# `HTTP

What is it?

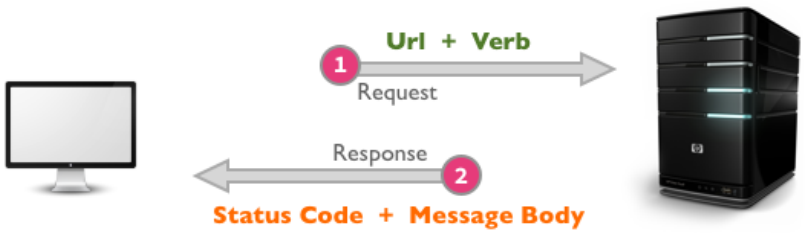
A protocol that defines how to move hypertext between client and server over Internet.

Where is it implemented?

Within browsers, web servers, proxy servers, load balancers, web application containers

**HTTP Overview**

- A browser is an HTTP client because it sends requests to an HTTP server (Web server), which then sends responses back to the client. The standard (and default) port for HTTP servers to listen on is 80, though they can use any port.
- HTTP is used to transmit resources, not just files. A resource is some chunk of information that can be identified by a URL (it's the R in URL).
- Almost all HTTP resources are either files or server-side script output.
- Can contain text or binary data.




**Protocol design considerations**

- **Clients should be able to unambiguously request one file from another**
  - URLs reveal the identity of the particular host with which we want to communicate
  - HTTP uses the client-server model
    - An HTTP client opens a connection and sends a request message to an HTTP server; the server then returns a response message, usually containing the resource that was requested.
    - After delivering the response, the server closes the connection (making HTTP a stateless protocol, i.e. not maintaining any connection information between transactions).

- **There should be a mechanism to differentiate good Client request from a bad request from a Client**
  - With URLs and methods, the client can initiate requests to the server. In return, the server responds with status codes and message payloads. The status code tells the client how to interpret the server response.
  - Error codes:

| Informational: 1## | Success: 2## | Redirection: 3## | Client Error: 4## | Server Error: 5## |
|---|---|---|---|---|

- **Files to serve may be large**
  - If a server wants to start sending a response before knowing its total length, server might use the simple chunked transfer-encoding, which breaks the complete response into smaller chunks and sends them in series.
  - You can identify such a response because it contains the "`Transfer-Encoding: chunked`" header. All HTTP 1.1 clients must be able to receive chunked messages.
  - A chunked message body contains a series of chunks, followed by a line with "0" (zero), followed by optional footers (just like headers), and a blank line. Each chunk consists of two parts:

        - a line with the size of the chunk data, in hex, possibly followed by a semicolon and extra parameters you can ignore (none are currently standard), and ending with CRLF.
        - the data itself, followed by CRLF.
    - So a chunked response might look like the following:

```
HTTP/1.1 200 OK
Date: Fri, 31 Dec 1999 23:59:59 GMT
Content-Type: text/plain
Transfer-Encoding: chunked

1a; ignore-stuff-here
abcdefghijklmnopqrstuvwxyz
10
1234567890abcdef
0
some-footer: some-value
another-footer: another-value
[blank line here]
```

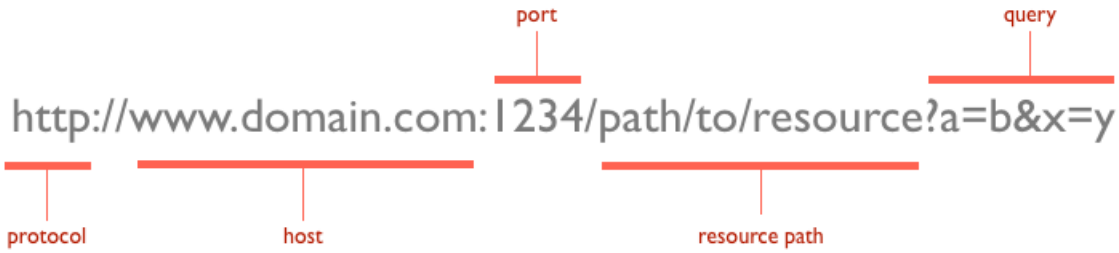
    - Note the blank line after the last footer. The length of the text data is 42 bytes (1a + 10, in hex), and the data itself is abcdefghijklmnopqrstuvwxyz1234567890abcdef. The footers should be treated like headers, as if they were at the top of the response.
    - Footers are rare, but might be appropriate for things like checksums or digital signatures.

- **Clients should be able to request transfer of files only if they don't already have them**
    - The goal of caching in HTTP/1.1 is to eliminate the need to send requests in many cases, and to eliminate the need to send full responses in many other cases.
    - The basic cache mechanisms in HTTP/1.1 are implicit directives to caches where server-specifies expiration times and validators. We use the Cache-Control header for this purpose.
        - Ex: `Cache-control: no-cache`
    - The Cache-Control header allows a client or server to transmit a variety of directives in either requests or responses. These directives typically override the default caching algorithms. The caching directives are specified in a comma-separated list.
    - The following cache **request** directives can be used by the client in its HTTP request:
        - `no-cache`
            - A cache must not use the response to satisfy a subsequent request without successful re-validation with the origin server.
        - `no-store`
            - The cache should not store anything about the client request or server response.
        - `max-age = seconds`
            - Indicates that the client is willing to accept a response whose age is not greater than the specified time in seconds.
        - `max-stale [ = seconds ]`
            - Indicates that the client is willing to accept a response that has exceeded its expiration time. If seconds are given, it must not be expired by more than that time.
        - `min-fresh = seconds`
            - Indicates that the client is willing to accept a response whose freshness lifetime is not less than its current age plus the specified time in seconds.
        - `no-transform`
            - Does not convert the entity-body.
        - `only-if-cached`

