etc.

We, Java EE developers, should write code utilizing the specification (i.e. import only javax.\* classes in our code instead of implementation specific classes such as org.jboss.wildfly.\*, com.sun.glassfish.\*, etc) and then we'll be able to run our code on any implementation (thus, on any application server). (If you're familiar with JDBC, it's basically the same concept as how JDBC drivers work. See also a.o. [In simplest terms, what is a factory?](https://stackoverflow.com/questions/7550612/in-simplest-terms-what-is-a-factory))

Java EE SDK = Java SE SDK + GlassFish

The [Java EE SDK download](http://www.oracle.com/technetwork/java/javaee/downloads/index.html) from Oracle.com contains basically the GlassFish server along a bunch of documentation and examples and optionally also the NetBeans IDE. You don't need it if you want a different server and/or IDE.

EJB is part of the Java EE specification. Look, [it's in the Java EE API](http://download.oracle.com/javaee/6/api/javax/ejb/package-summary.html). Full-fledged Java EE application servers support it out the box, but simple JSP/Servlet containers don't.

## Spring

Spring is a standalone framework which substitutes and improves many parts of Java EE. Spring doesn't necessarily require Java EE to run. A bare-bones servlet container like Tomcat is already sufficient. Simply put, Spring is a competitor of Java EE. E.g. "Spring" (standalone) competes EJB/JTA, Spring MVC competes JSF/JAX-RS, Spring DI/IoC/AOP competes CDI, Spring Security competes JAAS/JASPIC, etc.

Back during the old J2EE/EJB2 times, the EJB2 API was terrible to implement and maintain. Spring was then a much better alternative to EJB2. But since EJB3 (Java EE 5), the EJB API was much improved based on lessons learnt from Spring. Since CDI (Java EE 6), there's not really a reason to look at again another framework like Spring to make the developers more easy as to developing among others the service layer.

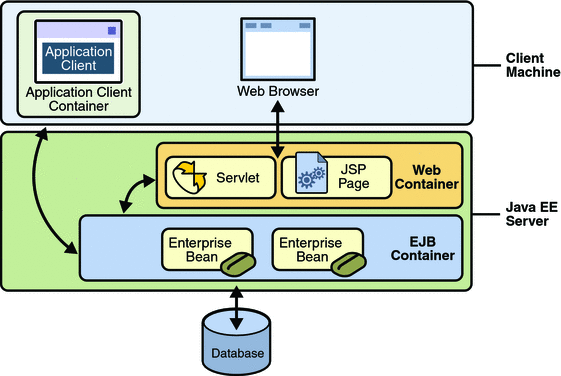
Only when you're using a bare-bones servlet container such as Tomcat and can't move on to a Java EE server, then Spring is more attractive as it's easier to install Spring on Tomcat. It isn't possible to install e.g. an EJB container on Tomcat without modifying the server itself, you would basically be reinventing TomEE.

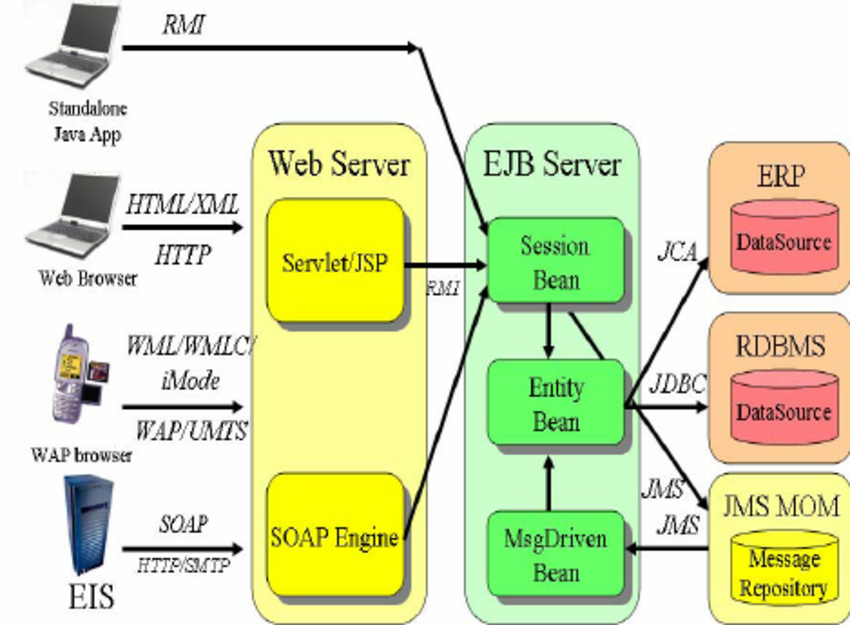
## Java EE

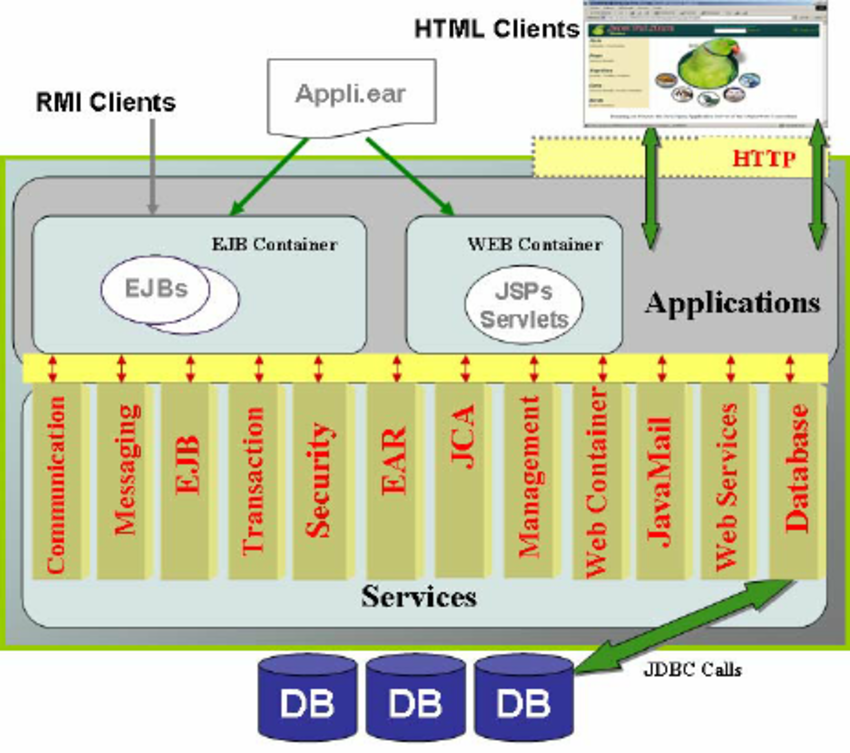
## The best tutorial

<https://docs.oracle.com/javaee/6/tutorial/doc/>

<https://javajee.com/category/java-web-java-ee>

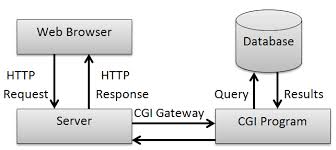






# Web server vs web container vs application server

## Web server and CGI



CGI (common gateway interface) is an interface specification between a web server (HTTP server) and an executable program of some type that is to handle a particular request. It describes how certain properties of that request should be communicated to the environment of that program and how the program should communicate the response back to the server and how the server should 'complete' the response to form a valid reply to the original HTTP request.

Programs implementing a CGI interface can be written in any language runnable on the target machine. They must be able to access environment variables and usually standard input and they generate their output on standard output. Compiled languages such as C were commonly used as were scripting languages such as Perl, often using libraries to make accessing the CGI environment easier.

One of the big disadvantages of CGI is that a new program is spawned for each request so maintaining state between requests could be a major performance issue. The state might be handled in cookies or encoded in a URL, but if it gets to large it must be stored elsewhere and keyed from encoded url information or a cookie. Each CGI invocation would then have to reload the stored state from a store somewhere. For this reason, and for a greatly simple interface to requests and sessions, better integrated environments between web servers and applications are much more popular.

## Web server vs web container vs application server

A web server is a software that serves HTTP request, usually by returning HTML pages. A pure web server can deliver only static web pages; however, with add-on modules, a web server can serve dynamic web.

A web container is a web server with servlet, JSP technology. A web container, or a servlet container, manages Java classes, called servlets, that generate dynamic HTML pages for HTTP requests. Receiving a HTTP request, a web container will call a corresponding servlet that handle the request. A more advanced technology is Java Server Page (JSP). (Note: a servlet is a Java class with HTML code, a JSP is a HMTL page with Java code).

An application server is a web server with Java EE technology; it has two components: Web container for processing presentation layer on server and Enterprise Java Bean (aka Application container) for processing business logic on server.

Most popular:

* Web server: **Apache**, NginX, IIS
* Web container: **Tomcat**, Jetty
* Application server: WebLogic (Oracle), Websphere (IBM), WildFly (Red Hat), **GlassFish** (Oracle)

## Web container (apache tomcat, jetty) vs web server (apache httpd)

"Apache" is the name of a foundation that write open-source software.

Apache web server (apache HTTPD server) is a web server written in portable C. It mostly serves static content by itself, but there are many add-on modules (some of which come with Apache itself) that let it modify the content and also serve dynamic content written in Perl, PHP, Python, Ruby, or other languages.  
  
Apache Tomcat is primarily a servlet/JSP container. It's written in [Java](http://www.javaranch.com/). It can serve static content, too, but its main purpose is to host [servlets](http://www.coderanch.com/forums/f-7/servlets) and JSPs. Although it's possible to get Tomcat to run Perl scripts and the like, you wouldn't use Tomcat unless most of your content was Java.  
  
It's actually possible to use both Apache and Tomcat together, so that Apache serves the static content, and Tomcat the Servlets and JSPs. Depending on various factors, this may or may not be a good idea.

# Tomcat

## Directories and Files

### Directories for Tomcat

* **/bin** - Startup, shutdown, and other scripts. The \*.sh files (for Unix systems) are functional duplicates of the \*.bat files (for Windows systems). Since the Win32 command-line lacks certain functionality, there are some additional files in here.
* **/conf** - Configuration files. The most important file is server.xml - main configuration file for the container.
* **/logs** - Log files are here by default.
* **/webapps** - This is where your webapps go.

### Directories for web application



The folder that stores the web application is: **webapps/ROOT**. Under **webapps/ROOT**, you should put:

**\*.html, \*.jsp, etc.** - The HTML and JSP pages, along with other files that must be visible to the client browser (such as JavaScript, stylesheet files, and images) for your application.

**/WEB-INF/web.xml** - The *Web Application Deployment Descriptor* for your application. This is an XML file describing the servlets and other components that make up your application, along with any initialization parameters and container-managed security constraints that you want the server to enforce for you.

**/WEB-INF/classes/** - This directory contains any Java class files (and associated resources) required for your application, including both servlet and non-servlet classes, that are not combined into JAR files. If your classes are organized into Java packages, you must reflect this in the directory hierarchy under /WEB-INF/classes/. For example, a Java class named com.mycompany.mypackage.MyServlet would need to be stored in a file named /WEB-INF/classes/com/mycompany/mypackage/MyServlet.class.

**/WEB-INF/lib/** - This directory contains JAR files that contain Java class files (and associated resources) required for your application, such as third party class libraries or JDBC drivers.

### CATALINA\_HOME and CATALINA\_BASE variables to store Tomcat location

Throughout the documentation, there are references to the two following properties:

* **CATALINA\_HOME**: Represents the root of your Tomcat installation, for example /home/tomcat/apache-tomcat-9.0.10 or C:\Program Files\apache-tomcat-9.0.10.
* **CATALINA\_BASE**: Represents the root of a runtime configuration of a specific Tomcat instance. By default, CATALINA\_HOME and CATALINA\_BASE point to the same directory. Set CATALINA\_BASE manually when you require running multiple Tomcat instances on one machine.

If you set CATALINA\_HOME, CATALINA to different locations, the CATALINA\_HOME location contains static sources, such as .jar files, or binary files. The CATALINA\_BASE location contains configuration files, log files, deployed applications, and other runtime requirements. Because all instances with single CATALINA\_HOME location share one set of .jar files and binary files, you can easily upgrade the files to newer version and have the change propagated to all Tomcat instances using the same CATALIA\_HOME directory.

## Install and start up/shut down Tomcat

Configuration and start-up is described in [RUNNING.txt](https://tomcat.apache.org/tomcat-9.0-doc/RUNNING.txt)file in the Tomcat folder.

You need JDK first.

To install Tomcat, you can:

* Either use the binary distribution and configure it manually
* Or use Tomcat installer and install Tomcat as a Windows service

### Use binary distribution and manually configure Tomcat.

Step 1: Configure Environment Variables

Tomcat is a Java application and does not use environment variables directly, but the scripts that starts Tomcat uses environment variables.

* 1. Set CATALINA\_HOME (required) and CATALINA\_BASE (optional)

The CATALINA\_HOME environment variable should be set to the location of the root directory of the "binary" distribution of Tomcat.

