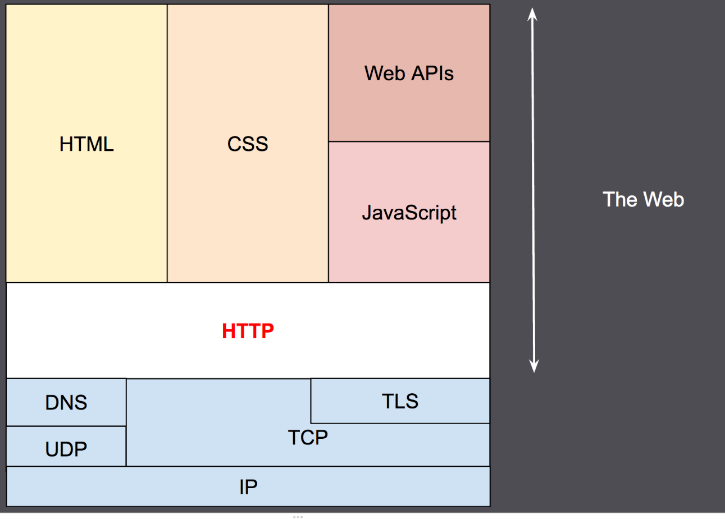
# HTTP

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

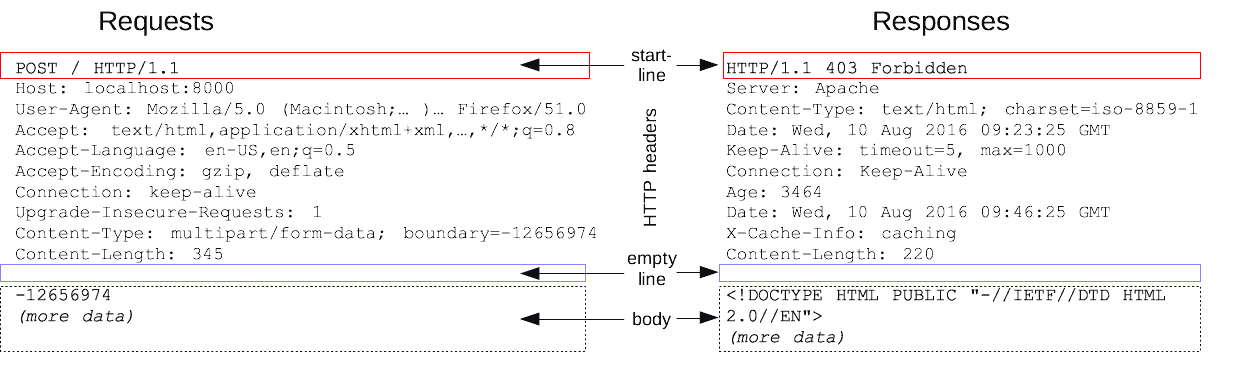
<!DOCTYPE html... (here comes the 29769 bytes of the requested web page)

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)

#### 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>

### HTTP message example

#### 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: 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 3: 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.

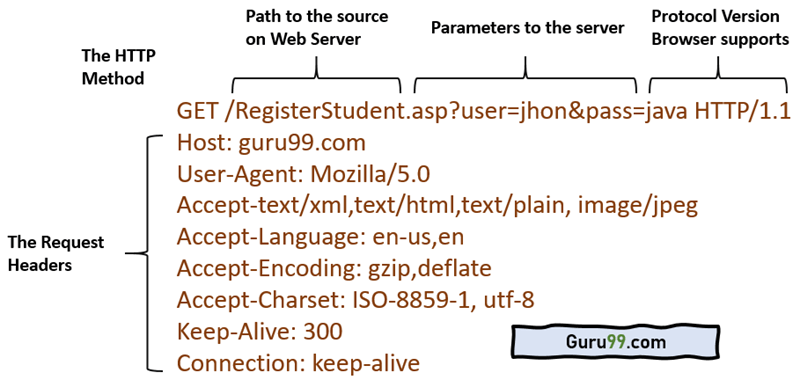
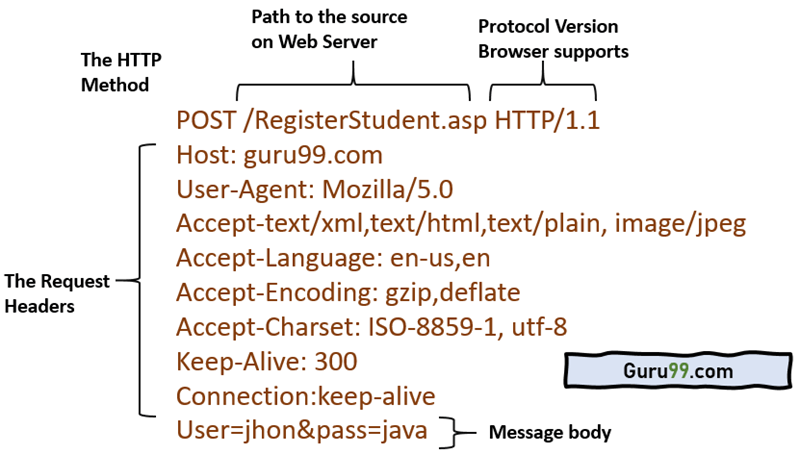
POST/RegisterStudent.asp HTTP/1.1

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.