            - Does not retrieve new data. The cache can send a document only if it is in the cache, and should not contact the origin-server to see if a newer copy exists.
    - The following cache **response** directives can be used by the server in its HTTP
        - `public`
            - Indicates that the response may be cached by any cache.
        - `private`
            - Indicates that all or part of the response message is intended for a single user and must not be cached by a shared cache.
        - `no-cache`
            - A cache must not use the response to satisfy a subsequent request without successful re-validation with the origin server.
        - `no-store`
            - The cache should not store anything about the client request or server response.
        - `no-transform`
            - Does not convert the entity-body.
        - `must-revalidate`
            - The cache must verify the status of stale documents before using it and expired ones should not be used.
    - **Problems with HTTP 1.0 Caching**
        - Too tied with absolute dates and times - May not work appropriately if the clocks on the server and client are not synchronized
        - Not fine-grained enough – Cannot work if a common cache is present for multiple different clients (“Shared” cache)

**HTTP Lingo**

- **URI** (Uniform Resource Identifier)
    - URI = "http:" "//" host [ ":" port ] [ abs_path [ "?" query ]]




**Methods**

- **GET**: fetch an existing resource.
    - The URL contains all the necessary information the server needs to locate and return the resource.
- **POST**: create a new resource.
    - POST requests usually carry a payload that specifies the data for the new resource.
- **PUT**: update an existing resource.
    - The payload may contain the updated data for the resource.
- **DELETE**: delete an existing resource.
- **HEAD**: this is similar to GET, but without the message body. It's used to retrieve the server headers for a particular resource, generally to check if the resource has changed, via timestamps.

**Utilities for Web browsing**

- **Proxy**: used to run within the clients perimeter (local network).
- **VPN**:(Virtual Private Network) usually outside the client’s perimeter.
- **Gateways**: Forward connections to a local server.

**Versions 1.0, 1.1**

- HTTP 1.1 improvements:
    - Faster response, by allowing multiple transactions to take place over a single persistent connection.
    - Faster response and great bandwidth savings, by adding cache support.
    - Faster response for dynamically-generated pages, by supporting chunked encoding, which allows a response to be sent before its total length is known.
    - Efficient use of IP addresses, by allowing multiple domains to be served from a single IP address.

**Headers**

- **General** -These header fields have general applicability for both request and response messages.
    - `Cache-Control -` used to specify directives that MUST be obeyed by all the caching system.
        - Syntax:
            - `Cache-Control : cache-request-directive|cache-response-directive`
    - `Connection -` allows the sender to specify options that are desired for that particular connection and must not be communicated by proxies over further connections.
        - Syntax:
            - `Connection : "Connection"`
        - HTTP/1.1 defines the "closed" connection option for the sender to signal that the connection will be closed after completion of the response. For example:
            - `Connection: Closed`
        - By default, HTTP 1.1 uses persistent connections, where the connection does not automatically close after a transaction. HTTP 1.0, on the other hand, does not have persistent connections by default. If a 1.0 client wishes to use persistent connections, it uses the keep-alive parameter as follows:
            - `Connection: keep-alive`
    - `Date -` all servers must timestamp every response with a `Date: header` containing the current time, in the form
        - `Date: Fri, 31 Dec 1999 23:59:59 GMT`
        - All responses except those with 100-level status (but including error responses) must include the Date: header.
        - All time values in HTTP use Greenwich Mean Time.
    - `Pragma -` used to include implementation- specific directives that might apply to any recipient along the request/response chain. All pragma directives specify optional behavior from the viewpoint of the protocol; however, some systems MAY require that behavior be consistent with the directives.
        - `Pragma: no-cache -` An intermediary should forward the request toward the origin server even if it has a cached copy of what is being requested.
        - Client's way to tell an intermediary to not serve the response from the cache
    - `Trailer -` indicates that the given set of header fields is present in the trailer of a message encoded with chunked transfer-coding.
        - Syntax:
            - `Trailer : field-name`
        - Message header fields listed in the Trailer header field must not include the following header fields:
            - Transfer-Encoding
            - Content-Length
            - Trailer
    - `Transfer-Encoding -` indicates what type of transformation has been applied to the message body in order to safely transfer it between the sender and the recipient. This is not the same as content-encoding because transfer-encodings are a property of the message, not of the entity-body.
        - Syntax:

            - `Transfer-Encoding: chunked`
        - All transfer-coding values are case-insensitive.
- **Request** - These header fields have applicability only for request messages.
    - `Authorization` - consists of credentials containing the authentication information of the user agent for the realm of the resource being requested
        - Syntax:
            - `Authorization : credentials`
        - The HTTP/1.0 specification defines the BASIC authorization scheme, where the authorization parameter is the string of `username:password` encoded in base64. Following is an example:
            - `Authorization: BASIC Z3Vlc3Q6Z3Vlc3QxMjM=`
        - The value decodes into is guest:guest123 where guest is user ID and guest123 is the password.
    - `User-Agent` - contains information about the user agent originating the request
        - Syntax:
            - `User-Agent : product | comment`
        - Example:
            - `User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)`
    - `If-Modified-Since` - used with a method to make it conditional. If the requested URL has not been modified since the time specified in this field, an entity will not be returned from the server; instead, a 304 (not modified) response will be returned without any message-body.
        - Syntax:
            - `If-Modified-Since : HTTP-date`
        - Example:
            - `If-Modified-Since: Sat, 29 Oct 1994 19:43:31 GMT`
        - If none of the entity tags match, or if "*" is given and no current entity exists, the server must not perform the requested method, and must return a 412 (Precondition Failed) response.
- **Response** - These header fields have applicability only for response messages.
    - `Last-Modified` – indicates the date and time at which the origin server believes the variant was last modified.
        - Syntax:
            - `Last-Modified: HTTP-date`
        - Example:
            - `Last-Modified: Tue, 15 Nov 1994 12:45:26 GMT`
    - `Content-Length` - indicates the size of the entity-body, in decimal number of OCTETs, sent to the recipient or, in the case of the HEAD method, the size of the entity-body that would have been sent, had the request been a GET.
        - Syntax:
            - `Content-Length : DIGITS`
        - Example:
            - `Content-Length: 3495`
        - Any Content-Length greater than or equal to zero is a valid value.
    - `Content-Type` - indicates the media type of the entity-body sent to the recipient or, in the case of the HEAD method, the media type that would have been sent, had the request been a GET.
        - Syntax:
            - `Content-Type : media-type`
        - Example:
            - `Content-Type: text/html; charset=ISO-8859-4`
- **Entity** - These header fields define meta-information about the entity-body or, if no body is present, about the resource identified by the request
    - `Etag` - provides the current value of the entity tag for the requested variant.
        - Syntax:

            - `ETag : entity-tag`
        - Examples:
            - `ETag: "xyzzy"`
            - `ETag: W/"xyzzy"`
            - `ETag: ""`
    - `If-None-Match` - Request header. Used with a method to make it conditional. This header requests the server to perform the requested method only if one of the given value in this tag matches the given entity tags represented by ETag.
        - Syntax:
            - `If-None-Match : entity-tag`
        - Examples:
            - `If-None-Match: "xyzzy"`
            - `If-None-Match: "xyzzy", "r2d2xxxx", "c3piozzzz"`
            - `If-None-Match: *`
    - `If-Match` - Request header. Used with a method to make it conditional. This header requests the server to perform the requested method only if the given value in this tag matches the given entity tags represented by ETag.
        - Syntax:
            - `If-Match : entity-tag`
        - An asterisk (*) matches any entity, and the transaction continues only if the entity exists.
        - Examples:
            - `If-Match: "xyzzy"`
            - `If-Match: "xyzzy", "r2d2xxxx", "c3piozzzz"`
            - `If-Match: *`
        - If none of the entity tags match, or if "*" is given and no current entity exists, the server must not perform the requested method, and must return a 412 (Precondition Failed) response.
    - `Expires -` gives the date/time after which the response is considered stale.
        - Syntax:
            - `Expires : HTTP-date`
        - Example:
            - `Expires: Thu, 01 Dec 1994 16:00:00 GMT`

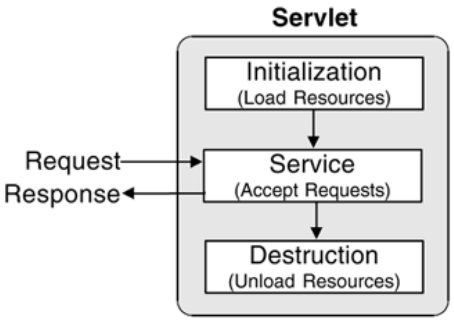
**Session tracking mechanisms**

- **Cookies**
    - Cookies are the mostly used technology for session tracking. Cookie is a key value pair of information, sent by the server to the browser. This should be saved by the browser in its space in the client computer. Whenever the browser sends a request to that server it sends the cookie along with it. Then the server can identify the client using the cookie.
    - **Advantages:**
        - Simple. Session tracking is easy to implement and maintain using the cookies.
        - Don't need to send data back
    - **Disadvantages:**
        - Size and number of cookies stored are limited.
        - Stored as plain-text in a specific directory, everyone can view and modify them. Personal information is exposed.
        - Won't work if the security level set too high in browser.
        - Users can opt to disable cookies using their browser preferences. In such case, the browser will not save the cookie at client computer and session tracking fails.
    - **Attributes**