The Tomcat startup scripts often does it automatically, based on the location of the startup script in \*nix and on the current directory in Windows, s*o, it’s likely that you don’t need to do anything.*

* 1. Set JAVA\_HOME (required)

The JAVA\_HOME variable is used to specify location of a JDK that is used to start Tomcat. In Windows:

set “JAVA\_HOME=C:\path\to\JDK”

Step 2: Run Tomcat

%JAVA\_HOME%\bin\startup.bat for starting up Tomcat

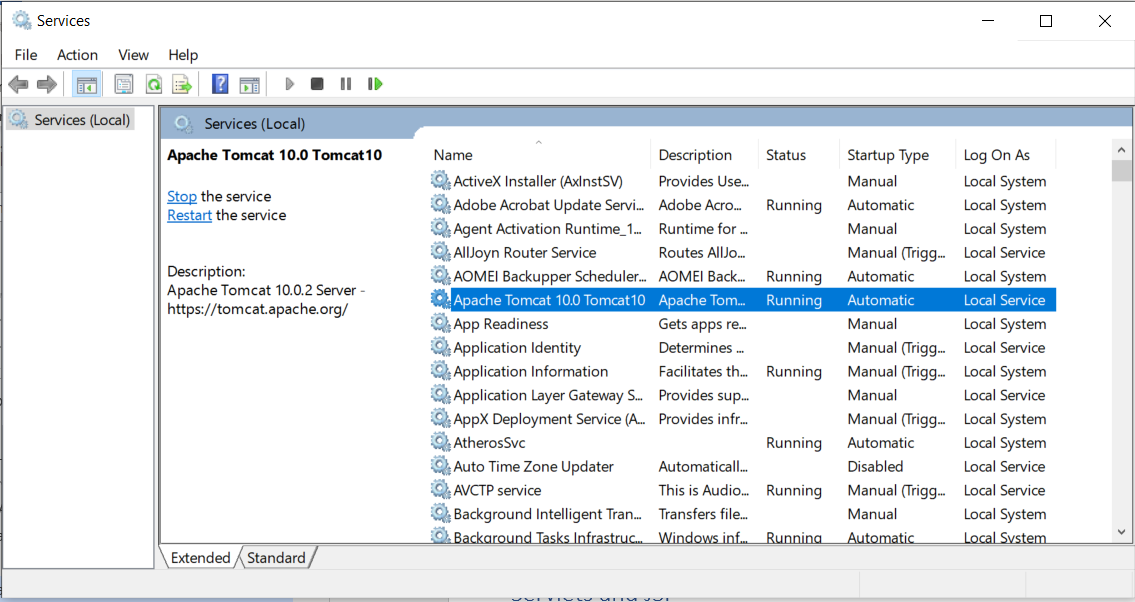
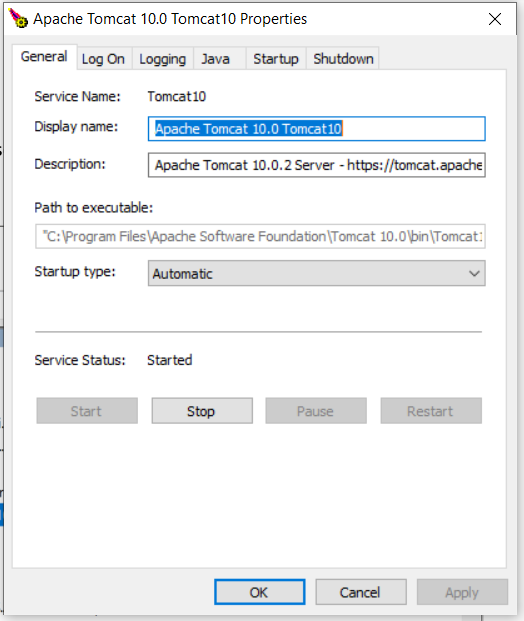
%JAVA\_HOME%\bin\shutdonw.bat for shutting down Tomcat.

Browse localhost:8080 to check if Tomcat runs.

### Install Tomcat as a Windows service

If you run Tomcat installer, it will be installed as a Windows service, so you can use Services to manage it.

Tomcat itself has a GUI for managing itself.

# JAR and WAR package format

JAR – or Java Archive – is a package file format.

## **JAR Packaging**

Simply put, JAR – or Java Archive – is a package file format. JAR files have the .jar extension and may contain **libraries, resources, and metadata files.**

Essentially, it's a zipped file containing the compressed versions of .class files and resources of compiled Java libraries and applications.

For example, here's a simple JAR file structure:

META-INF/

MANIFEST.MF

com/

baeldung/

MyApplication.class

The [META-INF/MANIFEST.MF file](https://www.baeldung.com/java-jar-executable-manifest-main-class) may contain additional metadata about the files stored in the archive.

We can [create a JAR](https://www.baeldung.com/java-create-jar) file using the jar command or with tools like [Maven](https://www.baeldung.com/executable-jar-with-maven).

## **WAR Packaging**

WAR stands for Web Application Archive or Web Application Resource. These archive files have the .war extension and are **used to package web applications** that we can deploy on any Servlet/JSP container.

Here's an example layout of a typical WAR file structure:

META-INF/

MANIFEST.MF

WEB-INF/

web.xml

jsp/

helloWorld.jsp

classes/

static/

templates/

application.properties

lib/

// \*.jar files as libs

Inside, it has a META-INF directory holding useful information in the MANIFEST.MF about the web archive. The META-INF directory is private and can't be accessed from the outside.

On the other hand, it also contains the WEB-INF public directory with all the static web resources, including HTML pages, images, and JS files. Moreover, it contains the web.xml file, servlet classes, and libraries.

We can use the same tools and commands that we used to build a JAR to build a .war archive.

## **Key Differences**

**File extension**. JARs have the .jar extension, whereas the WAR file has the .war extension.

**Purpose.** JAR files allow us to package multiple files in order to use it as a library, plugin, or any kind of application. On the other hand, WAR files are used only for web applications.

**The archive structure.** We can create a JAR with any desired structure. In contrast, WAR has a predefined structure with WEB-INF and META-INF directories.

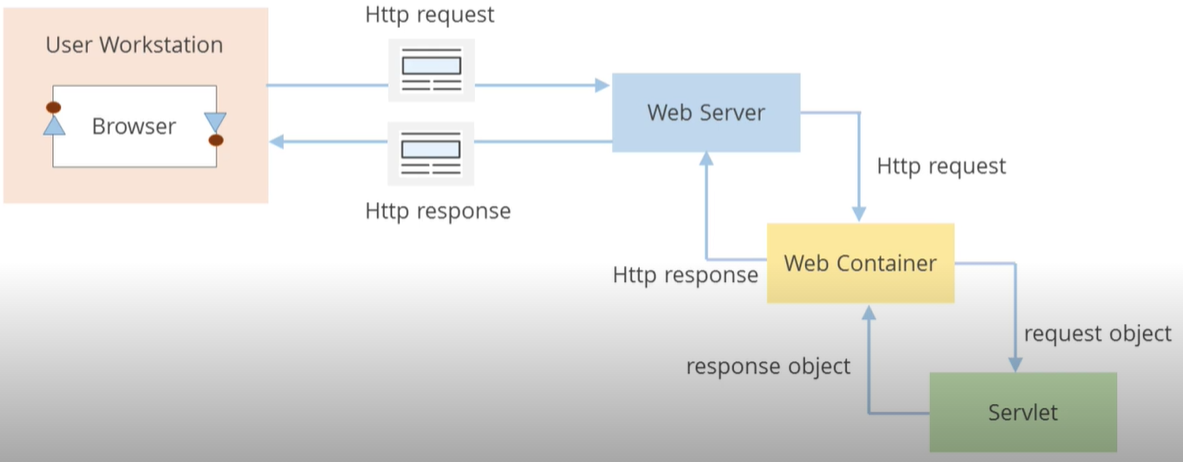
Finally, we can **run a JAR from the command line** if we build it as an [executable JAR](https://www.baeldung.com/executable-jar-with-maven) without using additional software. Or, we can use it as a library. In contrast, we **need a server to execute a WAR**.

# Servlets

## Servlet and Web container

A servlet is a Java class that generate a HTTP response.

A web container is a program that forward request/receive response to/from a servlet as the diagram below. Web container manages servlet, web container – servlet follows the container – component design.



**Difference: Servlet (thread) vs CGI (process)**

For each request, CGI creates a new process while Servlet creates a new thread. Threads have many benefits over the Processes such as they share a common memory area, lightweight, cost of communication between the threads are low.

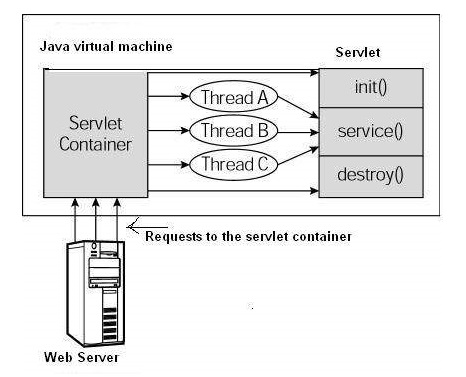




## Life cycle of servlet

A servlet life cycle can be defined as the entire process from its creation till the destruction. The web container controls servlet life cycles. The following are a a servlet’s life cycle:

* First the HTTP requests coming to the server are delegated to the servlet container (assuming that the servlet is mapped to the request).
* The servlet container create an instance of the servlet and runs **init()**.
* Then the servlet container handles multiple requests by spawning multiple threads, each thread executing the **service()** method of a single instance of the servlet.
* Eventually when the container or the application shuts down, or the container decides that there is a shortage of memory or when this servlet hasn't got a request in a long time, the servlet is terminated by calling the **destroy()** method.



### The init() Method

public void init() throws ServletException {

// Initialization code...

}

The init method is called only once. It is called only when the servlet is created, and not called for any user requests afterwards.

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 service() Method

public void service(ServletRequest request, ServletResponse response)

throws ServletException, IOException {

}

The service() method is the main method to perform the actual task. The servlet container 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.

The service () method is called by the container and service method invokes doGet, 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.

**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

public void destroy() {

// Finalization code...

}

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.

## Write a servlet

Servlets are Java classes which service HTTP requests and implement the **javax.servlet.Servlet** interface. Web application developers typically write servlets that extend javax.servlet.http.HttpServlet, an abstract class that implements the Servlet interface and is specially designed to handle HTTP requests.

**Since a servlet requires javax.servlet so you need to add a jar file to your build path.**

A servlet example to show Hello World:

// Import required java libraries

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

// Extend HttpServlet class

public class HelloWorld extends HttpServlet {

private String message;

public void init() throws ServletException {

// Do required initialization

message = "Hello World";

}

public void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

// Set response content type

response.setContentType("text/html");

// Actual logic goes here.

PrintWriter out = response.getWriter();

out.println("<h1>" + message + "</h1>");

}

public void destroy() {

// do nothing.

}

}

### javax.servlet and javax.servlet.http package

The javax.servlet and javax.servlet.http packages represent interfaces and classes for servlet api.

The javax.servlet package contains many interfaces and classes that are used by the servlet or web container. These are not specific to any protocol.

The javax.servlet.http package contains interfaces and classes that are responsible for http requests only.

|  |  |
| --- | --- |
| javax.servlet package | |
| Interfaces | Classes |
| Servlet  ServletRequest  ServletResponse  RequestDispatcher  ServletConfig  ServletContext  SingleThreadModel  Filter  FilterConfig  FilterChain  ServletRequestListener  ServletRequestAttributeListener  ServletContextListener  ServletContextAttributeListener | GenericServlet  ServletInputStream  ServletOutputStream  ServletRequestWrapper  ServletResponseWrapper  ServletRequestEvent  ServletContextEvent  ServletRequestAttributeEvent  ServletContextAttributeEvent  ServletException  UnavailableException |
| javax.servlet.http package | |
| Interfaces | Classes |
| HttpServletRequest  HttpServletResponse  HttpSession  HttpSessionListener  HttpSessionAttributeListener  HttpSessionBindingListener  HttpSessionActivationListener  HttpSessionContext (deprecated now) | HttpServlet  Cookie  HttpServletRequestWrapper  HttpServletResponseWrapper  HttpSessionEvent  HttpSessionBindingEvent  HttpUtils (deprecated now) |

### Servlet is a Java class extending GenericServlet or HttpServlet

A Java Servlet is just an ordinary Java class which implements the interface

javax.servlet.Servlet;

The easiest way to implement this interface is to **extend** either the class **GenericServlet** or **HttpServlet**.

HttpServlet specializes in Http protocol while GenericServlet is for any protocol, so for making web, HttpServlet is easier.

#### GenericServlet

import javax.servlet.GenericServlet;

import javax.servlet.ServletException;

import javax.servlet.ServletRequest;

import javax.servlet.ServletResponse;

import java.io.IOException;

public class SimpleServlet extends GenericServlet {

public void service(ServletRequest request, ServletResponse response)

throws ServletException, IOException {

// do something in here

}

}

When an HTTP request arrives at the web server, targeted for your Servlet, the web server calls your Servlet's service() method.

The service() method then reads the request, and generates a response which is sent back to the client (e.g. a browser).

Here is an example service() implementation:

public void service(ServletRequest request, ServletResponse response)

throws ServletException, IOException {

String yesOrNoParam = request.getParameter("param");

if("yes".equals(yesOrNoParam) ){

response.getWriter().write(

"<html><body>You said yes!</body></html>");

}

if("no".equals(yesOrNoParam) ){

response.getWriter().write(

"<html><body>You said no!</body></html>");

}

}

This service() method first reads the request parameter "param". Then it checks if the param is equal to the text "yes" or "no", and writes an HTML response back to the browser.

#### HttpServlet

The javax.servlet.http.HttpServlet class is a slightly more advanced base class than the GenericServlet shown in the example above.

The HttpServlet class reads the HTTP request, and determines if the request is an HTTP GET, POST, PUT, DELETE, HEAD etc. and calls one the corresponding method.

To respond to e.g. HTTP GET requests only, you will extend the HttpServlet class, and override the doGet() method only. Here is an example:

public class SimpleHttpServlet extends HttpServlet {

protected void doGet( HttpServletRequest request,

HttpServletResponse response)

throws ServletException, IOException {

response.getWriter().write("<html><body>GET response</body></html>");

}

}

The HttpServlet class has methods you can override for each HTTP method (GET, POST etc.). Here is a list of the methods you can override:

* doGet()
* doPost()
* doHead()
* doPut()
* doDelete()
* doOptions()
* doTrace()

Most often you just want to respond to either HTTP GET or POST requests, so you just override these two methods.

If you want to handle both GET and POST request from a given servlet, you can override both methods, and have one call the other. Here is how:

public class SimpleHttpServlet extends HttpServlet {

protected void doGet( HttpServletRequest request,

HttpServletResponse response)

throws ServletException, IOException {

**doPost(request, response);**

}

protected void doPost( HttpServletRequest request,

HttpServletResponse response)

throws ServletException, IOException {

response.getWriter().write("GET/POST response");

}

}

## HttpRequest and HttpResponse

The HttpServlet class’s request processing methods take two parameters.

1. javax.servlet.http.HttpRequest
2. javax.servlet.http.HttpResponse

For instance, here is the signature of the HttpServlet.doGet() method:

protected void doGet(

**HttpServletRequest request**, **HttpServletResponse response**)

throws ServletException, IOException {

}

### HttpRequest

The purpose of the HttpRequest object is to represent the HTTP request a browser sends to your web application. Thus, anything the browser may send, is accessible via the HttpRequest.

Recall that a HTTP request message consists of 3 parts: the request line, headers, body. For a HTTP request message, the followings are all a server needs:

- Parameters, i.e. usrname=Thang&gender=Male&address=Arizona, which are placed either in the first line of GET HTTP request message or in the form inside the body of POST HTTP request message.

- Headers, i.e. pair header\_name:header\_value

- Body

#### Parameters & getParameter(“…”)

Parameters are pairs name=value in the HTML form that browser sends to server.

For GET method, the form is the query string part in the URL request of the HTTP request message’s first line:

GET /example/message.html?**name=Jean&message=Yes** HTTP/1.1

For POST method, the form is placed in the body of the HTTP request message:

POST /example/message.html HTTP/1.1

Content-length: 24

Content-type: application/x-www-form-urlencoded

**name=Jean&message=Yes**

You can access these parameters from the HttpRequest object like this:

protected void doGet(

HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

**String param1 = request.getParameter("name");**

**String param2 = request.getParameter("message");**

}

#### Headers & getHeader(“…”)

Headers are name:value pairs in HTTP request message that tell server what browser software is being used, what file types the browser is capable of receiving etc. You can access the request headers from the HttpRequest object like this:

String contentLength = request.getHeader("Content-Length");

This example reads the Content-Length header sent by the browser. If it’s a POST request, the Content-Length is the number of bytes sent in the HTTP request body; if it’s a GET request, the return is null.