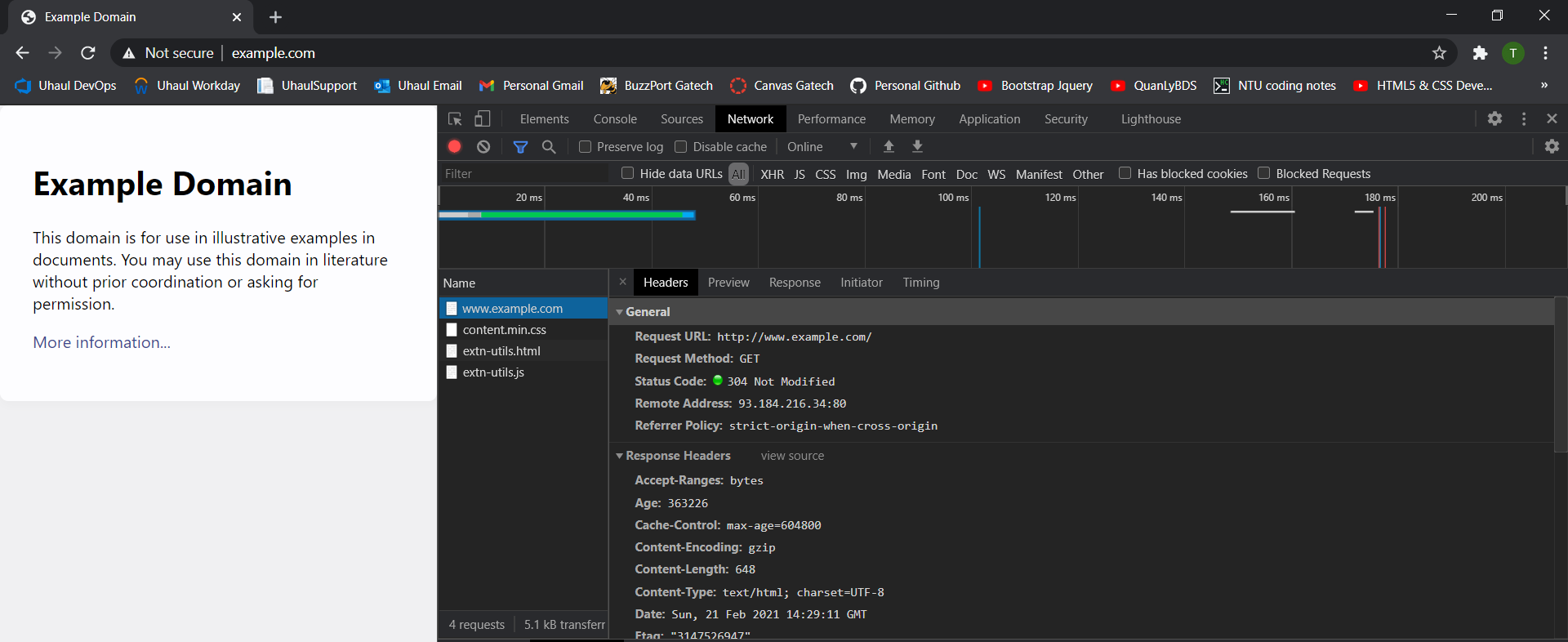
Headers can be grouped into four categories by their context:

* **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 (browser cookies, web 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.

(In brief, A cookie is a piece of data that server sends and browser stores and send back to tell server two requests from a same browser because HTTP protocol is stateless.)

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.

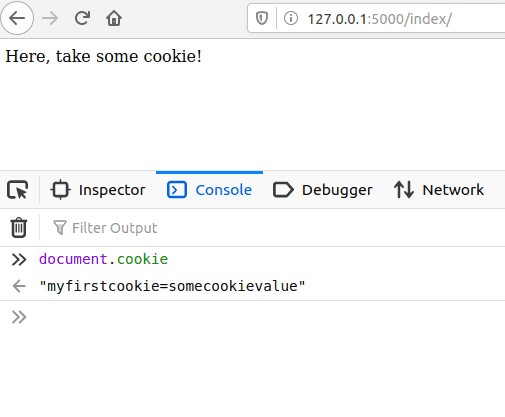
A cookie with the Secure attribute is sent to the server only with an encrypted request over the HTTPS protocol, never with unsecured HTTP (except on localhost), and therefore can't easily be accessed by a [man-in-the-middle](https://developer.mozilla.org/en-US/docs/Glossary/MitM) attacker. Insecure 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:

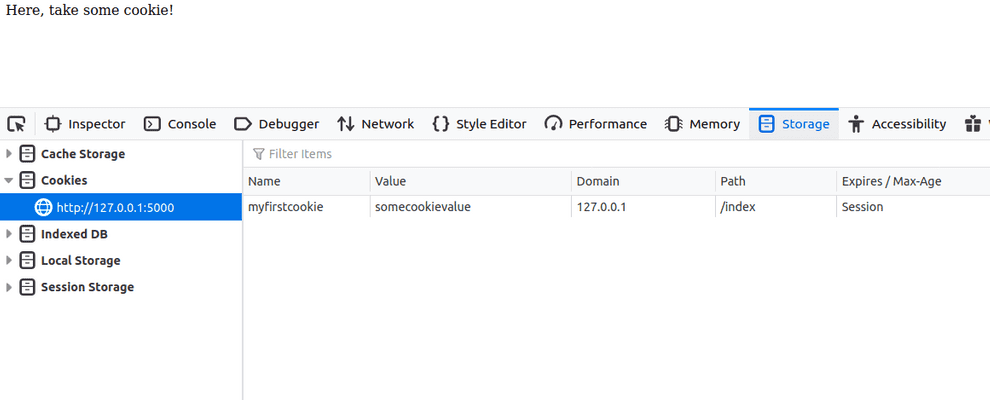
Set-Cookie: id=a3fWa; Expires=Thu, 21 Oct 2021 07:28:00 GMT; Secure; HttpOnly

### 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.

From **this point on for convenience I'll use Flask's** response.set\_cookie() **to create cookies on the backend**.