        - **Domain and Path:** define the scope of the cookie. They essentially tell the browser what website the cookie belongs to.
        - **Expires and Max-Age:** defines a specific date and time for when the browser should delete the cookie.
        - **Secure and HttpOnly:** do not have associated values. Rather, the presence of just their attribute names indicates that their behaviors should be enabled.
- **URL rewriting**
    - Original URL: http://server:port/servlet/ServletName
    - Rewritten URL: http://server:port/servlet/ServletName?sessionid=7456
    - When a request is made, additional parameter is appended with the url. In general added additional parameter will be sessionid or sometimes the userid. It will suffice to track the session. This type of session tracking doesn't need any special support from the browser.
    - **Advantages:**
        - It will always work whether cookie is disabled or not (browser independent).
        - Extra form submission is not required on each page.
        - Data is appended on the URL => easy to debug.
    - **Disadvantages:**
        - implementing this type of session tracking is tedious.
            - Need to keep track of the parameter as a chain link until the conversation completes and also should make sure that, the parameter doesn't clash with other application parameters.
        - Works only with links.
        - Can send Only textual information.
        - URL is lengthy and the length of the URL is limited, can't store much information.
        - URL contains data. If you send this URL to your friends, they might see your information.
        - If the user surfs to a different site, or to a static section of the same site, the state is lost.
- **Session Objects**
    - Session objects usually use either cookies or URL rewriting (depends on security setting of browser) to make it function. Session object is representation of a user session. User Session starts when a user opens a browser and sends the first request to server. Session object is available in all the requests (in entire user session) so attributes stored in HTTP session in will be available in any servlet or in a jsp.
    - When session is created, server generates a unique ID and attach that ID with the session. Server sends back this Id to the client and the browser sends back this ID with every request of that user to server with which server identifies the user
        - **Advantages:**
            - Easy to implement.
            - Ensures data durability, since session state retains data even if ASP.NET work process restarts as data in Session State is stored in other process space.
            - It works in the multi-process configuration, thus ensures platform scalability.

- **Disadvantages:**
    - Since data in session state is stored in server memory, it is not advisable to use session state when working with large sum of data. Session state variable stays in memory until you destroy it, so too many variables in the memory effect performance

# Servlets




What are they?

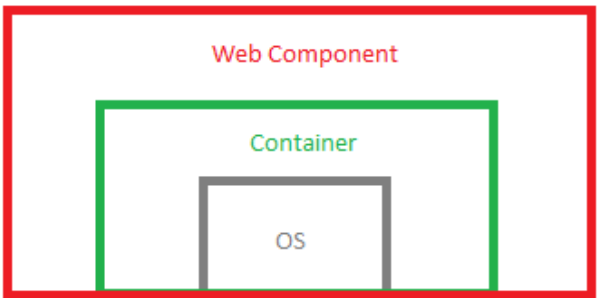
- The primary purpose of the Servlet specification is to define a robust mechanism for sending content to a client as defined by the Client/Server model. Servlets are most popularly used for generating dynamic content on the Web and have native support for HTTP
- **Life Cycle**
    - **Initialization Phase**
        - Creation and initialization of resources the Servlet may need to service requests.
    - **Service Phase**
        - All interactions with requests until the Servlet is destroyed.
    - **Destruction Phase**
        - When a Servlet is being removed from use by a container.
- **How Servlet container and web server process a request?**
    - Web server receives HTTP request
    - Web server forwards the request to servlet container
    - The servlet is dynamically retrieved and loaded into the address space of the container, if it is not in the container.
    - The container invokes the init() method of the servlet for initialization(invoked once when the servlet is loaded first time)
    - The container invokes the service() method of the servlet to process the HTTP request, i.e., read data in the request and formulate a response. The servlet remains in the container's address space and can process other HTTP requests.
    - Web server return the dynamically generated results to the correct location

**Deployment descriptor**

- describes how the web application should be deployed
- web.xml
    - The contents of this file direct a deployment tool to deploy a module or application with the specified security settings, and describes other specific configuration requirements and/or container options
    - `<web-app>` represents the whole application.
    - `<servlet>` is sub element of <web-app> and represents the servlet.
    - `<servlet-name>` is sub element of <servlet> represents the name of the servlet.
    - `<servlet-class>` is sub element of <servlet> represents the class of the servlet.
    - `<servlet-mapping>` is sub element of <web-app>. It is used to map the servlet.
    - `<url-pattern>` is sub element of <servlet-mapping>. This pattern is used at client side to invoke the servlet.
- At runtime J2EE server reads the deployment descriptor and understands it and then acts upon the component or module based the information mentioned in descriptor.

**Servlet container**

- The basic idea of Servlet container is using Java to dynamically generate the web page on the server side. So servlet container is essentially a part of a web server that manages the servlets.




**ServletContext**

- Represents the context for an entire web application (all Servlets)
- Initialization information to all the servlets can be obtained from ServletContext
- An object of ServletContext is created by the web container at time of deploying the project. This object can be used to get configuration information from web.xml file. There is only one ServletContext object per web application. This can be set using `<artifactId>NAME-OF-CONTEXTROOT</artifactId>`
- If any information is shared to many servlets, it is better to provide it from the web.xml file using the `<context-param>` element.
- **Advantages:**
    - Easy to maintain if any information is shared to all the servlet, it is better to make it available for all the servlet. We provide this information from the web.xml file, so if the information is changed, we don't need to modify the servlet. Thus it removes maintenance problem.
- **Usage of ServletContext Interface**
    - The object of ServletContext provides an interface between the container and servlet.
    - The ServletContext object can be used to get configuration information from the web.xml file.
    - The ServletContext object can be used to set, get or remove attribute from the web.xml file.
    - The ServletContext object can be used to provide inter-application communication.

**ServletConfig**

- Used to initialize a specific servlet
- ServletContext is implemented by the servlet container for all servlet to communicate with its servlet container
- ServletContext object is contained within the ServletConfig object
- An object of ServletConfig is created by the web container for each servlet. This object can be used to get configuration information from web.xml file.
- If the configuration information is modified from the web.xml file, we don't need to change the servlet. So it is easier to manage the web application if any specific content is modified from time to time.
- **Advantages:**
    - The core advantage of ServletConfig is that you don't need to edit the servlet file if information is modified from the web.xml file.