#### Body & getReader() or getInputStream()

To read the body of a HTTP response, the first step is to set up an input character/byte stream for the request’s message body. You use BufferedReader character stream if the body is textual, or InputStream byte stream if the body is binary

BufferedReader reader = request.getReader(); // character stream

String s = reader.readLine(...);

InputStream inputStream = request.getInputStream(); // byte stream

byte[] b;

inputStream.read(b);

Note that if the body is merely a form, a JSON, etc, then use specialized tools to parse. Manually reading the raw body is not the best way.

#### HttpSession

It is possible to obtain the session object from the HttpRequest object too.

The session object can hold information about a given user, between requests. So, if you set an object into the session object during one request, it will be available for you to read during any subsequent requests within the same session time scope.

Here is how you access the session object from the HttpRequest object:

HttpSession session = request.getSession();

I will not get into more detail about the session object here. It is covered in more detail in its own text.

### HttpResponse

The purpose of the HttpResponse object is to represent the HTTP response message your web application sends back to the browser, in response to the HTTP request the browser send to your web application.

Recall that a HTTP response message consists of 3 parts: status code, headers, body.

#### Status code & setStatus(…)

By default, a Servlet will send a status code “OK”. If you want to reply code “nnn” other than the default code, use:

response.setStatus(nnn);

HTTP status codes are numeric; for each code, there is a corresponding constant (begin with SC\_). For example, to send a status of 403 (“forbidden”), you can call:

res.setStatus(HttpServletResponse.SC\_FORBIDDEN);

Some status code constants:

400 SC\_BAD\_REQUEST

403 SC\_FORBIDDEN

404 SC\_NOT\_FOUND

405 SC\_METHOD\_NOT\_ALLOWED

500 SC\_INTERNAL\_SERVER\_ERROR

#### Headers & setHeader(“name”, “val”)

A header is a pair header\_name:header\_value. Headers must be set before any data is written to the response. To set a header, use setHeader(…,…) as below:

response.setHeader(“Header-Name”, “Header Value”);

Example: an important header is Content-Type, which tells the browser how to read the HTTP message’s body as a zip file or a text file, etc.

response.setHeader(“Content-Type”, “application/zip”);

#### Body & getWriter() or getOutputStream()

To compose the body of the HTTP response, the first step is to set up an output character/byte stream for the response’s message body. You use PrintWriter character stream if the body is HTML or JSON message etc, or OutputStream byte stream if the body is binary data like image, zip files.

PrintWriter writer = response.getWriter(); // character stream

Writer.print(…);

OutputStream outputStream = response.getOutputStream(); // byte stream

outputStream.write(…);

Example: HTTP response with a HTML page

response.setHeader(“Content-Type”, “text/html”);

PrintWriter writer = response.getWriter();

writer.println(“<html><body>Browser, take my response</body></html>”);

Example: HTTP response with binary data

response.setHeader(“Content-Length”, “31642”);

response.setHeader(“Content-Type”, “application/zip”);

OutputStream outputStream = response.getOutputStream();

outputStream.write(…);

#### Redirecting to a Different URL

You can redirect the browser to a different URL from your servlet. You cannot send any data back to the browser when redirecting. Here is how you redirect:

response.sendRedirect(“http://jenkov.com”);

## Call another servlet, jsp from inside a servlet

### RequestDispatcher

The RequestDispatcher class enables your servlet to “call” another servlet, JSP, HTTP from inside another servlet. The other servlet is called as if an HTTP request was sent to it by a browser.

You can obtain a RequestDispatcher from the HttpServletRequest object, like this:

protected void doPost(HttpServletRequest request,

HttpServletResponse response)

throws ServletException, IOException {

RequestDispatcher requestDispatcher =

request.getRequestDispatcher(“/anotherURL.simple”);

}

The above code obtains a RequestDispatcher that targets at (but not calls yet) whatever Servlet, JSP, resources that is mapped to the URL /anotherUrl.simple. The next step is to use this RequestDispatcher object to forward the request to the targeted URL (forward() method) or reply the request by including the response of the targeted URL (include() method).

Public void forward(ServletRequest request, ServletResponse response)

public void include(ServletRequest request, ServletResponse response)



As you see in the above figure, response of second servlet is sent to the client. Response of the first servlet is not displayed to the user.



As you can see in the above figure, response of second servlet is included in the response of the first servlet that is being sent to the client.

#### Example of RequestDispatcher

We use servlet to validate user login and a servlet to welcome the user once passing the validation:



* **index.html** file: for getting input from the user.
* **Login.java** file: a servlet class for processing the response. If password is servet, it will forward the request to the welcome servlet.
* **WelcomeServlet.java** file: a servlet class for displaying the welcome message.
* **web.xml** file: a deployment descriptor file that contains the information about the servlet.



**index.html**

<form action=”LoginServlet” method=”post”>

Name:<input type=”text” name=”sername”/><br/>

Password:<input type=”password” name=”userPass”/><br/>

<input type=”submit” value=”login”/>

</form>

**Login.java**

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

public class LoginServlet extends HttpServlet {

public void doPost(HttpServletRequest request, HttpServletResponse response)

        throws ServletException, IOException {

    response.setContentType(“text/html”);

    PrintWriter out = response.getWriter();

    String n = request.getParameter(“sername”);

    String p = request.getParameter(“userPass”);

    if(p.equals(“sername”)){

        RequestDispatcher rd = request.getRequestDispatcher(“servlet2”);

        rd.forward(request, response);

    }

    else{

        out.print(“Sorry UserName or Password is invalid!”);

        RequestDispatcher rd = request.getRequestDispatcher(“/index.html”);

        rd.include(request, response);

        }

    }

}

**WelcomeServlet.java**

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

public class WelcomeServlet extends HttpServlet {

    public void doPost(HttpServletRequest request, HttpServletResponse response)

        throws ServletException, IOException {

    response.setContentType(“text/html”);

    PrintWriter out = response.getWriter();

    String n=request.getParameter(“sername”);

    out.print(“Welcome “ + n);

    }

}

**web.xml**

<web-app>

 <servlet>

    <servlet-name>Login</servlet-name>

    <servlet-class>LoginServlet</servlet-class>

  </servlet>

  <servlet-mapping>

    <servlet-name>Login</servlet-name>

    <url-pattern>/login</url-pattern>

  </servlet-mapping>

  <servlet>

    <servlet-name>Welcome</servlet-name>

    <servlet-class>WelcomeServlet</servlet-class>

  </servlet>

  <servlet-mapping>

    <servlet-name>Welcome</servlet-name>

    <url-pattern>/welcome</url-pattern>

  </servlet-mapping>

  <welcome-file-list>

   <welcome-file>index.html</welcome-file>

  </welcome-file-list>

</web-app>

### sendRedirect

The **sendRedirect()** method of **HttpServletResponse**  can be used to redirect response to another resource (servlet, jsp or html file).

It accepts relative as well as absolute URL.

It works at client side because it uses the url bar of the browser to make another request. So, it can work inside and outside the server.

**Difference between the forward() method of RequestDispatcher and sendRedirect() method of HttpServletResponse:**

|  |  |
| --- | --- |
| forward() | sendRedirect() |
| The forward() method works at server side. | The sendRedirect() method works at client side. |
| It sends the same request and response objects to another servlet. | It always sends a new request. |
| It can work within the server only. | It can be used within and outside the server. |
| Example: request.getRequestDispacher(“servlet2”).forward(request,response); | Example:  response.sendRedirect(“servlet2”); |

Example: use sendRedirect method to send request to google server

index.html

<!DOCTYPE html>

<html>

<head>

<title>sendRedirect example</title>

</head>

<body>

<form action=”MySearcher”>

<input type=”text” name=”name”>

<input type=”submit” value=”Google Search”>

</form>

</body>

</html>

MySearcher.java

import java.io.IOException;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

public class MySearcher extends HttpServlet {

protected void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

String name=request.getParameter(“name”);

response.sendRedirect(“https://www.google.co.in/#q=”+name);

}

}

## ServletContext and ServletConfig

When you have some configuration information, for example database driver, that can be changed over time and used by many servlets, it’s better to put it in web.xml rather than repeat it in every servlet code so that when you want to change the configuration information, you change only web.xml, not every servlet. ServletContext and ServletConfig helps servlets to read/write configuration information in web.xml.

**ServletConfig** and **ServletContext**, both are objects created at the time of servlet initialization and used to provide some initial parameters or configuration information to the servlet. The difference is the scope: information shared by ServletConfig is for a specific servlet, while information shared by ServletContext is available for all servlets in the web application.

|  |  |
| --- | --- |
| **ServletConfig** | **ServletContext** |
| ServletConfig is specific for each servlet | ServletContext is for all servlets in the application |
| Parameters of servletConfig are present as name-value pair in **<init-param>** inside <servlet>. | Parameters of servletContext are present as name-value pair in **<context-param>** which is outside of <servlet> (but still inside <web-app>) |
| ServletConfig object is obtained by **getServletConfig()** method. | ServletContext object is obtained by **getServletContext()** method. |
| Each servlet has got its own ServletConfig object. | ServletContext object is only one and used by different servlets of the application. |
| Use ServletConfig when only one servlet needs information shared by it. | Use ServletContext when whole application needs information shared by it |
| Toget an initial parameter, use **getInitParameter(“parameter\_name”)** from Context/Config object. | |

Code snippet:

Servlet**Config** configure = getServlet**Config**();

String configParam = configure.getInitParameter(“configParam”);

Servlet**Context** context = getServlet**Context**();

String contextParam = context.getInitParameter(“contextParam”);

Note: there is another way to get the ServletContext object (but not ServletConfig) through HttpRequest object in doGet(), doPost():

protected void doGet(**HttpServletRequest request**, HttpServletResponse response)

throws ServletException, IOException {

**ServletContext context = request.getServletContext();**

// the above is equivalent to:   
 // Servlet**Context** context = getServlet**Context**();

}

Example: a job portal that has two pages: one for applicants (Applicant.java servlet), one for recruiters (Recruiter.java servlet). The both shares portal’s name “NewSite.tg”, but each has its own “email”: [forApplicant@xyz.com](mailto:forApplicant@xyz.com) for applicants, [forRecruiter@xyz.com](mailto:forRecruiter@xyz.com) for recruiters. So the email should be a ServletConfig, and hence put in <init-param> (inside <servlet>), and the portal name should be a ServletContext, and hence put in <context-param> (outside <servlet>).

Output: Deploy the app and open urls on localhost:

|  |  |
| --- | --- |
| /recruiter: | /applicant: |

**web.xml:** email is placed in <init-param> (inside <servlet>), website name is placed in <context-param> (outside <servlet>)

<web-app>

    <servlet>

        <servlet-name>recruiter</servlet-name>

        <servlet-class>Recruiter</servlet-class>

        <init-param>

            <param-name>Email</param-name>

            <param-value>forRecruiter@xyz.com</param-value>

        </init-param>

    </servlet>

    <servlet-mapping>

        <servlet-name>recruiter</servlet-name>

        <url-pattern>/recruiter</url-pattern>

    </servlet-mapping>

    <servlet>

        <servlet-name>applicant</servlet-name>

        <servlet-class>Applicant</servlet-class>

        <init-param>

            <param-name>Email</param-name>

            <param-value>forApplicant@xyz.com</param-value>

        </init-param>

    </servlet>

    <servlet-mapping>

        <servlet-name>applicant</servlet-name>

        <url-pattern>/applicant</url-pattern>

    </servlet-mapping>

    <context-param>

        <param-name>Website-name</param-name>

        <param-value>NewWebsite.tg</param-value>

    </context-param>

</web-app>

**recruiter.java and applicant.java are identical, except**

**Recruiter.java servlet**

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

public class Recruiter extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse response)

        throws ServletException, IOException {

        String email = getServlet**Config**().getInitParameter("Email");

        String website = getServlet**Context**().getInitParameter("Website-name");

PrintWriter out = response.getWriter();

        out.println("<center><h1>" + website

                    + "</h1></center><br><p>Contact us:"

                    + email);

    }

}

**Applicant.java servlet**

import java.io.\*;

import javax.servlet.ServletException;

import javax.servlet.http.\*;

public class Applicant extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse response)

        throws ServletException, IOException {

        String email = getServlet**Config**().getInitParameter("Email");

        String website = getServlet**Context**().getInitParameter("Website-name");

        PrintWriter out = response.getWriter();

        out.println("<center><h1>" + website

                    + "</h1></center><br><p>Contact us:"

                    + email);

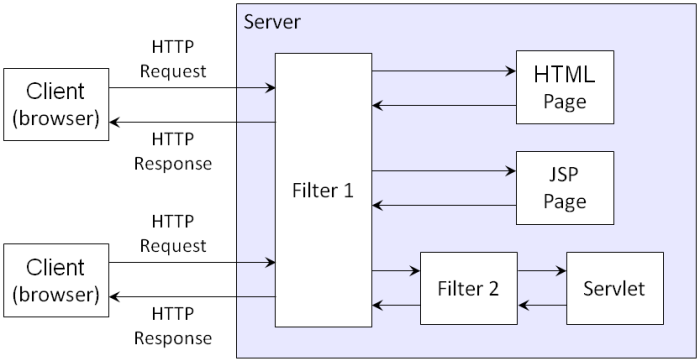
    }

}

## Servlet filter

### Filter is a pluggable component to intercept request/response

Servlet Filters are pluggable java components that we can use to intercept and process requests *before* they are sent to servlets and response *after* servlet code is finished and before container sends the response back to the client.



Some common tasks that we can do with servlet filters are:

* Logging request parameters to log files.
* Authentication and autherization of request for resources.
* Formatting of request body or header before sending it to servlet.
* Compressing the response data sent to the client.
* Alter response by adding some cookies, header information etc.

Note that **servlet filters are pluggable** and configured in deployment descriptor (web.xml) file. Servlets and filters both are unaware of each other and we can add or remove a servlet filter just by editing web.xml.

We can have multiple filters for a single resource and we can create a chain of filters for a single resource in web.xml. We can create a Servlet Filter by implementing javax.servlet.Filter interface.

### Filter API

Like servlet filter have its own API. The javax.servlet package contains the three interfaces of Filter API.