### 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:

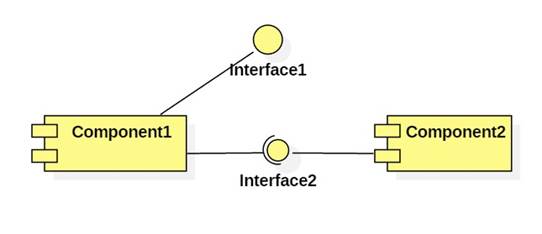
·       Companies A and B are veiled references to Apple and IBM. IBM published details of the PC motherboard, thus creating a market for expansion cards.

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

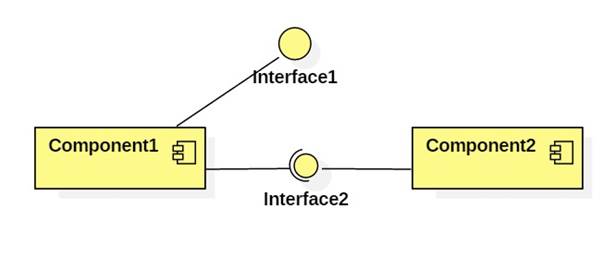
·       Most browsers employ some form of container-component architecture.

## 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

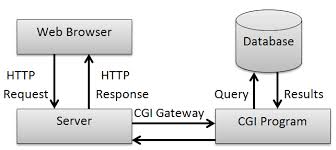
Spring is a standalone framework which substitutes and improves many parts of Java EE. Spring doesn't necessarily require Java EE to run. A bare-bones servlet container like Tomcat is already sufficient. Simply put, Spring is a competitor of Java EE. E.g. "Spring" (standalone) competes EJB/JTA, Spring 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.

# 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.

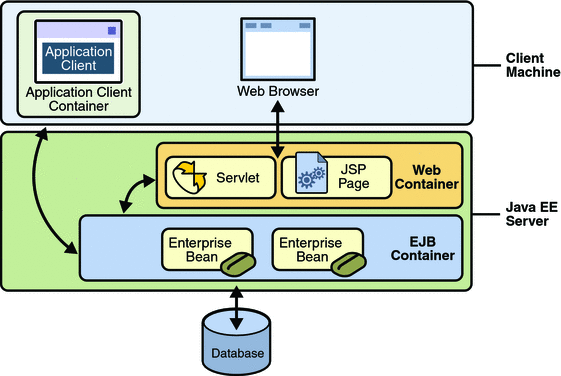
Apache web server (apache HTTPD server) is a web server written in portable C. It mostly serves static content by itself, but there are many add-on modules (some of which come with Apache itself) that let it modify the content and also serve dynamic content written in Perl, PHP, Python, Ruby, or other 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.