**doGet**

- **HTTP GET: form data is part of the URL**
- Overriding this method to support a GET request also automatically supports an HTTP HEAD request. A HEAD request is a GET request that returns no body in the response, only the request header fields.
- 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.
- parameters are appended to the URL and sent along with the header information.
- shall be used when small amount of data and insensitive data like a query has to be sent as a request

- Called by the server (via the service method) to allow a servlet to handle a GET request.
- The GET method should be safe, that is, without any side effects for which users are held responsible. For example, most form queries have no side effects.
- The GET method should also be idempotent, meaning that it can be safely repeated. Sometimes making a method safe also makes it idempotent. For example, repeating queries is both safe and idempotent, but buying a product online or modifying data is neither safe nor idempotent.

**doPost**

- **HTTP POST: the form data appears in the message body**
- A POST request results from an HTML form that specifically lists POST as the METHOD and it should be handled by doPost() method.
- the parameters are sent separately
- shall be used when comparatively large amount of sensitive data has to be sent. Examples are sending data after filling up a form or sending login id and password
- Called by the server (via the service method) to allow a servlet to handle a POST request.
- This method does not need to be either safe or idempotent. Operations requested through POST can have side effects for which the user can be held accountable, for example, updating stored data or buying items online.
- If the HTTP POST request is incorrectly formatted, doPost returns an HTTP "Bad Request" message.

**HttpServletRequest object**

- Extends the ServletRequest interface to provide request information for HTTP servlets.
- The servlet container creates an HttpServletRequest object and passes it as an argument to the servlet's service methods (doGet, doPost, etc).
- Contains all the client's request information

**HttpServletResponse object**

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

**Concurrency**

- the servlet container handles multiple requests by spawning multiple threads, each thread executing the service() method of a single instance of the servlet.

**Advantages of servlets**

A servlet can be imagined to be as an applet running on the server side. Some of the other server side technologies available are Common Gateway Interface (CGI), server side JavaScript and Active Server Pages (ASP). Advantages of servlets over these server side technologies are as follows:

- **Persistent:** Servlets remain in memory until explicitly destroyed. This helps in serving several incoming requests. Servlets establishes connection only once with the database and can handle several requests on the same database. This reduces the time and resources required to establish connection again and again with the same database. Whereas, CGI programs are removed from the memory once the request is processed and each time a new process is initiated whenever new request arrives.
- **Portable:** Since servlets are written in Java, they are portable. That is, servlets are compatible with almost all operating systems. The programs written on one operating system can be executed on other operating system.
- **Server-independent:** Servlets are compatible with any web server available today. Most of the software vendors today support servlets within their web server products. On the other hand, some of the server side

technologies like server side JavaSricpt and ASP can run on only selected web servers. The CGI is compatible with the web server that has features to supports it.

- **Protocol-independent:** Servlets can be created to support any of the protocols like FTP commands, Telnet sessions, NNTP newsgroups, etc. It also provides extended support for the functionality of HTTP protocol.
- **Extensible:** Servlets being written in Java, can be extended and polymorphed into the objects that suits the user requirement.
- **Secure:** Since servlets are server side programs and can be invoked by web server only, they inherit all the security measures taken by the web server. They are also safe from the problems related to memory management as Java does not support the concept of pointers and perform garbage collection automatically.
- **Fast:** Since servlets are compiled into bytecodes, they can execute more quickly as compared to other scripting languages. The bytecode compilation feature helps servlets to give much better performance. In addition, it also provides advantage of strong error and type checking.

**What are some of the mechanisms available for passing parameters to a Servlet?**

- use ServletConfig “init-param” in web.xml
    - Put your parameter value in “init-param” and make sure inside the “servlet” element
- **Query Parameters** - Parameters that are passed with the resource URL as a query string
    - When the parameters are concerned with the resource itself
        - Search queries that will provide subset of resources in response
- **Request Headers**
    - When the parameters are concerned with the request/response interaction
        - cookies
- **Request Body**
    - When the parameters are related to the resource's content
        - Form submission

## Layered Architecture

**What is it?**

- A multilayered software architecture is a software architecture that uses many layers for allocating the different responsibilities of a software product.

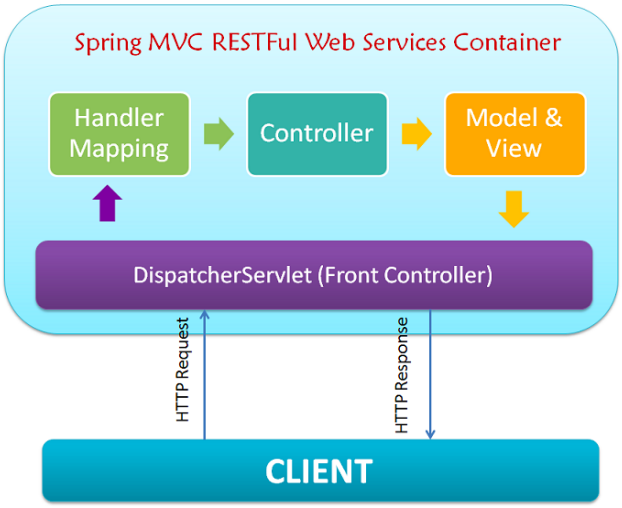
**Why do we use it?**

- clear separation of responsibilities — each layer being only responsible for itself
- exposed workflow — as opposed to the spaghetti code we’ve all see way too many times
- ability to replace one or several layers implementation with minimum effort and side effects