* Filter
* FilterChain
* FilterConfig

1) Filter interface

For creating any filter, you must implement the Filter interface. Filter interface provides the life cycle methods for a filter.

|  |  |
| --- | --- |
| Method | Description |
| public void init(FilterConfig config) | init() method is invoked only once. It is used to initialize the filter. |
| public void doFilter(HttpServletRequest request,HttpServletResponse response, FilterChain chain) | doFilter() method is invoked every time when user request to any resource, to which the filter is mapped. It is used to perform filtering tasks. |
| public void destroy() | This is invoked only once when filter is taken out of the service. |

2) FilterChain interface

The object of FilterChain is responsible to invoke the next filter or resource in the chain. This object is passed in the doFilter method of Filter interface. The FilterChain interface contains only one method:

**public void doFilter(HttpServletRequest request, HttpServletResponse response):** it passes the control to the next filter or resource.

### Mapping filter: <filter> tag in web.xml vs WebFilter annotation

**@WebFilter annotation**

javax.servlet.annotation.WebFilter was introduced in Servlet 3.0 and we can use this annotation to declare a servlet filter. We can use this annotation to define init parameters, filter name and description, servlets, url patterns and dispatcher types to apply the filter. If you make frequent changes to the filter configurations, its better to use web.xml because that will not require you to recompile the filter class.

**<filter> tag in Web.xml**

We can define filter same as servlet. Let's see the elements of filter and filter-mapping.

<web-app>

<filter>

<filter-name> ... </filter-name>

<filter-class> ... </filter-class>

</filter>

<filter-mapping>

<filter-name> ... </filter-name>

<url-pattern> ... </url-pattern>

</filter-mapping>

</web-app>

For mapping filters, we can use, either url-pattern or servlet-name. The url-pattern elements have an advantage over servlet-name element i.e. it can be applied on servlet, JSP or HTML.

### Example of a filter

In the following, when you call /servlet1, i.e. HelloServlet, MyFilter will intercept and print

**index.html**

<a href="servlet1">click here</a>

**MyFilter.java**

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.\*;

public class MyFilter implements Filter{

public void init(FilterConfig arg0) throws ServletException {}

public void doFilter(ServletRequest req, ServletResponse resp,

    FilterChain chain) throws IOException, ServletException {

    PrintWriter out=resp.getWriter();

    out.print("filter is invoked before");

    chain.doFilter(req, resp);//sends request to next resource

    out.print("filter is invoked after");

    }

    public void destroy() {}

}

**HelloServlet.java**

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.ServletException;

import javax.servlet.http.\*;

public class HelloServlet extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)

            throws ServletException, IOException {

        response.setContentType("text/html");

        PrintWriter out = response.getWriter();

        out.print("<br>welcome to servlet<br>");

    }

}

**web.xml**

For defining the filter, filter element of web-app must be defined just like servlet.

<web-app>

    <servlet>

        <servlet-name>s1</servlet-name>

        <servlet-class>HelloServlet</servlet-class>

    </servlet>

    <servlet-mapping>

        <servlet-name>s1</servlet-name>

        <url-pattern>/servlet1</url-pattern>

    </servlet-mapping>

    <filter>

        <filter-name>f1</filter-name>

        <filter-class>MyFilter</filter-class>

    </filter>

    <filter-mapping>

        <filter-name>f1</filter-name>

        <url-pattern>/servlet1</url-pattern>

    </filter-mapping>

</web-app>

## Session tracking

Http protocol is a stateless, i.e. each time a client makes a request to a server, server treats the request as a new request, so it doesn’t know if two requests come from a same client. More general, we say two requests are in a same **session** if they relate to each other in some sense, e.g. from a same browser, same users, in a same short period of time, etc. Stateless Http protocol cannot tell if two requests are in a same session, and hence we need session management technique to differentiate sessions for requests.

### General strategy to track sessions

Four techniques for tracking sessions, Cookies, Hidden Forms, URL Rewriting, HttpSession, share the following common theme:

* When server receives a request, it checks if the request contains some sessional information that is stored in cookies/hidden forms/session id in url/HttpSession object depending on the session tracking method it uses.
  + If the request does contain sessional information, the request must be from an OLD session. If browser doesn’t store the sessional information, server include that old sessional information to the response to browser.
  + If the request does NOT contain sessional information, the request must come from a NEW session. Server will create sessional information and attach it to server’s response to the browser.
* When client receives a response,
  + Cookie method (assuming cookies are enabled in browser): the response goes with cookies in headers, so browser stores the cookie and include the cookies in the next request.
  + Hidden Form or URL Rewriting method: the response that browser receives is usually a web page, in which all URL sthat lead to another request, are modified to include a session ID at the end. So, when users click on those links, a request with session ID is sent to the server.
  + HttpSession method (this method is equivalent to either Cookie or URL rewriting): the response goes with both cookies in headers and session ID in all URL in the response’s web page (as in URL rewriting method). If cookies are enabled, the cookie method is used; otherwise, the URL rewriting method is used.

UCLA: <http://web.cs.ucla.edu/classes/winter15/cs144/projects/java/session/>

### Cookies

A Cookie is just a Http header that server sends to browser so that browser stores it and include it in following requests to server to mark what requests belong to a same session. 

More specifically, when a browser sends a request to the server, the server checks if the request contains Cookies. If the request has Cookies, it means that the request comes from an old session; otherwise, the request comes from a new session and server set new Cookie headers in its response to browser. When a browser receives a response, if it contains Cookie headers, the browser stores the Cookies and include the cookie in its following requests to server.

#### HTTP header: “Set-Cookie” in response, “Cookie” in request

The [Set-Cookie](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie) HTTP response header sends cookies from the server to the user agent.

Example of a HTTP response with 2 cookies

HTTP/2.0 200 OK

Content-Type: text/html

Set-Cookie: yummy\_cookie=choco

Set-Cookie: tasty\_cookie=strawberry

[page content]

Then, with every subsequent request to the server, the browser sends back all previously stored cookies to the server using the [Cookie](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cookie) header.

GET /sample\_page.html HTTP/2.0

Host: www.example.org

Cookie: yummy\_cookie=choco; tasty\_cookie=strawberry

#### Cookie API

Cookies are headers in HTTP messages, and we have setHeader(), getHeader() to work with headers; however we will use a specialized class, named Cookie to work with cookies.

You create cookies and set properties for them by using Cookie class, and from HttpServletResponse, HttpServletRequest, you will read/add those Cookie objects.

**Cookie class**

**javax.servlet.http.Cookie** class provides the functionality of using cookies. It provides a lot of useful methods for cookies.

Constructor of Cookie class

|  |  |
| --- | --- |
| Constructor | Description |
| Cookie() | constructs a cookie. |
| Cookie(String name, String value) | constructs a cookie with a specified name and value. |

Useful Methods of Cookie class

|  |  |
| --- | --- |
| Method | Description |
| public void setMaxAge(int expiry) | Sets the maximum age of the cookie in seconds. |
| public String getName() | Returns the name of the cookie. The name cannot be changed after creation. |
| public String getValue() | Returns the value of the cookie. |
| public void setName(String name) | changes the name of the cookie. |
| public void setValue(String value) | changes the value of the cookie. |

**HttpServletResponse, HttpServletRequest with cookies**

For adding cookie or getting the value from the cookie, we need some methods provided by other interfaces. They are:

|  |  |
| --- | --- |
| public void addCookie(Cookie ck) | HttpServletResponse’s method to add cookie in response object. |
| public Cookie[] getCookies() | HttpServletRequest’s method to return all the cookies from the browser or null if the browser doesn’t send any cookie. |

**Code snippets**

**How to create Cookie?**

Cookie ck = new Cookie("user","sonoo jaiswal");//creating cookie object

response.addCookie(ck);//adding cookie in the response

**How to delete Cookie?**

Deleting cookies is used to logout users.

Cookie ck=**new** Cookie("user","");//deleting value of cookie

ck.setMaxAge(0);//changing the maximum age to 0 seconds

response.addCookie(ck);//adding cookie in the response

**How to get Cookies?**

Cookie ck[]=request.getCookies();

**for**(**int** i=0;i<ck.length;i++){

 out.print("<br>"+ck[i].getName()+" "+ck[i].getValue());//printing name and value of cookie

}

#### Example

Example 1: In this example, we are storing a username in a cookie object and accessing it in another servlet.



**index.html**

<form action="servlet1" method="post">

Name:<input type="text" name="userName"/><br/>

<input type="submit" value="go"/>

</form>

**FirstServlet.java**

**import** java.io.\*;

**import** javax.servlet.\*;

**import** javax.servlet.http.\*;

**public** **class** FirstServlet **extends** HttpServlet {

**public** **void** doPost(HttpServletRequest request, HttpServletResponse response){

**try**{

    response.setContentType("text/html");

    PrintWriter out = response.getWriter();

    String n = request.getParameter("userName");

    out.print("Welcome "+n);

    Cookie ck = **new** Cookie("uname", n);//creating cookie object

    response.addCookie(ck);//adding cookie in the response

    //creating submit button

    out.print("<form action='servlet2'>");

    out.print("<input type='submit' value='go'>");

    out.print("</form>");

    out.close();

        }**catch**(Exception e){System.out.println(e);}

  }

}

**SecondServlet.java**

**import** java.io.\*;

**import** javax.servlet.\*;

**import** javax.servlet.http.\*;

**public** **class** SecondServlet **extends** HttpServlet {

**public** **void** doPost(HttpServletRequest request, HttpServletResponse response){

**try**{

    response.setContentType("text/html");

    PrintWriter out = response.getWriter();

    Cookie ck[] = request.getCookies();

if(Cookie == null)

out.print("You have not logged in yet!");

else

     out.print("Hello " + ck[0].getValue());

    out.close();

         }**catch**(Exception e){System.out.println(e);}

    }

}

**web.xml**

<web-app>

    <servlet>

        <servlet-name>s1</servlet-name>

        <servlet-class>FirstServlet</servlet-class>

    </servlet>

    <servlet-mapping>

        <servlet-name>s1</servlet-name>

        <url-pattern>/servlet1</url-pattern>

    </servlet-mapping>

    <servlet>

        <servlet-name>s2</servlet-name>

        <servlet-class>SecondServlet</servlet-class>

    </servlet>

    <servlet-mapping>

        <servlet-name>s2</servlet-name>

        <url-pattern>/servlet2</url-pattern>

    </servlet-mapping>

</web-app>

Example 2: <https://www.javatpoint.com/servlet-login-and-logout-example-using-cookies>

### Hidden Form

Server can send browser a webpage that contains a hidden HTML form and buttons, links, events that submit a hidden form to the server. This is similar to that server sends browser a HTTP response with cookies, and then browser silently attaches the cookie to its subsequent requests to the server.

Using hidden forms for session tracking doesn’t require cookies to be enabled in the browser. However, it’s maintained at server and extra form submission is required on every page.

Example of using Hidden Form Field

In this example, we are storing the name of the user in a hidden textfield and getting that value from another servlet.



**index.html**

<form action="servlet1">

Name:<input type="text" name="userName"/><br/>

<input type="submit" value="go"/>

</form>

**FirstServlet.java**

**import** java.io.\*;

**import** javax.servlet.\*;

**import** javax.servlet.http.\*;

**public** **class** FirstServlet **extends** HttpServlet {

**public** **void** doGet(HttpServletRequest request, HttpServletResponse response){

**try**{

        response.setContentType("text/html");

        PrintWriter out = response.getWriter();

        String n=request.getParameter("userName");

        out.print("Welcome "+n);

        //creating form that have invisible textfield

        out.print("<form action='servlet2'>");

        out.print("<input type='hidden' name='uname' value='"+n+"'>");

        out.print("<input type='submit' value='go'>");

        out.print("</form>");

        out.close();

                }**catch**(Exception e){System.out.println(e);}

    }

}

**SecondServlet.java**

**import** java.io.\*;

**import** javax.servlet.\*;

**import** javax.servlet.http.\*;

**public** **class** SecondServlet **extends** HttpServlet {

**public** **void** doGet(HttpServletRequest request, HttpServletResponse response)

**try**{

        response.setContentType("text/html");

        PrintWriter out = response.getWriter();

        //Getting the value from the hidden field

        String n = request.getParameter("uname");

        out.print("Hello "+n);

        out.close();

                }**catch**(Exception e){System.out.println(e);}

    }

}

**web.xml**

<web-app>

    <servlet>

        <servlet-name>s1</servlet-name>

        <servlet-class>FirstServlet</servlet-class>

    </servlet>

    <servlet-mapping>

        <servlet-name>s1</servlet-name>

        <url-pattern>/servlet1</url-pattern>

    </servlet-mapping>

    <servlet>

        <servlet-name>s2</servlet-name>

        <servlet-class>SecondServlet</servlet-class>

    </servlet>

    <servlet-mapping>

        <servlet-name>s2</servlet-name>

        <url-pattern>/servlet2</url-pattern>

    </servlet-mapping>

</web-app>

### URL rewriting

Assume without session tracking, a server replies browser with a webpage/servlet S. Now if the server uses URL rewriting method, it will modify webpage/servlet S by appending a sessional information (i.e. a token/identifier/session ID, etc) to all URLs in S that leads to another request:

url?sessionID=XYZ&username=ABC

So, when users click on any link in S, the original request plus sessional information is submitted to the server. When server receives a request with sessional information, the server knows this request relates to a previous request, and the server will repeat appending this sessional information in URLs in subsequent responses to maintain the session.

HttpServletResponse has **String encodeURL(String url)** method to encode the specified URL by including the session ID in it, or, if encoding is not needed, returns the URL unchanged. For example, if the browser supports cookies, or session tracking is turned off, URL encoding is unnecessary.

**Example of using URL Rewriting**

In this example, we are maintaining the state of the user using link. For this purpose, we are appending the name of the user in the query string and getting the value from the query string in another page.



**index.html**

<form action="servlet1">

Name:<input type="text" name="userName"/><br/>

<input type="submit" value="go"/>

</form>

**FirstServlet.java**

**import** java.io.\*;

**import** javax.servlet.\*;

**import** javax.servlet.http.\*;

**public** **class** FirstServlet **extends** HttpServlet {

**public** **void** doGet(HttpServletRequest request, HttpServletResponse response){

**try**{

        response.setContentType("text/html");

        PrintWriter out = response.getWriter();

        String n = request.getParameter("userName");

        out.print("Welcome " + n);

        //appending the username in the query string

        out.print("<a href = 'servlet2?uname=" + n + "'>visit</a>");

        out.close();

        }**catch**(Exception e){System.out.println(e);}

    }

}

**SecondServlet.java**

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

public class SecondServlet extends HttpServlet {

public void doGet(HttpServletRequest request, HttpServletResponse response)

        try{

        response.setContentType("text/html");

        PrintWriter out = response.getWriter();

        //getting value from the query string

        String n = request.getParameter("uname");

        out.print("Hello " + n);

        out.close();

                }catch(Exception e){System.out.println(e);}

    }

}

**web.xml**

<web-app>

    <servlet>

        <servlet-name>s1</servlet-name>

        <servlet-class>FirstServlet</servlet-class>

    </servlet>

    <servlet-mapping>

        <servlet-name>s1</servlet-name>

        <url-pattern>/servlet1</url-pattern>

    </servlet-mapping>

    <servlet>

        <servlet-name>s2</servlet-name>

        <servlet-class>SecondServlet</servlet-class>

    </servlet>

    <servlet-mapping>

        <servlet-name>s2</servlet-name>

        <url-pattern>/servlet2</url-pattern>

    </servlet-mapping>

</web-app>

### HttpSession

A **HttpSession** object is an object that contains sessional information like user, the creation time, last access time, etc about a session between a client and a server. The HttpSession class gives you convenience for setting/retrieving sessional information, but how server and client maintain this HttpSession object for a session? The underlying mechanism here is Cookies or URL rewriting; HttpSession method is nothing new, except the convenience that it does Cookies or URL rewriting for web developers.