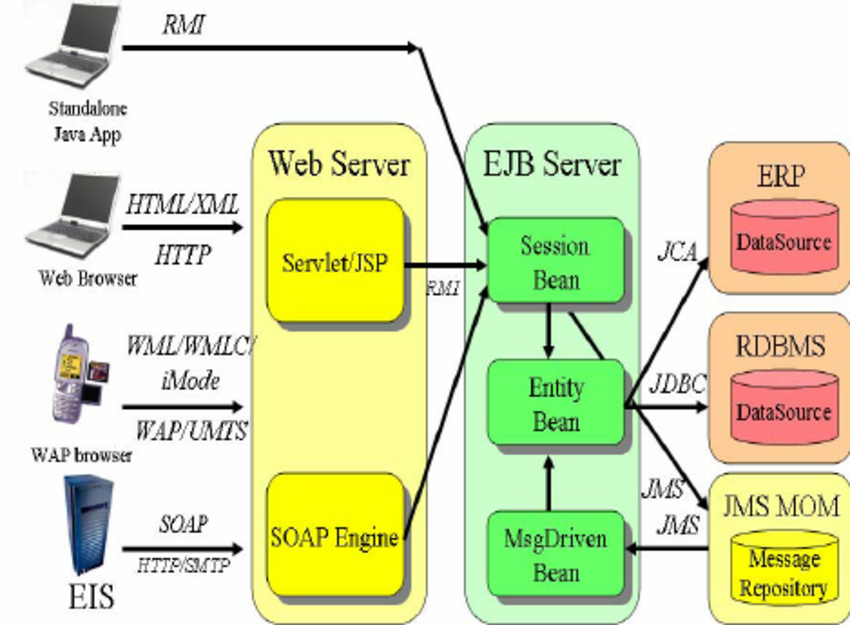
# Java EE

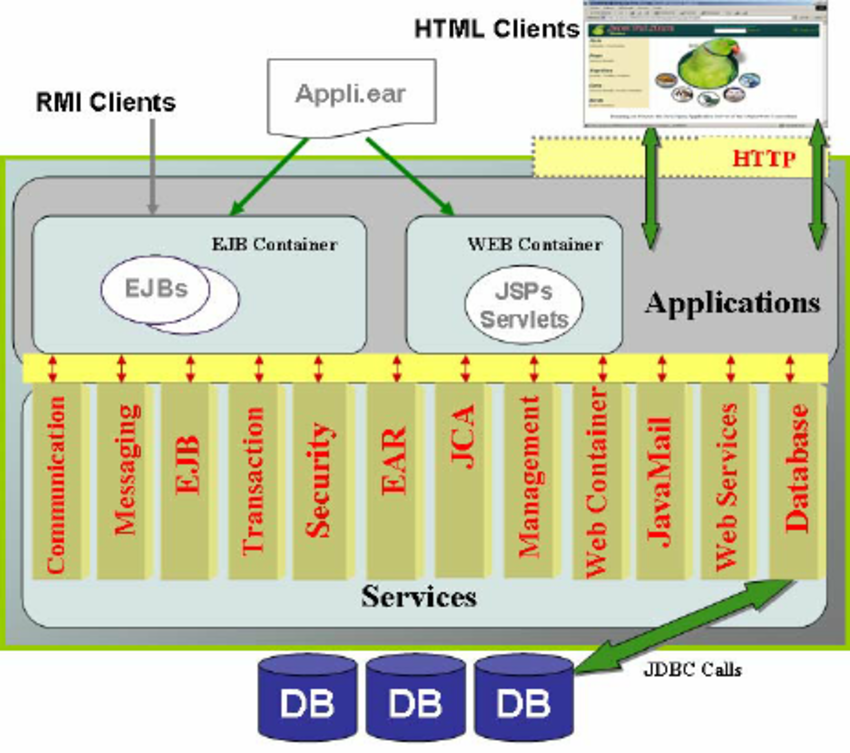
## The best tutorial

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

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







software that serves HTTP request usually by

# 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 your 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

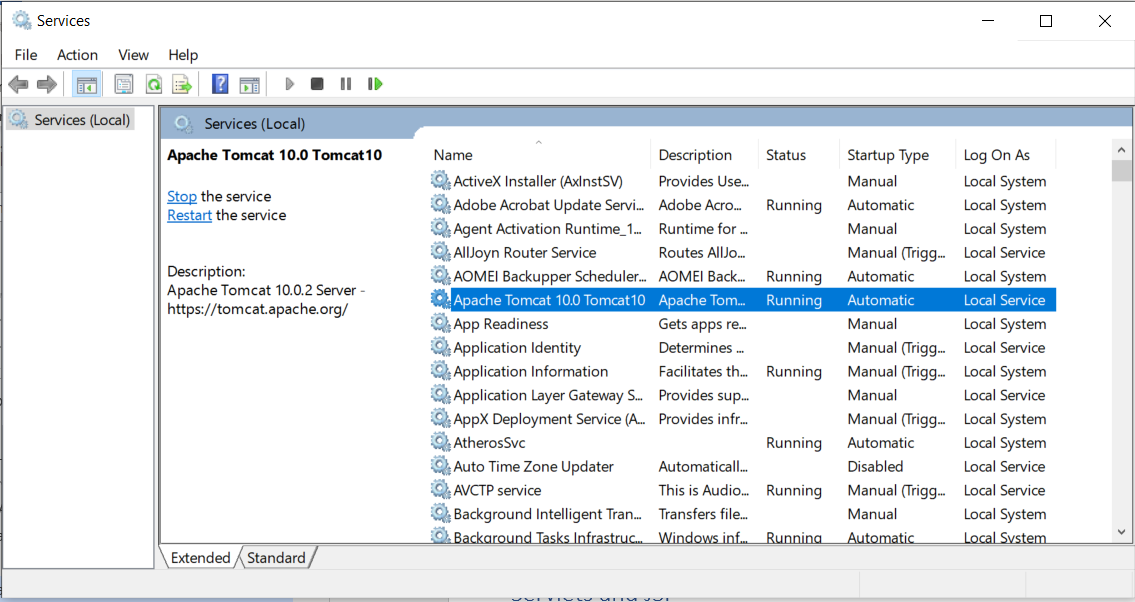
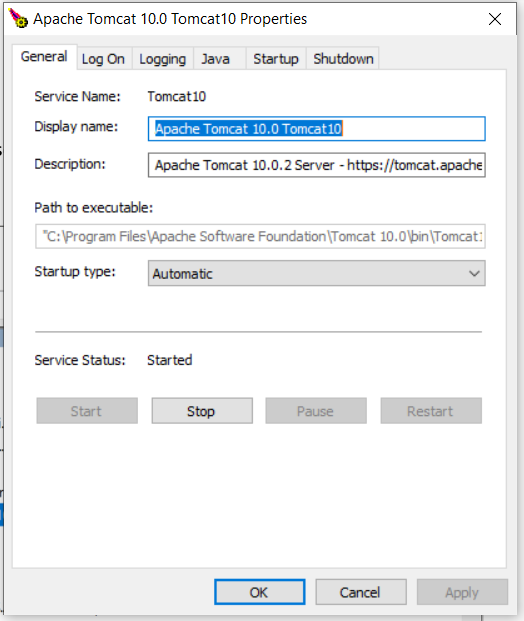
%JAVA\_HOME%\bin\shutdonw.bat for shutting down Tomcat.

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.

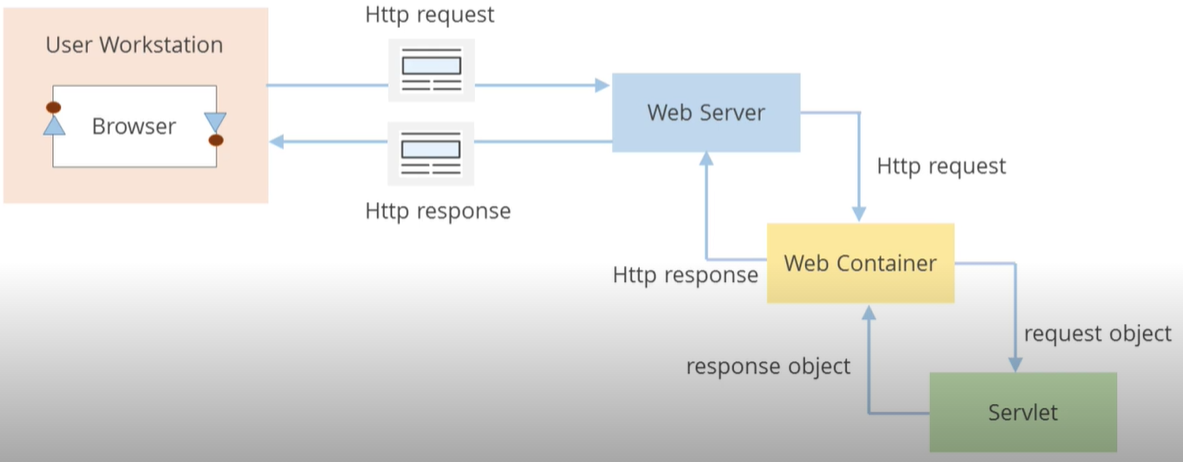
Finally, we can **run a JAR from the command line** if we build it as an [executable JAR](https://www.baeldung.com/executable-jar-with-maven) without using additional software. Or, we can use it as a library. In contrast, we **need a server to execute a WAR**.