**Disadvantage:**

- When a lower level fails, it can cause the entire application to fail

**Splitting your code into layers**




- **Spring**
  - Framework that easies writing of web applications
  - How it works
    - After receiving an HTTP request, DispatcherServlet consults the HandlerMapping to call the appropriate **Controller**.
    - The **Controller** takes the request and calls the appropriate **service methods** based on used GET or POST method. The **service method** will set model data based on defined business logic and returns view name to the **DispatcherServlet**.
    - The **DispatcherServlet** will take help from **ViewResolver** to pickup the defined view for the request.
    - Once view is finalized, The **DispatcherServlet** passes the model data to the view which is finally rendered on the browser.
  - You need to map requests that you want the DispatcherServlet to handle, by using a URL mapping in the web.xml file. The following is an example to show declaration and mapping for HelloWeb DispatcherServlet example:

```
<web-app id="WebApp_ID" version="2.4"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <display-name>Spring MVC Application</display-name>

   <servlet>
      <servlet-name>HelloWeb</servlet-name>
      <servlet-class>
         org.springframework.web.servlet.DispatcherServlet
      </servlet-class>
      <load-on-startup>1</load-on-startup>
   </servlet>

   <servlet-mapping>
      <servlet-name>HelloWeb</servlet-name>
      <url-pattern>*.jsp</url-pattern>
   </servlet-mapping>

</web-app>
```

- upon initialization of HelloWeb DispatcherServlet, the framework will try to load the application context from a file named [servlet-name]-servlet.xml located in the application's WebContent/WEB-INF directory. In this case our file will be HelloWeb-servlet.xml.
- Next, <servlet-mapping> tag indicates what URLs will be handled by the which DispatcherServlet. Here all the HTTP requests ending with .jsp will be handled by the HelloWeb DispatcherServlet.

- Now, let us check the required configuration for HelloWeb-servlet.xml file, placed in your web application's WebContent/WEB-INF directory:

```
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
   http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
   http://www.springframework.org/schema/context
   http://www.springframework.org/schema/context/spring-context-3.0.xsd">

   <context:component-scan base-package="com.tutorialspoint" />

   <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
      <property name="prefix" value="/WEB-INF/jsp/" />
      <property name="suffix" value=".jsp" />
   </bean>

</beans>
```

- Following are the important points about HelloWeb-servlet.xml file:
    - The [servlet-name]-servlet.xml file will be used to create the beans defined, overriding the definitions of any beans defined with the same name in the global scope.
    - The <context:component-scan...> tag will be use to activate Spring MVC annotation scanning capability which allows to make use of annotations like @Controller and @RequestMapping etc.
    - The InternalResourceViewResolver will have rules defined to resolve the view names. As per the above defined rule, a logical view named hello is delegated to a view implementation located at /WEB-INF/jsp/hello.jsp .
    - DispatcherServlet delegates the request to the controllers to execute the functionality specific to it. The **@Controller** annotation indicates that a particular class serves the role of a controller. The @RequestMapping annotation is used to map a URL to either an entire class or a particular handler method.

```
@Controller
@RequestMapping("/hello")
public class HelloController{

   @RequestMapping(method = RequestMethod.GET)
   public String printHello(ModelMap model) {
      model.addAttribute("message", "Hello Spring MVC Framework!");
      return "hello";
   }

}
```

        - The **@Controller** annotation defines the class as a Spring MVC controller. Here, the first usage of **@RequestMapping** indicates that all handling methods on this controller are relative to the /hello path. Next annotation @RequestMapping(method = RequestMethod.GET) is used to declare the printHello() method as the controller's default service method to handle HTTP GET request. You can define another method to handle any POST request at the same URL.
    - **@Component** – marks a java class as a bean so the component-scanning mechanism of spring can pick it up and pull it into the application context.
    - **@Service** - used in your service layer and annotates classes that perform service tasks
    - **@PathVariable** - used for accessing the values from the URI template
    - **@RequestParam** - used for accessing the values of the query parameters
    - **@Controller** - indicates that a particular class serves the role of a controller
- **JAX-RS**
    - Resources - web service layer components focused on web service related processing such as translating input, preparing the response, setting the response code
    - Services -