When receiving browser’s request, if the request doesn’t contain cookies or jsessionID in URL query string, server knows that the request is in a new session. Then the server creates a new session by doing both: adding a Cookie jsessionID:XYZ field and appending jsessionID=XYZ to the end of all URLs in the response’s webpage; the server does both because it doesn’t know if the browser enables cookies. If browser enables cookies, browser will add cookie jsessionID in its subsequent request signaling to the server to stop using URL rewriting; otherwise, the server uses URL rewriting.

If you want to add more sessional information beyond jsessionid, you can create a HttpSession object and associate it with a response. Also, if you want to access

#### HttpSession API

##### HttpSession interface

public String getId():Returns a string containing the unique identifier value.

public long getCreationTime():Returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.

public long getLastAccessedTime(): 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 void setAttribute(String name, Object value): Binds an object to this session, using the name specified.

public void invalidate(): Invalidates this session then unbinds any objects bound to it.

##### HttpServletRequest has getSession() method to get/create the HttpSession object

public HttpSession getSession(): Returns the current session associated with this request, or if the request does not have a session, creates one. So the return is never null.

public HttpSession getSession(boolean create): Similar to the above, but with create boolean to choose to create a session (true) or not (false) when the session doesn’t exist.

#### Code snippet

##### Looking up the HttpSession object associated with the current request.

This is done by calling the getSession method of HttpServletRequest. If this returns null, you can create a new session, but this is so commonly done that there is an option to automatically create a new session if there isn't one already. Just pass true to getSession. Thus, your first step usually looks like this:

HttpSession session = request.getSession(true);

##### Looking up Information Associated with a Session.

HttpSession objects live on the server; they're just automatically associated with the requester by a behind-the-scenes mechanism like cookies or URL-rewriting. These session objects have a built-in data structure that let you store and retrieve any number of keys and associated values. You can use getAttribute("key") to look up a previously stored value. The return type is Object, so you have to do a typecast to whatever more specific type of data was associated with that key in the session. The return value is null if there is no such attribute. You can use setAttribute("key", value) to store any value for the key, where the value can be any Java object.

Here's one representative example, assuming ShoppingCart is some class you've defined yourself that stores information on items being purchased.

HttpSession session = request.getSession(true);

ShoppingCart previousItems =

(ShoppingCart)session.getAttribute("previousItems");

if (previousItems != null) {

doSomethingWith(previousItems);

} else {

previousItems = new ShoppingCart(...);

doSomethingElseWith(previousItems);

}

In most cases, you have a specific attribute name in mind, and want to find the value (if any) already associated with it. However, you can also discover all the attribute names in a given session by calling getAttributeNames, which returns an Enumeration of all attribute names.

Although the data that was explicitly associated with a session is the part you care most about, there are some other pieces of information that are sometimes useful as well.

**getId.** This method returns the unique identifier generated for each session. It is sometimes used as the key name when there is only a single value associated with a session, or when logging information about previous sessions.

**isNew.** This returns true if the client (browser) has never seen the session, usually because it was just created rather than being referenced by an incoming client request. It returns false for preexisting sessions.

**getCreationTime.** This returns the time, in milliseconds since the epoch, at which the session was made. To get a value useful for printing out, pass the value to the Date constructor or the setTimeInMillis method of GregorianCalendar.

**getLastAccessedTime.** This returns the time, in milliseconds since the epoch, at which the session was last sent from the client.

**getMaxInactiveInterval.** This returns the amount of time, in seconds, that a session should go without access before being automatically invalidated. A negative value indicates that the session should never timeout.

##### Associating Information with a Session

As discussed in the previous section, you read information associated with a session by using getAttribute. To specify information, you use setAttribute, supplying a key and a value. Note that setAttribute replaces any previous values. Sometimes that's what you want (as with the referringPage entry in the example below), but other times you want to retrieve a previous value and augment it (as with the previousItems entry below). Here's an example:

HttpSession session = request.getSession(true);

session.setAttribute("referringPage", request.getHeader("Referer"));

ShoppingCart previousItems =

(ShoppingCart)session.getAttribute("previousItems");

if (previousItems == null) {

previousItems = new ShoppingCart(...);

}

String itemID = request.getParameter("itemID");

previousItems.addEntry(Catalog.getEntry(itemID));

// You still have to do setAttribute, not just modify the cart, since

// the cart may be new and thus not already stored in the session.

session.setAttribute("previousItems", previousItems);

#### Example: Showing Session Information

Here is a simple example that generates a Web page showing some information about the current session. The file session.zip contains the sample code that you can deploy and test on the Tomcat server on our VM.

import java.io.IOException;

import javax.servlet.Servlet;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import javax.servlet.http.HttpSession;

import java.io.PrintWriter;

import java.util.Date;

public class SessionServlet extends HttpServlet implements Servlet {

public SessionServlet() {}

public void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

HttpSession session = request.getSession(true);

response.setContentType("text/html");

PrintWriter out = response.getWriter();

String title = "Session Demo";

String heading;

Integer accessCount = new Integer(0);;

if (session.isNew()) {

heading = "Welcome, Newcomer";

} else {

heading = "Welcome Back";

Integer oldAccessCount = (Integer)session.getAttribute("accessCount");

if (oldAccessCount != null) {

accessCount = new Integer(oldAccessCount.intValue() + 1);

}

}

session.setAttribute("accessCount", accessCount);

out.println("<HTML><HEAD><TITLE>"+title+"</TITLE></HEAD>\n" +

"<BODY BGCOLOR=\"#FDF5E6\">\n" +

"<H1 ALIGN=\"CENTER\">" + heading + "</H1>\n" +

"<H2>Information on Your Session:</H2>\n" +

"<TABLE BORDER=1 ALIGN=CENTER>\n" +

"<TR BGCOLOR=\"#FFAD00\">\n" +

" <TH>Info Type<TH>Value\n" +

"<TR>\n" +

" <TD>ID\n" +

" <TD>" + session.getId() + "\n" +

"<TR>\n" +

" <TD>Creation Time\n" +

" <TD>" + new Date(session.getCreationTime()) + "\n" +

"<TR>\n" +

" <TD>Time of Last Access\n" +

" <TD>" + new Date(session.getLastAccessedTime()) + "\n" +

"<TR>\n" +

" <TD>Number of Previous Accesses\n" +

" <TD>" + accessCount + "\n" +

"</TABLE>\n" +

"</BODY></HTML>");

}

public void doPost(HttpServletRequest request,

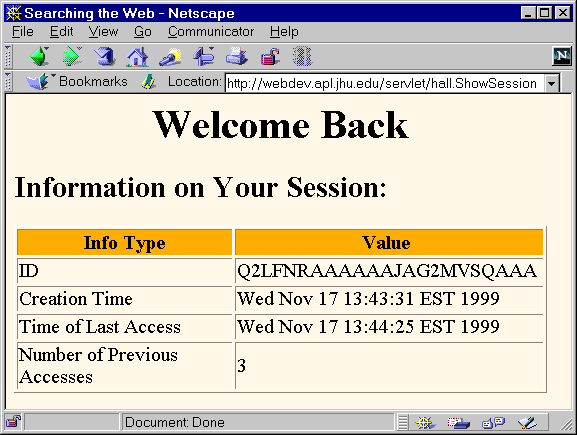
HttpServletResponse response)

throws ServletException, IOException {

doGet(request, response);

}

}

After building and deploying the downloaded sample code, if you visit http://localhost:1448/SessionDemo/session several times without quiiting your browser in between, you will see a page like this 

## **Type of Servlet Scopes**

There are mainly three types of servlet scopes: Request scope, Session scope, Context scope

### Request scope

Data stored in HttpServletRequest will be available until request object destroyed

Scope of such data called request scope

Request data can be accessed by single user within same request and before sending the response

### Session scope

Data stored in HttpSession object will be available until session persists

Scope of such data called session scope

Session data can be accessed by single user across multiple requests

User should belong to same session

### Context scope

Data stored in ServletContext will be available throughout the life of web application

Scope of such data is called context scope

Context data can be accessed by multiple users across multiple requests

## Deploy a servlet

### Steps to deploy a servlet

#### The general process

Step 0: Write a servlet

Step 1: Put the .class files of servlet to the server

When one deploys a program, only the compiled executable files are used so that one can run the program immediately without compiling it. Similarly, when one deploys a servlet, only the .class file (not source code) of servlets are placed on the web server.

Step 2: Mapping a servlet to a URL request

You do either annotation or web.xml, not the both.

For servlet 3.0 and newer, this can be done by use annotation @WebServlet("/FirstServlet1")

For servlet 2.5 and older, this is done through /web-inf/web.xml as following.

#### Create and Test a servlet in Eclipse

Create a Web Dynamic Project

Convert the project to Maven.

Add **servlet-api.jar** file by Maven to the project.

Create a servlet (4.0) and use annotation.

Run the servlet.

### Web descriptor /web-inf/web.xml

Web descriptor /web-inf/web.xml is an .xml file that map servlets with URL request. Everything is put in **<web-app>** and each mapping is described by two tags: **<servlet>** to tell name and the servlet’s Java class, and **<servlet-mapping>** to map servlet to an url.

<web-app>

<servlet>

<servlet-name>HelloWorld</servlet-name>

<servlet-class>HelloWorld</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>HelloWorld</servlet-name>

<url-pattern>/HelloWorld</url-pattern>

</servlet-mapping>

</web-app>

A **welcome file** is the file that is invoked automatically by the server, if you don't specify any file name.

By default server looks for the welcome file in following order:

1. welcome-file-list in web.xml
2. index.html
3. index.htm
4. index.jsp

If none of these files are found, server renders 404 error.

<web-app>

 ....

  <welcome-file-list>

    <welcome-file>home.html</welcome-file>

    <welcome-file>default.html</welcome-file>

  </welcome-file-list>

</web-app>

# Java Server Page

## What is Java Server Page (JSP)?

Java Server Pages (JSP) is a technology which is used to develop web pages by inserting Java code into the HTML pages by making special JSP tags. The 4 fundamental JSP tags are:

|  |  |  |  |
| --- | --- | --- | --- |
| JSP tags | Syntax | Function | Example |
| **Scriplets** | <% Java code %> | Java code | <% out.print("Hello"); %> |
| **Declaration** | <%! Declare variables %> | Declare variables | <%! int var=10; %> |
| **Expression** | <%= Expression %> | Evaluates & convert an expression to a string | <% num1 = num1+num2 %> |
| **Comments** | <%-- Comments --%> | Comment | <%-- Comments --%> |

JSP allows you to mix *static* HTML with *dynamically generated* HTML - in the way that the *business logic* and the *presentation* are well separated.

The advantages of JSP:

1. **Separation of static and dynamic contents**: JSP enables the separation of *static* contents from *dynamic* contents.  The dynamic contents are generated via programming logic and inserted into the *static template*. This greatly simplifies the creation and maintenance of web contents.
2. **Reuse of components and tag libraries:** The dynamic contents can be provided by reusable components such as JavaBean, Enterprise JavaBean (EJB) and tag libraries - you do not have to re-inventing the wheels.
3. **Java's power and portability**

**JSP vs Servlet**

Anything that can be done using JSPs can also be accomplished using Java servlets. However, it is important to note that servlets and JSPs are *complementary* technologies, NOT replacement of each other. Servlet can be viewed as "***HTML inside Java***", which is better for implementing business logic - as it is Java dominant. JSP, on the other hand, is "***Java inside HTML***", which is superior for creating presentation - as it is HTML dominant.

In a typical *Model-View-Control* (MVC) application, servlets are often used for the Controller (C), which involves complex programming logic. JSPs are often used for the View (V), which mainly deals with presentation. The Model (M) is usually implemented using JavaBean or EJB.

### First JSP Example - "Java inside HTML"

First of all, create a new web application (aka web context) called "hellojsp" in Tomcat, by creating a directory "hellojsp" under Tomcat's "webapps" directory (i.e., "<CATALINA\_HOME>\webapps\hellojsp" where <CATALINA\_HOME> denotes Tomcat's installed directory). The "<CATALINA\_HOME>\webapps\hellojsp" directory is known as context root for webapp (web context) "hellojsp".

Use a programming text editor to enter the following HTML/JSP codes and save as "first.jsp" under the context root "hellojsp". The file type of ".jsp" is mandatory for JSP script.

|  |  |
| --- | --- |
| 2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | <%@page language="java" contentType="text/html" pageEncoding="UTF-8" %>  <!DOCTYPE html>  <html>  <head>  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">  <title>First JSP</title>  </head>    <body>  <%  double num = Math.random();  if (num > 0.95) {  %>  <h2>You'll have a luck day!</h2><p>(<%= num %>)</p>  <%  } else {  %>  <h2>Well, life goes on ... </h2><p>(<%= num %>)</p>  <%  }  %>  <a href="<%= request.getRequestURI() %>"><h3>Try Again</h3></a>  </body>  </html> |

(Find it weird that after if (num > 0.95) is a <h2> HTML code? This makes sense because <h2> HTML code will be put in out.print(“<h2> …”) when the JSP is translated to Servlet. So, you have if() … and then print(“<h2> ….”))

To run this JSP, launch the Tomcat server. Check the console message to confirm that hellojsp has been deployed:

xxxx, xxxx xx:xx:xx xx org.apache.catalina.startup.HostConfig deployDirectory

INFO: Deploying web application directory **hellojsp**

......

Start a browser. Issue this URL (assume that Tomcat is running in port number 8080):

http://localhost:8080/hellojsp/first.jsp



Try "View Source" to check the response message received by the browser.