# 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.

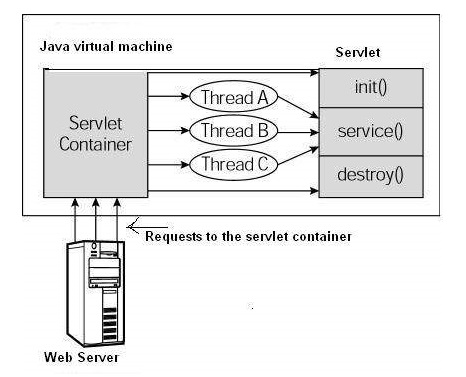




## Life cycle of servlet

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 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 used in methods processing requests

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.

The HttpRequest object has a lot of methods, so I will just cover the most commonly used here. The rest you can read about in the JavaDoc, if you are interested.

#### Parameters (query string in URL request)

The request parameters are parameters that are sent from the browser along with the request. Request parameters are typically sent as part of the URL (in the "query string"), or as part of the body of an HTTP request. For instance:

http://jenkov.com/somePage.html?param1=hello¶m2=world

Notice the "query string" part of the URL: ?param1=hello¶m2=world This part contains two parameters with parameter values:

param1=hello

param2=world

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("param1");**

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

}

You would use the same code, if the request parameters were sent in the body part of the HTTP request. If no parameter exists with the given name, null is returned.

In general, if the browser sends an HTTP GET request, the parameters are included in the query string in the URL. If the browser sends an HTTP POST request, the parameters are included in the body part of the HTTP request.

#### Headers

The request headers are name:value pairs sent by the browser along with the HTTP request. The request headers contain information about e.g. what browser software is being used, what file types the browser is capable of receiving etc. In short, at lot of meta data around the HTTP request.

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.

The Content-Length header contains the number of bytes sent in the HTTP request body, in case the browser sends an HTTP POST request. If the browser sends an HTTP GET request, the Content-Length header is not used, and the above code will return null.

In general, If no header exists with the name passed to getHeader(), null is returned.

#### InputStream

If the browser sends an HTTP POST request, request parameters and other potential data is sent to the server in the HTTP request body. It doesn't have to be request parameters that is sent in the HTTP request body. It could be pretty much any data, like a file or a SOAP request (web service request).

To give you access to the request body of an HTTP POST request, you can obtain an InputStream pointing to the HTTP request body. Here is how it is done:

InputStream requestBodyInput = request.getInputStream();

NOTE: You will have to call this method **before** calling any getParameter() method, because calling the getParameter() method on an HTTP POST request will cause the servlet engine to parse the HTTP request body for parameters. Once parsed, you cannot access the body as a raw stream of bytes anymore.

What you do with the data read from the InputStream is up to you. The servlet engine does not help you parse or interprete that data. You just get it raw.

#### Session

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.

#### ServletContext

You can access the ServletContext object from the HttpRequest object too. The ServletContext contains meta information about the web application. For instance, you can access context parameters set in the web.xml file, you can forward the request to other servlets, and you can store application wide parameters in the ServletContext too.

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

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

As you can see, you have to first get the session object, to get access to the ServletContext object.

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

### HttpResponse

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

The HttpResponse object has a lot of methods, so I will just cover the most commonly used here. The rest you can read about in the JavaDoc, if you are interested.

#### Writing HTML

To send HTML back to the browser, you have to obtain the a PrintWriter from the HttpResponse object. Here is how:

PrintWriter writer = response.getWriter();

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

#### Headers

Just like the request object, the HttpRequest can contain HTTP headers. Headers must be set before any data is written to the response. You set a header on the response object like this:

response.setHeader("Header-Name", "Header Value");

As you can see, a response header is a name, value pair.

#### Writing Text

You can write text back to the browser instead of HTML, like this:

response.setHeader("Content-Type", "text/plain");

PrintWriter writer = response.getWriter();

writer.write("This is just plain text");

First the Content-Type header is set to text/plain. Then a plain text string is written to the writer obtained from the response object.

**Content-Type**

The Content-Type header is a response header that tells the browser the type of the content you are sending back to it. For instance, the content type for HTML is text/html. Similarly, if what you send back to the browser is plain text, you use the content type text/plain.

Here is how you set the Content-Type header on the HttpResponse object:

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

#### Writing Binary Data

You can also write binary data back to the browser instead of text. For instance, you can send an image back, a PDF file or a Flash file or something like that.

Again, you will first have to set the Content-Type header to the type matching the data you are sending back. For instance, the content type for a PNG image is image/png.

You can search for "mime types" in your favourite search engine to find a list of mime types (content types), so you can find the mime type for the content you are sending back.

In order to write binary data back to the browser you cannot use the Writer obtained from response.getWriter(). Afterall, Writer's are intended for text.