- **Spring vs JAX-RS**
  - JAX-RS targets the development of web services (as opposed to HTML web applications) while Spring MVC has its roots in web application development
  - Spring MVC does not have the notion of root resources and sub-resources
  - JAX-RS has the notion of "root" resources (marked with @Path) and sub-resources
  - In Spring MVC controllers are always created as singletons.
  - In JAX-RS resources can also be created as singletons.
  - Both frameworks also support the use of regular expressions for extracting path variables
- **Developing against an Interface**
  - The Java interface is a development contract. It ensures that a particular object satisfies a given set of methods. Interfaces are used throughout the Java API to specify the necessary functionality for object interaction.
  - Coding to interfaces is a technique by which developers can expose certain methods of an object to other objects in the system. The developers who receive implementations of these interfaces have the ability to code to the interface in place of coding to the object itself. In other words, the developers would write code that did not interact directly with an object as such, but rather with the implementation of that object's interface.
  - Another reason to code to interfaces rather than to objects is that it provides higher efficiency in the various phases of a system's lifecycle:
    - Design: the methods of an object can be quickly specified and published to all affected developers
    - Development: the Java compiler guarantees that all methods of the interface are implemented with the correct signature and that all changes to the interface are immediately visible to other developers
    - Integration: there is the ability to quickly connect classes or subsystems together, due to their well-established interfaces
    - Testing: interfaces help isolate bugs because they limit the scope of a possible logic error to a given subset of methods
  - Interfaces give developers the ability to expose and limit certain methods and information to the users of their objects without changing the permissions and internal structure of the object itself. The use of interfaces can help eliminate the pesky bugs that appear when code developed by multiple development teams is integrated.
- **Concept of dependencies**
  - Dependency Injection (or sometime called wiring) helps in gluing these classes together and same time keeping them independent.
  - Consider you have an application which has a text editor component and you want to provide spell checking. Your standard code would look something like this:

```
public class TextEditor {
   private SpellChecker spellChecker;

   public TextEditor() {
      spellChecker = new SpellChecker();
   }
}
```

  - What we've done here is create a dependency between the TextEditor and the SpellChecker. In an inversion of control scenario we would instead do something like this:

```
public class TextEditor {
   private SpellChecker spellChecker;

   public TextEditor(SpellChecker spellChecker) {
      this.spellChec+ker = spellChecker;
   }
}
```

- Here TextEditor should not worry about SpellChecker implementation. The SpellChecker will be implemented independently and will be provided to TextEditor at the time of TextEditor instantiation and this entire procedure is controlled by the Spring Framework.
- Here, we have removed the total control from TextEditor and kept it somewhere else (ie. XML configuration file) and the dependency ( ie. class SpellChecker) is being injected into the class TextEditor through a Class Constructor. Thus flow of control has been "inverted" by **Dependency Injection** (DI) because you have effectively delegated dependances to some external system.
- Second method of injecting dependency is through **Setter Methods** of TextEditor class where we will create SpellChecker instance and this instance will be used to call setter methods to initialize TextEditor's properties.
- **Constructor-based dependency injection**
    - Constructor-based DI is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on other class.
- **Setter-based dependency injection**
    - 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.
- You can mix both, Constructor-based and Setter-based DI but it is a good rule of thumb to use constructor arguments for mandatory dependencies and setters for optional dependencies.
- Code is cleaner with the DI principle and decoupling is more effective when objects are provided with their dependencies. The object does not look up its dependencies, and does not know the location or class of the dependencies rather everything is taken care by the Spring Framework.
- **Autowiring**
    - The @Autowired annotation can apply to bean property setter methods, non-setter methods, constructor and properties.
    - The Spring container can autowire relationships between collaborating beans without using <constructor-arg> and <property> elements which helps cut down on the amount of XML configuration you write for a big Spring based application.
    - There are following autowiring modes which can be used to instruct Spring container to use autowiring for dependency injection. You use the autowire attribute of the <bean/> element to specify autowire mode for a bean definition.
        - **no** - This is default setting which means no autowiring and you should use explicit bean reference for wiring. You have nothing to do special for this wiring. This is what you already have seen in Dependency Injection chapter.
        - **byName** - Autowiring by property name. Spring container looks at the properties of the beans on which autowire attribute is set to byName in the XML configuration file. It then tries to match and wire its properties with the beans defined by the same names in the configuration file.
        - **byType -** Autowiring by property datatype. Spring container looks at the properties of the beans on which autowire attribute is set to byType in the XML configuration file. It then tries to match and wire a property if its type matches with exactly one of the beans name in configuration file. If more than one such beans exists, a fatal exception is thrown.
        - **constructor** - Similar to byType, but type applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised.
        - **autodetect** - Spring first tries to wire using autowire by constructor, if it does not work, Spring tries to autowire by byType.
    - **Limitations:**
        - Overriding possibility - You can still specify dependencies using <constructor-arg> and <property> settings which will always override autowiring.

        - Primitive data types - You cannot autowire so-called simple properties such as primitives, Strings, and Classes.
        - Confusing nature - Autowiring is less exact than explicit wiring, so if possible prefer using explict wiring.

**Setter injection**

- The <property> subelement of <bean> is used for setter injection.
- 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.
- The basic-ideas is that you have a no argument-constructor which creates the object with “reasonable-defaults”. The user of the object can then call setters on the object to override the collaborators of the object in order to wire the object graph together or to replace the key collaborators with test-doubles.
- **Advantages:**
    - Setter injection works well with optional dependencies. If you do not need the dependency, then just do not call the setter.
    - You can call the setter multiple times. This is particularly useful if the method adds the dependency to a collection. You can then have a variable number of dependencies.
- **Disadvantages:**
    - The setter can be called more than just at the time of construction so you cannot be sure the dependency is not replaced during the lifetime of the object (except by explicitly writing the setter method to check if it has already been called).
    - You cannot be sure the setter will be called and so you need to add checks that any required dependencies are injected.