<!DOCTYPE HTML>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

<title>First JSP</title>

</head>

<body>

<h2>Well, life goes on ... </h2><p>(0.16450910536318764)</p>

<a href="/hellojsp/first.jsp"><h3>Try Again</h3></a>

</body>

</html>

Observe that the response message does not include the JSP source codes, but merely the output of the JSP script. This clearly illustrates that JSP (like servlets) are server-side programs, that are executed in the server. The output is then sent to the client (browser) as the response message.

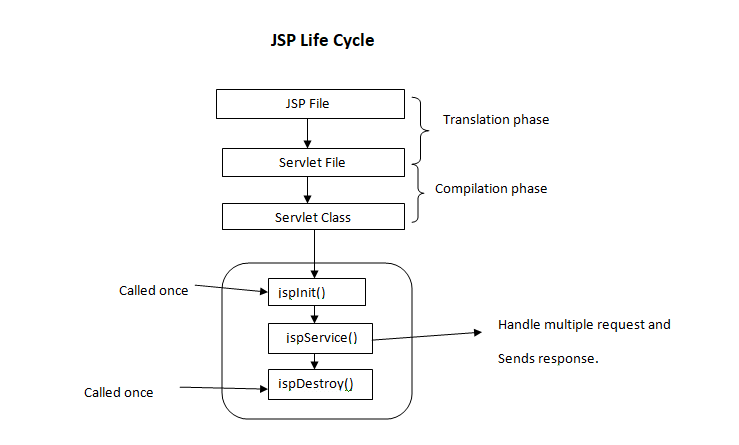
**Explanations**

* A JSP script is a regular HTML page containing additional Java codes.
* In line 1, we declare this is a JSP script via a JSP directive. The <!DOCTYPE> tag identifies that the response is an HTML document. The <html>...</html> tags enclose the HTML document, which consists of two sections: head and body, enclosed by <head>...</head> and <body>...</body>, respectively.
* The JSP's Java codes are enclosed within special tags in the form of <% ... %> (similar to ASP). We shall explain these codes later.
* "JSP is Java inside HTML" - Java codes are embedded inside an HTML page. On the other hand, "Servlet is HTML inside Java" - HTML page is produced using Java's out.println() in a Java program.

## Life cycle of JSP

### Life cycle of JSP

A Java Server Page life cycle is defined as the process started with its creation which later translated to a servlet and afterward servlet lifecycle comes into play. This is how the process goes on until its destruction.



Following steps are involved in JSP life cycle:

**Translation of JSP page to Servlet**: This is the first step of JSP life cycle. This translation phase deals with Syntactic correctness of JSP. Here test.jsp file is transllated to test.java.  
  
**Compilation of JSP page**: Here the generated java servlet file (test.java) is compiled to a class file (test.class).  
  
**Classloading**: Servlet class which has been loaded from JSP source is now loaded into container.  
  
**Instantiation**: Here instance of the class is generated. The container manages one or more instance by providing response to requests.  
  
**Initialization**: jspInit() method is called only once during the life cycle immediately after the generation of Servlet instance from JSP.  
  
**Request processing**: \_jspService() method is used to serve the raised requests by JSP. It takes request and response object as parameters.This method cannot be overridden.  
  
**JSP Cleanup**: In order to remove the JSP from use by the container or to destroy method for servlets jspDestroy()method is used. This method is called once, if you need to perform any cleanup task like closing open files, releasing database connections jspDestroy() can be overridden.

We can override jspInit(), jspDestroy() but we can’t override \_jspService() method.

### How a JSP is translated into a Servlet

#### Recall what a servlet looks like

A typical Java servlet (as shown below) contains three groups of methods: init(), destroy(), and one or more service() methods such as doGet() and doPost(). init() runs (once) when the servlet is loaded into the server. destroy() runs (once) when the servlet is unloaded. service() runs once per HTTP request (e.g., doGet() runs once per GET request, doPost() run once per POST request). The service() methods takes two arguments: request and response, encapsulating HTTP request and response messages respectively. A PrintWriter object called out is used for writing out the response message to the client over the network.

public class **MyServlet** extends HttpServlet {

// Instance variables or methods, e.g., database Connection, Statement, helper methods

......

......

// init() runs only once when the servlet is loaded into the server

public void **init**() { ...... }

// doGet() runs once per HTTP GET request

// It takes two arguments, representing the request and response messages

public void **doGet(HttpServletRequest request, HttpServletResponse response)**

throws IOException, ServletException {

// Set the MIME type for the response message

response.setContentType("text/html");

// Create a Writer to write the response message to the client over the network

**PrintWriter out = response.getWriter();**

// The programming logic to produce a HTML page

out.println("<html>");

out.println( ...... );

out.println("</html>");

}

// doPost() runs once per HTTP Post request

public void **doPost(HttpServletRequest request, HttpServletResponse response)**

throws IOException, ServletException {

......

}

// destroy() runs only once when the servlet is unloaded from the server.

public void **destroy**() { ...... }

}

#### Translate JSP code to Servlet code

When a JSP page is first requested, Tomcat translates the JSP into a servlet, compiles the servlet, load, and execute the servlet. The best way to understand JSP is to check out the generated servlet and study the JSP-to-Servlet translation. The generated servlet for "first.jsp" is kept under Tomcat's "work" directory ("<CATALINA\_HOME>\work\Catalina\localhost\hellojsp\..."). The relevant part of the generated servlet is extracted below:

public final class **first\_jsp** extends org.apache.jasper.runtime.HttpJspBase

implements org.apache.jasper.runtime.JspSourceDependent {

public void **\_jspInit()** { ...... }

public void **\_jspDestroy()** { ...... }

public void **\_jspService**(HttpServletRequest **request**, HttpServletResponse **response**)

throws java.io.IOException, ServletException {

// JSP pre-defined variables (in service() method)

**PageContext pageContext** = null;

**HttpSession session** = null;

**ServletContext application** = null;

**ServletConfig config** = null;

**JspWriter out** = null;

**Object page** = this;

JspWriter \_jspx\_out = null;

PageContext \_jspx\_page\_context = null;

try {

**response.setContentType("text/html");**

pageContext = \_jspxFactory.getPageContext(this, request, response,

null, true, 8192, true);

\_jspx\_page\_context = pageContext;

application = pageContext.getServletContext();

config = pageContext.getServletConfig();

session = pageContext.getSession();

out = pageContext.getOut();

\_jspx\_out = out;

**out.write("<!DOCTYPE HTML>\r\n");**

**out.write("<html>\r\n");**

**out.write("<head>\r\n");**

**out.write(" <meta http-equiv=\"Content-Type=\" content=\"text/html; charset=UTF-8\">");**

**out.write(" <title>First JSP</title>\r\n");**

**out.write("</head>\r\n");**

**out.write("\r\n");**

**out.write("<body>\r\n");**

**out.write(" ");**

**double num = Math.random();**

**if (num > 0.95) {**

**out.write("\r\n");**

**out.write(" <h2>You will have a luck day!</h2><p>(");**

**out.print( num );**

**out.write(")</p>\r\n");**

**out.write(" ");**

**} else {**

**out.write("\r\n");**

**out.write(" <h2>Well, life goes on ... </h2><p>(");**

**out.print( num );**

**out.write(")</p>\r\n");**

**out.write(" ");**

**}**

**out.write("\r\n");**

**out.write(" <a href=\"");**

**out.print( request.getRequestURI() );**

**out.write("\"><h3>Try Again</h3></a>\r\n");**

**out.write("</body>\r\n");**

**out.write("</html>\r\n");**

} catch (Throwable t) {

if (!(t instanceof SkipPageException)) {

out = \_jspx\_out;

if (out != null && out.getBufferSize() != 0)

try { out.clearBuffer(); } catch (java.io.IOException e) {}

if (\_jspx\_page\_context != null) \_jspx\_page\_context.handlePageException(t);

}

} finally {

\_jspxFactory.releasePageContext(\_jspx\_page\_context);

}

}

}

Clearly, the translation is carried out as follows:

* The methods \_jspInit(), \_jspDestroy() and \_jspService() corresponds to init(), destroy() and service() (doGet(), doPost()) of a regular servlet. Similar to servlet's service(), \_jspService() takes two parameters, request and response, encapsulating the HTTP request and response messages. A JspWriter called out, corresponding to servlet's PrintWriter, is allocated to write the response message over the network to the client.
* The HTML statements in the JSP script are written out as part of the response via out.write(...), as "it is" without modification.
* The Java codes in the JSP script are translated according to their respective types:
  + JSP Scriptlet <% .... >: used to include Java statements, or part of Java statement. The Java statements are placed inside the \_jspService() method of the translated servlet as "it is". Scriptlets form the program logic.
  + JSP Expression <%= .... >: used to evaluate a single Java expression to obtain a value. The Java expression is placed inside a out.print(...). In other words, the Java expression will be evaluated, and the result of the evaluation written out as part of the response message.

Subsequent accesses to the this JSP page will be much faster, because they will be re-directed to the translated and compiled servlet directly (no JSP-to-servlet translation and servlet compilation needed again), unless the JSP page has been modified.

## JSP Pre-Defined Variables

JSP pre-defined 9 variables, that are available to the script writer. They are:

|  |  |
| --- | --- |
| Object | Type |
| out | JspWriter |
| request | HttpServletRequest |
| response | HttpServletResponse |
| config | ServletConfig |
| application | ServletContext |
| session | HttpSession |
| page | Object |
| pageContext | PageContext |
| exception | Throwable |

**request**: A HttpServletRequest object, keeping track of the HTTP request message. It is often used to retrieve the query parameters in the request message. For example,

// Single-value parameter

String paramValue = request.getParameter("paramName");

if (paramValue != null && paramValue.legnth() != 0) { // param exists and not empty string

// process the parameter

......

}

// Multiple-value parameter

String[] paramValues = request.getParameterValues("paramName");

if (paramValues != null && paramValues.length > 0) { // param exists and at least one item

// Process parameters

for (String paramValue : paramValues) {

......

}

}

**response**: A HttpServletResponse object, keeping track of the HTTP response message.

**out**: A Writer (JspWriter) object used to write response message to the client over the network socket, via methods print() or println().

**session**: A HttpSession object, keeping track of the current client session (from the moment the user accesses the first page, until he/she closes the browser or session timeout). You can use session's attributes to pass information between pages within this session, via methods getAttribute("name") and setAttribute("name", object). For example,

// Allocate a shopping cart

List<String> shoppingCart = new ArrayList<>();

.....

// Place the shopping cart inside the session

session.setAttribute("cart", shoppingCart);

.....

// Any page can retrieve the shopping cart

List<String> theCart = (List<String>)session.getAttribute("cart");

if (theCart != null) { // cart exists?

for (String item : theCart) { // process the cart items

......

}

}

**application**: A ServletContext object retrieved via getServletContext(), which maintains information about this web context (web application). You can use the application's attributes to pass information between JSP pages and servlets, via methods getAttribute("name") and setAttribute("name", object).

**config**: A ServletConfig object, obtained via getServletConfig(). It could be used to retrieve the servlet initialization parameters provided in "WEB-INF\web.xml", via method getInitParameter("paramName").

**page**: Can be used to access the elements of this page.

The translated servlet shows these seven variables are allocated and initialized as follows (with re-arrangements):

public void \_jspService(**HttpServletRequest request**, **HttpServletResponse response**)

javax.servlet.jsp.PageContext pageContext

= \_jspxFactory.getPageContext(this, request, response, null, true, 8192, true);

/\* getPageContext(javax.servlet.Servlet servlet,

javax.servlet.ServletRequest request,

javax.servlet.ServletResponse response,

java.lang.String errorPageURL,

boolean needsSession,

int bufferSize,

boolean autoFlushBuffer) \*/

javax.servlet.http.**HttpSession session** = pageContext.getSession();

javax.servlet.**ServletContext application** = pageContext.getServletContext();

javax.servlet.**ServletConfig config** = pageContext.getServletConfig();

javax.servlet.jsp.**JspWriter out** = pageContext.getOut();

java.lang.**Object page = this**;

......

## JSP Scripting Elements

JSP scripting element are enclosed within <% ...... %>, similar to other server-side scripts such as ASP and PHP. To print "<%", use escape sequence "<\%".

### JSP Comment <%-- comments --%>

JSP comments <%-- JSP comments --%> are ignored by the JSP engine. For example,

<%-- anything but a closing tag here will be ignored -->

Note that HTML comment is <!-- html comments -->. JSP expression within the HTML comment will be evaluated. For example,

<!-- HTML comments here <%= Math.random() %> more comments -->

### JSP Expression <%= JavaExpression %>

A JSP expression is used to insert the resultant value of a single Java expression into the response message. The Java expression will be placed inside a out.print(...) method. Hence, the expression will be evaluated and resultant value printed out as part of the response message. Any valid Java expression can be used.

There is no semi-colon at the end of the expression.

For examples:

<%= Math.sqrt(5) %>

<%= item[10] %>

<p>The current data and time is: <%= new java.util.Date() %></p>

The above JSP expressions will be converted to:

out.print( Math.sqrt(5) );

out.print( item[10] );

out.write("<p>Current time is: ");

out.print( new java.util.Date() );

out.write("</p>");

You can use the pre-defined variables in the expressions. For examples:

<p>You have choose author <%= request.getParameter("author") %></p>

<%= request.getRequestURI() %>

You can also use the XML-compatible syntax of <jsp:expression>Java Expression</jsp:expression>.

### JSP Scriptlet <% Java Statements %>

JSP scriptlets allow you to implement more complex programming logic. You can use scriptlets to insert any valid Java statements into the \_jspService() method of the translated servlet. The Java codes must be syntactically correct, with Java statements terminated by a semi-colon.

For example:

<%

String author = request.getParameter("author");

if (author != null && !author.equals(""))) {

%>

<p>You have choose author <%= author %></p>

<%

}

%>

In the translated servlet, the above will be inserted into the service() method as follows:

public void **\_jspService**(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

......

String author = request.getParameter("author");

if (author != null && !author.equals(""))) {

out.write("<p>You have choose author ");

out.print(author);

out.write("</p>");

}

}

The source codes of scriptlets are only available in the server, and not sent to the client. That is, scriptlets are safe and secure!

You can also use the XML-compatible syntax of <jsp:scriptlet>Java Statements</jsp:scriptlet>.

### JSP Declaration <%! Java Statements %>

JSP declarations can be used to define variables and methods for the class. **Unlike scriptlets that are inserted inside the \_jspService() method, the declaration codes are inserted inside the class, at the same level as \_jspService(), as variables or methods of the class**.

For example,

<%! private int count; %>

<%! public int incrementCount() { ++count; } %>

will be translated to:

public final class first\_jsp extends org.apache.jasper.runtime.HttpJspBase

implements org.apache.jasper.runtime.JspSourceDependent {

**private int count;**

**public int incrementCount() { ++count; }**

public void **\_jspInit**() { ...... }

public void **\_jspDestroy**() { ...... }

public void **\_jspService**(HttpServletRequest **request**, HttpServletResponse **response**)

throws java.io.IOException, ServletException { ...... }

}

You can also use the XML-compatible syntax of <jsp:declaration>Java Statements</jsp:declaration>.