Instead you have to use the OutputStream obtained from the response.getOutputStream() method. Here is how:

OutputStream outputStream = response.getOutputStream();

outputStream.write(...);

**Content-Length**

The Content-Length header tells the browser how many bytes your servlet is sending back. If you are sending binary data back you need to set the content length header. Here is how:

response.setHeader("Content-Length", "31642");

#### 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");

## 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>

# Tutorial for servlet

<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/

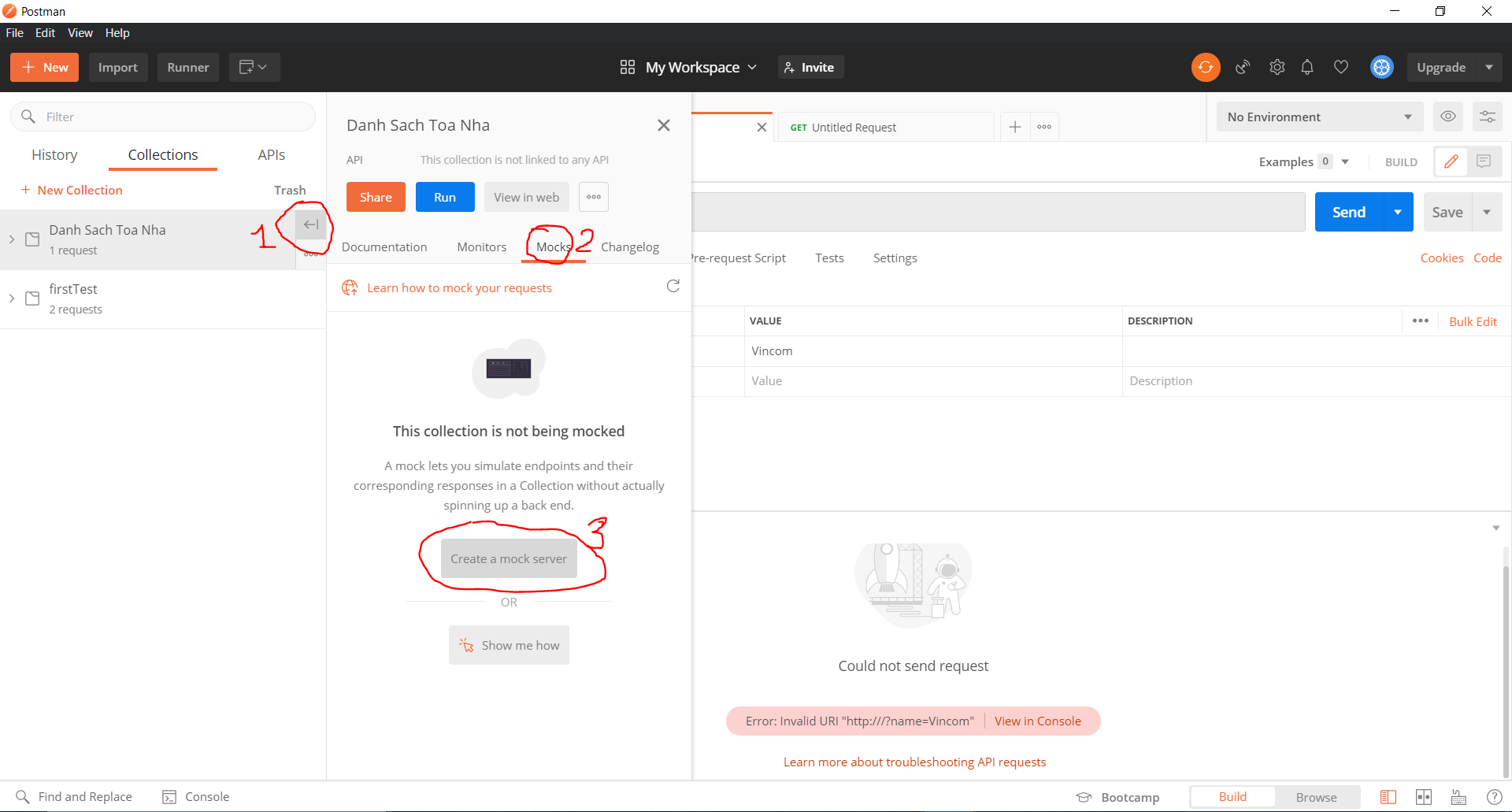
# POSTMAN - Create a mock server, mock response

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

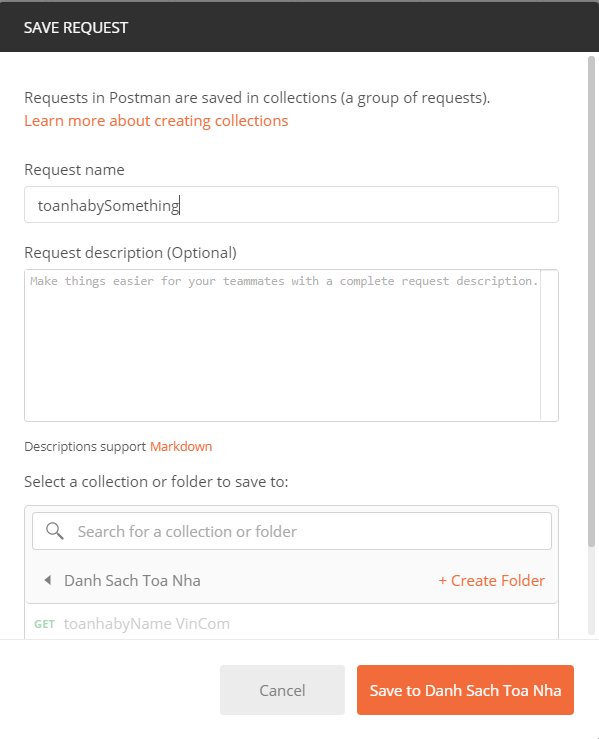
Create a collection first and then create a mock server or you create a mock server based on an existing collection:



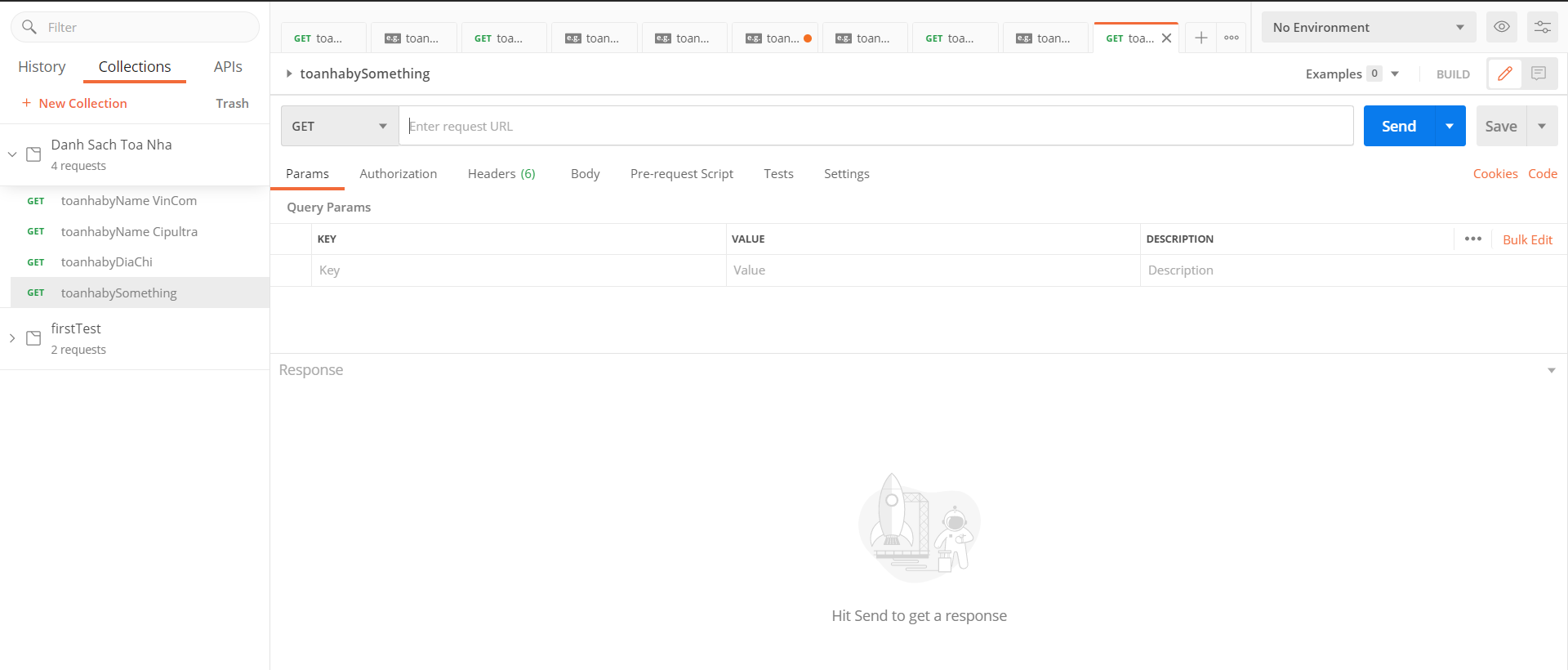
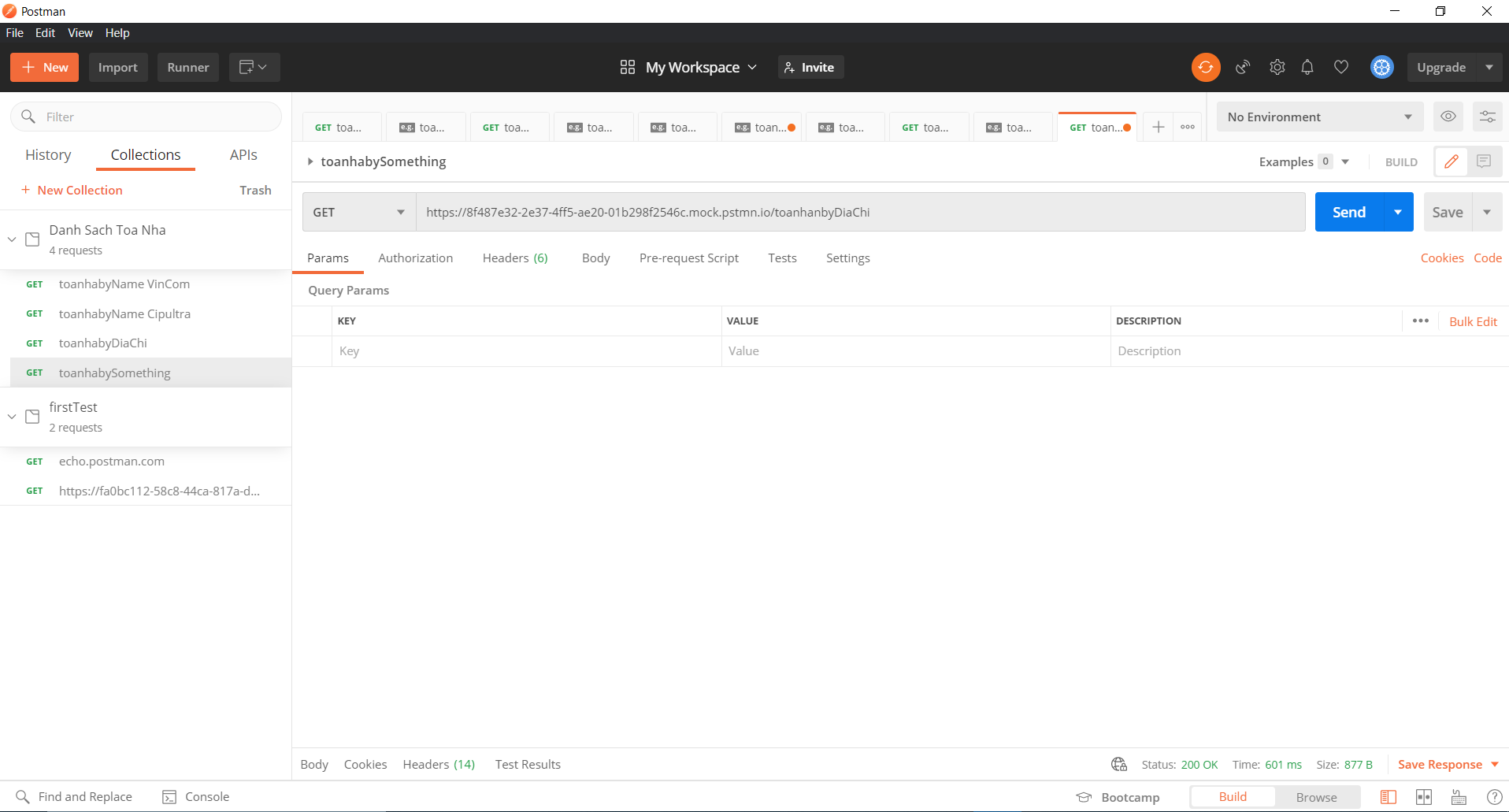
At the last step, note the URL for the mock server, which is needed in the next step.