### JSP Directive <%@ ... %>

JSP directives provide instructions to the JSP engine to aid in the translation. The syntax of the JSP directive is:

<%@ directiveName attribute1="attribute1value" attribute2="attribute2value" ... %>

The directives include: page, include, taglib.

### JSP Page Directive <%@page ... %>

Typically, a JSP file starts with the page directive:

<%@page language="java" contentType="text/html" %>

The "page import" directive lets you import classes. The syntax is:

<%@page import="packageName.\*" %>

<%@page import="packageName.className" %>

<%@page import="packageName.className1, packageName.className2, ..." %>

This directive generates proper import statements at the top of the servlet class. User-defined classes must be stored in "*webContextRoot*\WEB-INF\classes", and kept in a proper package (default package not allowed).

For example,

<%-- import package java.sql.\* --%>

<%@page import="java.sql.\*" %>

<%-- import a user-defined class --%>

<%@page import="mypkg.myClass1, mypkg.myClass2" %>

The "page" directives are also used to set the MIME type, character set of the response message. These information will be written to the response message's header. For example,

<%-- Set the response message's MIME type, character set --%>

<%@page contentType="image/gif" %>

<%@page contentType="text/html" charset="UTF-8" %>

<%@page pageEncoding="UTF-8" %>

<%-- Set an information message for getServletInfo() method --%>

<%@page info="Hello world example" %>

The "page session" directive allows you to designate whether this page belongs to the session. The default is "true". Setting to "false" could reduce the server's load, if session tracking is not needed in your application.

<%@page session="true|false" @>

Other "page" directives include: errorPage, isErrorPage, buffer, isThreadSafe, isELIgnored, etc.

### JSP Include Directive <%@include ... %>

The "include" directive lets you insert the unprocessed content of an external file. You can include any JSP files or static HTML files. You can use include directive to include navigation bar, copyright statement, logo, etc. on every JSP pages. The syntax is:

<%@include file="url" %>

<%@include page="url" &>

If another JSP page is included using "include file", the JSP engine will not check for update of the included JSP file during execution.

For example:

<%@include file="header.jsp" %>

....

<%@include file="footer.jsp" %>

### JSP Taglib Directive <%@taglib ... %>

JSP allows you to create your own libraries of JSP tags. You can use the taglib directive to tell Tomcat what libraries to load and where they are. For example,

<%@taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>

<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

Taglib will be elaborated later.

## JSP Standard Actions

The purpose of JSP actions is to specify activities to be performed when a page is processed. In contrast, JSP scripting elements such as directives (<%@ ... %>) are processed when translating the page, which produces static contents.

A JSP action may contain sub-action. The syntax is as follows.

<jsp:action-name attribute-list>

<jsp:subaction-name subaction-attribute-list />

</jsp:action-name>

There are eight JSP standard actions (forward, include, useBean, getProperty, setProperty, plugin, text, element) and five sub-actions (attribute, body, fallback, param, and params).

### <jsp:forward>

To terminate execution of the current page and forward the request to another page. For example,

<jsp:forward page="nextPage.jsp">

<jsp:param name="parmName" value="parmValue"/>

</jsp:forward>

The destination page can access the new parameters via request.getParameter(paramName). Tomcat clears the output buffer upon executing a forward action.

<jsp:forward> is translated as follows:

<jsp:forward page="index.jsp" />

// is translated to

\_jspx\_page\_context.forward("index.jsp");

// Re-direct, or forward the current ServletRequest and ServletResponse to

// another active component in the application.

### <jsp:include>

To execute another page and append its output to the current page. That is, it places the generated HTML code into the current JSP page. This is different from the JSP include directive <%@ include ... %>, which insert unprocessed content.

<jsp:include page="headers.jsp"/>

<jsp:include> is translated as follows:

<jsp:include page="index.jsp"/>

// is translated to

org.apache.jasper.runtime.JspRuntimeLibrary.include(request, response, "index.jsp", out, false);

// Invoke the include method during execution of the servlet.

// Cause the target page to be processed, and output written to the response message.

// The last boolean argument specifies whether JspWriter is to be flushed before the include.

Other actions will be presented in the next section.

## Using JavaBeans in JSP (JSP Actions useBean, getProperty, setProperty)

### Recall JavaBeans

A JavaBean is a Java class which conforms to the following rules:

* It has a no-arg constructor.
* It does not have public variables.
* It is defined in a named package. It can not be kept in the default no-name package.
* For a private variable xxx, there is a public getter getXxx() (or isXxx() for boolean) and a public setter setXxx().
* For an event xxxEvent, there is an interface xxxListener, and methods addXxxListener() and removeXxxListener().
* It implements Serializable interface, so that its state can be stored and retrieved to and from external storage, for persistent.

### JSP Actions for JavaBean

#### <jsp:useBean>

Create a new bean instance. The bean class must be kept in "<WebContextRoot>\WEB-INF\classes" within a proper package directory structure.

<jsp:useBean id="beanName" class="packageName.ClassName" scope="page|request|session|application" />

// Equivalent to JSP Declaration:

// <%! PackageName.className beanName = new PackageName.className(); %>

// setAttribute according to the scope.

The attribute scope specifies the scope of this bean:

* page: default, stored in PageContext, available to this page only.
* request: stored in ServletRequest, can be forwarded to another JSP page or servlet.
* session: stored in HttpSession, available to all pages and servlets within this session.
* application: stored in ServletContext, shared by all servlets and JSP pages in this web context (application).

#### <jsp:setProperty>

Set the value of a property (variable) of a bean, by invoking its setter.

<jsp:setProperty name="beanName" property="propertyName" value="propertyValue" />

// Equivalent to Scriptlet: <% beanName.setPropertyName(propertyValue); %>

#### <jsp:getProperty>

Get the value of a property (variable) of a bean, by invoking its getter.

<jsp:getProperty name="beanName" property="propertyName" />

// Equivalent to Expression: <%= beanName.getPropertyName() %>

### JSP's JavaBean and HTML form

Bean is frequently used in a HTML form, to capture the value of the selected request parameters (such as username), and pass it over to the other processing pages or server-side programs, depending on its scope.

Instead of setting each of the bean's properties using many <jsp:setProperty>, you can use wildcard '\*' in the property attribute to set the properties of a bean to the request parameters with the matching names. Simple type conversion from String will be carried out. For example,

<jsp:useBean name="mybean" class="mypkg.myClass" scope="session' />

// Instantiate a bean

<jsp:setProperty name="mybean" property="\*" />

// Binds the properties with matching request parameter names

// Same as mybean.setXxx(request.getParameter("xxx")) for all the properties in the bean

Suppose the request contains a parameter username=xxx and mybean has a property username, then setUsername(xxx) will be invoked.

### A JSP with JavaBean Example

Let's create a JavaBean called "UserBean.java", which contains a variable (property) called username, with getter and setter as follows:

package mypkg;

public class UserBean {

private String username;

public UserBean() {

username = "";

}

public String getUsername() {

return username;

}

public void setUsername(String username) {

this.username = username;

}

}

Next, create a input form to prompt user for his/her name called index.jsp:

<%@page contentType="text/html" pageEncoding="UTF-8"%>

<!DOCTYPE HTML >

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

<title>Input Form</title>

</head>

<body>

<h1>Hello,</h1>

<form action="response.jsp">

Enter your name: <input type="text" name="username" />

<input type="submit" />

</form>

</body>

</html>

The Input Form submits the request to "response.jsp", as below:

<%@page contentType="text/html" pageEncoding="UTF-8"%>

<!DOCTYPE HTML >

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

<title>Response Page</title>

</head>

<body>

<jsp:useBean id="user" scope="session" class="mypkg.UserBean" />

<jsp:setProperty name="user" property="\*" />

<h1>Hello, <jsp:getProperty name="user" property="username" />!</h1>

</body>

</html>

* The <jsp:useBean> directive construct an instance of mypkg.UserBean called "user". This bean has scope of session, which is kept in the HTTPSession, and available to all JSP pages and servlets within this session.
* The <jsp:setProperty> sets the value of username in mykpg.UserBean to the value of the request parameter "username" (i.e., user.setUsername(request.getParameter("username")).
* The <jsp:getProperty> retrieves the username of mypkg.UserBean (i.e., user.getUserName()).

**Behind the Scene**

For the above example, the <jsp:useBean>, <jsp:setProperty> and <jsp:getProperty> for "session" scope is translated as follows:

<jsp:useBean id="user" scope="session" class="mypkg.UserBean" />

<jsp:setProperty name="user" property="username" />

<h1>Hello, <jsp:getProperty name="user" property="username" />!</h1>

// <jsp:useBean>

mypkg.UserBean user = null;

synchronized (**session**) { // prevent concurrent update

// Check if bean already exists in the HTTPSession.

// If so, retrieve from the session

user = (mypkg.UserBean) \_jspx\_page\_context.**getAttribute**(

"user", javax.servlet.jsp.PageContext.**SESSION\_SCOPE**);

if (user == null) {

// Otherwise, allocate an instance, and place it in the session

**user = new mypkg.UserBean();**

\_jspx\_page\_context.**setAttribute**(

"user", user, javax.servlet.jsp.PageContext.**SESSION\_SCOPE**);

}

}

// <jsp:setProperty>

org.apache.jasper.runtime.JspRuntimeLibrary.introspecthelper(

\_jspx\_page\_context.**findAttribute**("user"), "username",

request.getParameter("username"), request, "username", false);

// <jsp:getProperty>

out.write(org.apache.jasper.runtime.JspRuntimeLibrary.toString(

(((mypkg.UserBean)\_jspx\_page\_context.**findAttribute**("user")).**getUsername**())));

<jsp:setProperty> with wildcard '\*':

<jsp:setProperty name="user" **property="\*"** />

org.apache.jasper.runtime.JspRuntimeLibrary.introspect(

\_jspx\_page\_context.findAttribute("user"), request);

<jsp:useBean> with "application" scope:

<jsp:useBean id="user" **scope="application"** class="mypkg.UserBean" />

mypkg.UserBean user = null;

synchronized (**application**) {

user = (mypkg.UserBean) \_jspx\_page\_context.getAttribute(

"user", javax.servlet.jsp.PageContext.**APPLICATION\_SCOPE**);

if (user == null) {

   user = new mypkg.UserBean();

\_jspx\_page\_context.setAttribute(

"user", user, javax.servlet.jsp.PageContext.**APPLICATION\_SCOPE**);

}

}

# Spring

<https://www.tutorialspoint.com/spring/spring_architecture.htm>

## Random Notes

Spring is a Inversion of Control container

Dependency injection is implementation of IoC

2 types of IoC containers:

Bean Factory and Application Context

Spring is a framework of framework

## Temp notes

### Inversion of Control and Dependency Injection

Inversion of Control: as Martin Fowler <https://stackoverflow.com/questions/3058/what-is-inversion-of-control>

Exp1: Inversion of Control is what you get when your program callbacks, e.g. like a gui program.

For example, in an old school menu, you might have:

print "enter your name"

read name

print "enter your address"

read address

etc...

store in database

thereby controlling the flow of user interaction.

In a GUI program or some such, instead we say:

when the user types in field a, store it in NAME

when the user types in field b, store it in ADDRESS

when the user clicks the save button, call StoreInDatabase

So now control is inverted... instead of the computer accepting user input in a fixed order, the user controls the order in which the data is entered, and when the data is saved in the database.

Basically, **anything** with an event loop, callbacks, or execute triggers falls into this category.

Exp 2: Two types of dependency injections:

|  |  |
| --- | --- |
| 1 | [Constructor-based dependency injection](https://www.tutorialspoint.com/spring/constructor_based_dependency_injection.htm)  Constructor-based DI is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on the other class. |
| 2 | [Setter-based dependency injection](https://www.tutorialspoint.com/spring/setter_based_dependency_injection.htm)  Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean. |

# Tutorial for servlet - great

<https://www.javatpoint.com/servletconfig>

<https://beginnersbook.com/2013/05/http-session/>

# Data Access Object

<http://tutorials.jenkov.com/java-persistence/dao-design-problems.html>

# Notes about Eclipse, Maven

## Java.\* vs javax.\*

Java.\* contains **built-in** packages of Java so when you use it, you don’t need to add a jar file.

Javax.\* contains **external** packages so when you use it, you need to add jar file.

Over time, javax.\* are moved to java.\*

All the jar files of javax.\* are developed by Java Community Process. However, people often get them through Maven.

## Directory structure of Eclipse, Maven project for a Java web application

Servlet requires an external .JAR library for it.

# JPA and Hibernate

Annotation with jointable, one-to-one

# Tutorial Java EE

https://docs.oracle.com/javaee/6/tutorial/doc/

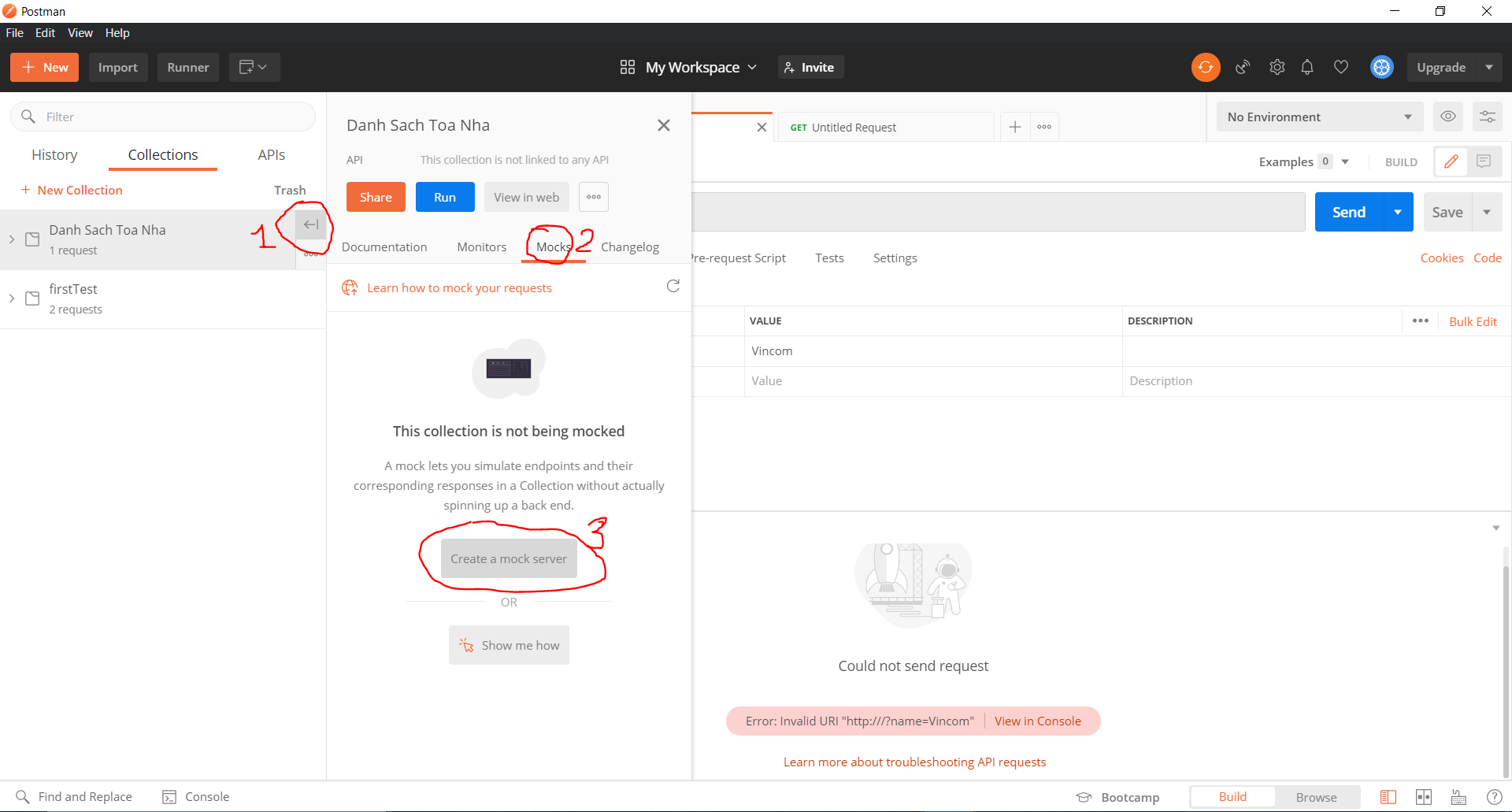
# POSTMAN - Create a mock server, mock response

To create a mock server and then some mock response from the server, you need to:

* Create a mock server
* Add a request that has a link to the server

## Create a mock server

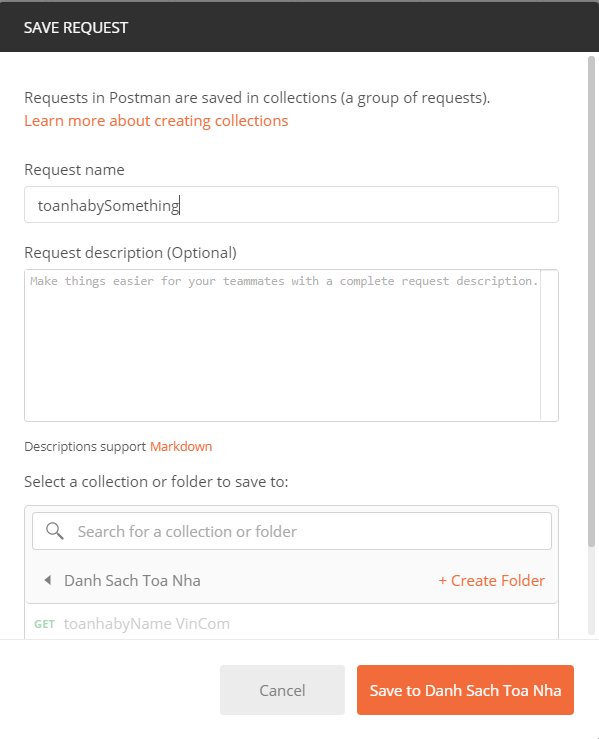
Create a collection first and then create a mock server or you create a mock server based on an existing collection:



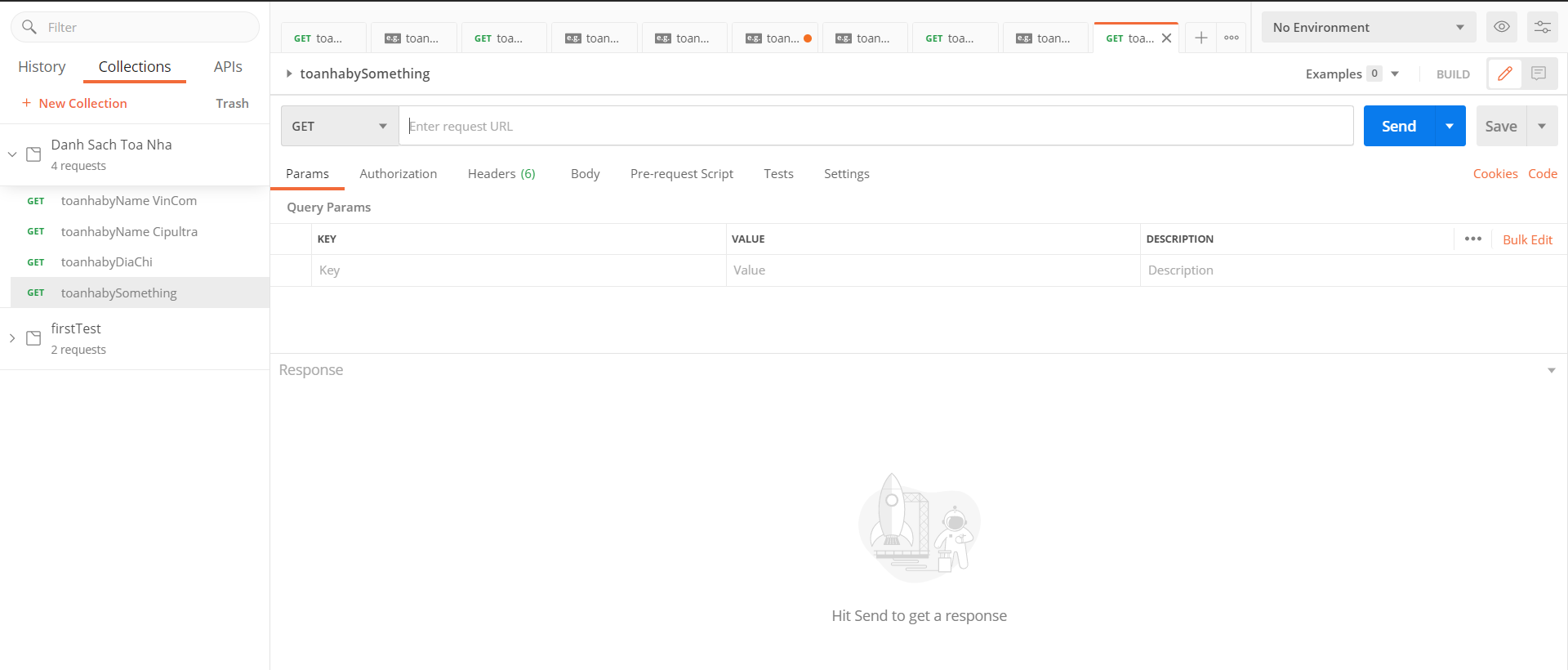
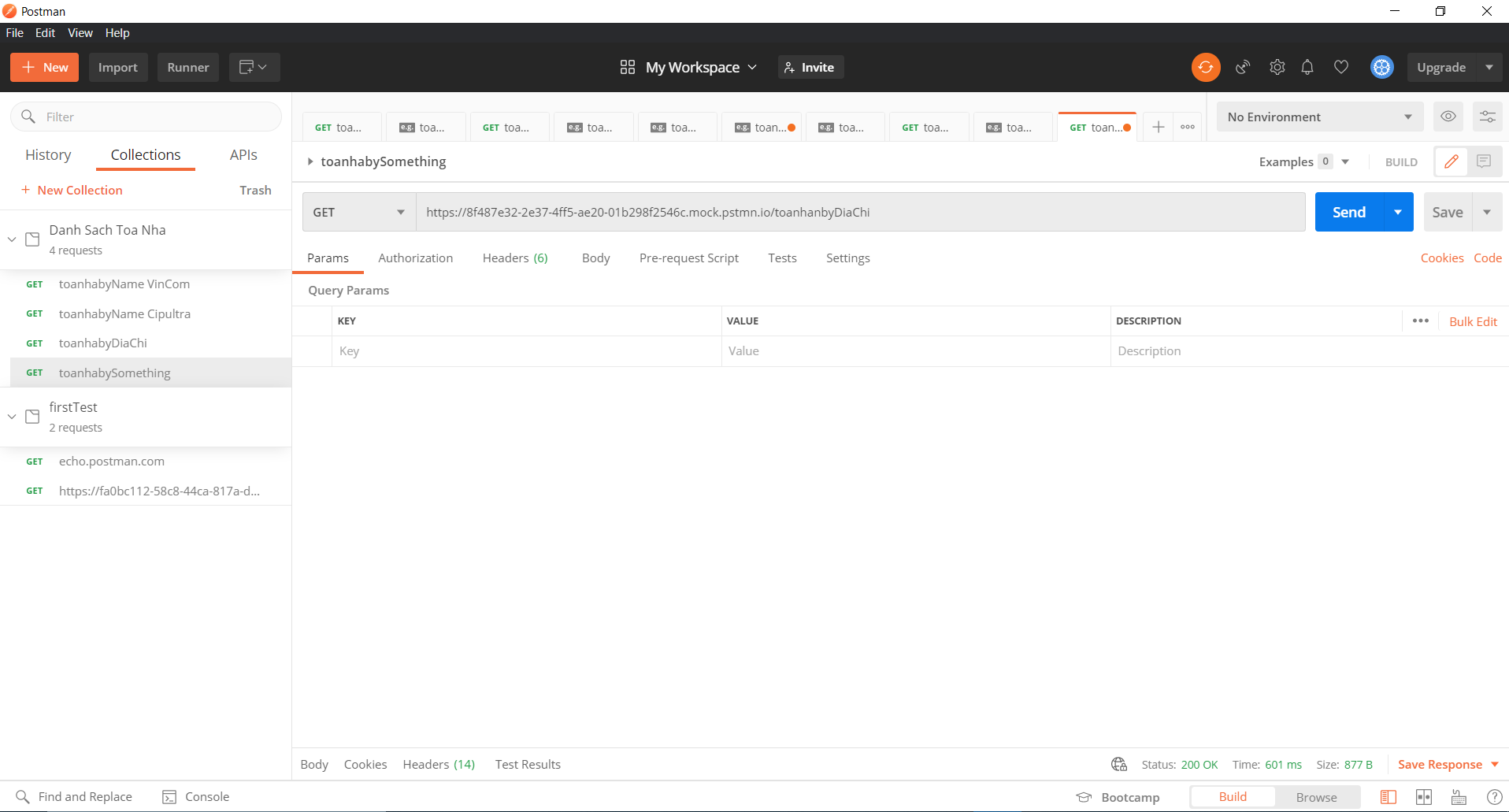
At the last step, note the URL for the mock server, which is needed in the next step.

## Add a request-response example

When you add a request, add it to a collection

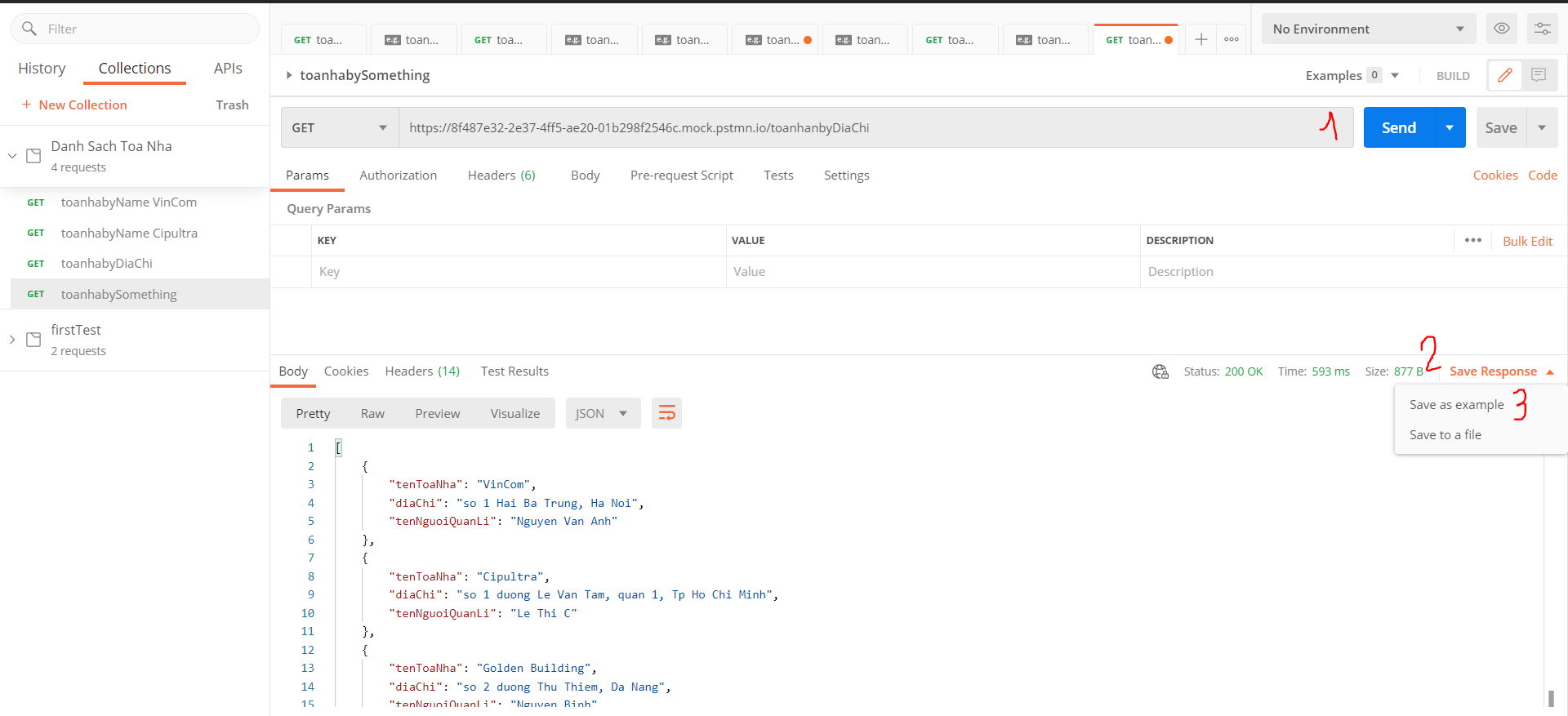


The newly added request has only Query Parameter, but not yet the Response like this:

or 

To open the Response part,

* Step 1: add the address that you created in the “Mock Server” stage
* Step 2: Hit SEND and then “Save Response”/”Save as example”



If the step above is right then the Example Response as below should appear