## Add a request-response example

When you add a request, add it to a collection

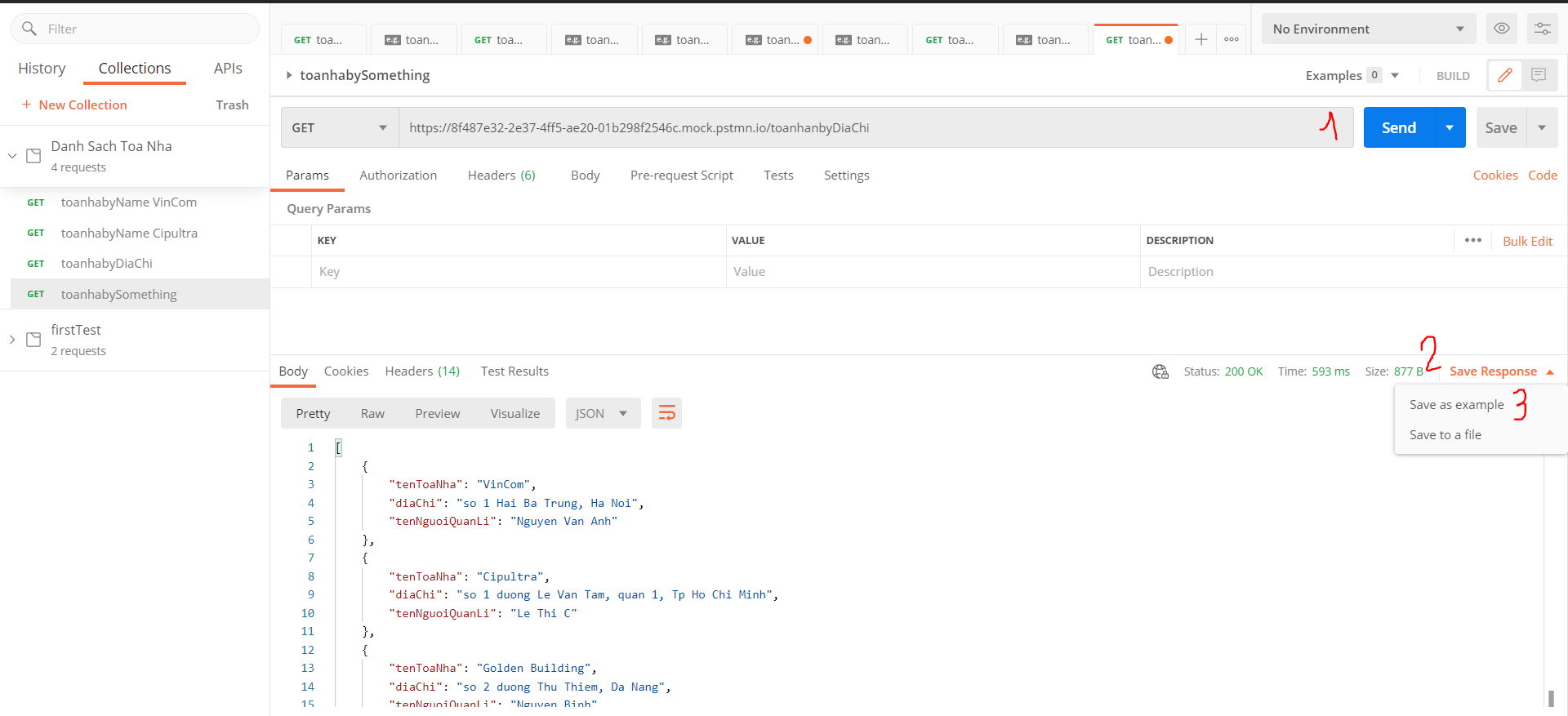


The newly added request has only Query Parameter, but not yet the Response like this:

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