**Constructor injection**

- Constructor-based DI is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on other class.
- The basic idea with constructor-injection is that the object has no defaults and instead you have a single constructor where all of the collaborators and values need to be supplied before you can instantiate the object.
- **Advantages:**
    - If the dependency is a requirement and the class cannot work without it then injecting it via the constructor ensures it is present when the class is used as the class cannot be constructed without it.
    - The constructor is only ever called once when the object is created, so you can be sure that the dependency will not change during the object's lifetime.
- **Disadvantages:**
    - Constructor injection is not suitable for working with optional dependencies. It is also more difficult to use in combination with class hierarchies: if a class uses constructor injection then extending it and overriding the constructor becomes problematic.

**Setter vs Constructor Injection**

- Setter injection in Spring uses setter methods like setDependency() to inject dependency on any bean managed by Spring's IOC container. On the other hand constructor injection uses constructor to inject dependency on any Spring-managed bean.
- Setter injection does not ensures dependency Injection. You can not guarantee that certain dependency is injected or not, which means you may have an object with incomplete dependency. On other hand constructor Injection does not allow you to construct object, until your dependencies are ready.
- One more drawback of setter Injection is Security. By using setter injection, you can override certain dependency which is not possible which is not possible with constructor injection because every time you call the constructor, a new object is gets created.

- Constructor-injection enforces the order of initialization and prevents circular dependencies. With setter-injection it is not clear in which order things need to be instantiated and when the wiring is done.

**Beans**

- A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. These beans are created with the configuration metadata that you supply to the container, for example, in the form of XML <bean/>
- **@Bean** is a method-level annotation and a direct analog of the XML <bean/> element.
- To declare a bean, simply annotate a method with the @Bean annotation.
- When JavaConfig encounters such a method, it will execute that method and register the return value as a bean within a BeanFactory.
- By default, the bean name will be the same as the method name
- Java classes that satisfy certain criteria
    - Have getter and setter methods for the properties
    - Setter method names follow a specific format

2. What is the purpose of servletContext.xml file?

    is the Spring Web Application Context Configuration. It's for configuring your Spring beans in a web application. If you use a root-context.xml file, you should put non-web beans in it, and web beans in servletContext.xml.

4. Write a unit test for a given Spring service method
5. Use constructor injection to setup a Spring controller with a Spring bean
6. Use setter injection to setup a Spring controller with a Spring bean

# REST and RESTEasy

What is it?

- Representational State Transfer
- Used to manipulate and manage resources through state representations:
- Architectural Principles
    - **Addressability**
        - Every resource is reachable through a unique identifier
        - Ex. .../departments/cs/courses/378
    - **Uniform constrained Interface**
        - Finite set of actions that make it easy to use and universal
        - CRUD operations: POST (create), GET(read), PUT(update), DELETE, PATCH
        - Idempotent sequences: single execution of the entire sequence always yields a result that is not changed by a re-execution.
            - Ex. <GET GET GET ...>, <PUT>, <DELETE>, or <DELETE DELETE>
    - **Representation Oriented**
        - Addressable through specific URI
    - **Stateless Communication**
        - No client session data is stored on server.
        - Server only records and manages state of exposed resources.
        - If session data is required, use cookies, URLRewriting, or body parameters.
    - **Hypermedia as the Engine of Application State(HATEOS)**
        - One resource points to other resources (links)
        - Advantages: Location transparency (change links without clients knowing) and logical name to a state transition.

**Uniform Interface**

- The uniform interface that any REST services must provide is fundamental to its design.
- Its constraint defines the interface between clients and servers. The four guiding principles of the uniform interface are:
    - **Resource-Based.** Individual resources are defined in requests using URIs as resource identifiers and are separate from the responses that are returned to the client.
    - **Manipulation of Resources Through Representations.** When a client gets a representation of a resource, including any metadata attached, it has enough information to customize or delete the resource on the server, if it has permission to do so.
    - **Self-descriptive Messages.** Each message includes a precise information that describes how to process it. The responses also clearly indicate their cache-ability.
    - **Hypermedia as the Engine of Application State (HATEOAS).** Clients deliver the state via body contents, query-string parameters, request headers and the requested URI. Services deliver state to clients via body content, response codes, and response headers.

**Idempotence**

From a RESTful service standpoint, for an operation (or service call) to be idempotent, clients can make that same call repeatedly while producing the same result. In other words, making multiple identical requests has the same effect as making a single request. Note that while idempotent operations produce the same result on the server (no side effects), the response itself may not be the same (e.g. a resource's state may change between requests).

The PUT and DELETE methods are defined to be idempotent. However, there is a caveat on DELETE. The problem with DELETE, which if successful would normally return a 200 (OK) or 204 (No Content), will often return a 404 (Not Found) on subsequent calls, unless the service is configured to "mark" resources for deletion without actually deleting

them. However, when the service actually deletes the resource, the next call will not find the resource to delete it and return a 404. However, the state on the server is the same after each DELETE call, but the response is different.

GET, HEAD, OPTIONS and TRACE methods are defined as safe, meaning they are only intended for retrieving data. This makes them idempotent as well since multiple, identical requests will behave the same.

| **HTTP Method** | **Idempotent** | **Safe** |
|---|---|---|
| OPTIONS | yes | yes |
| GET | yes | yes |
| HEAD | yes | yes |
| PUT | yes | no |
| POST | no | no |
| DELETE | yes | no |
| PATCH | no | no |

**Steps to design a REST API**

Step 1: Identify potential resources in the system
- Look for nouns (course, department, instructor, ...)

Step 2: Map actions to the identified resources
- Map GET and POST to resources, etc.

Step 3: Identify actions that cannot be mapped to resources
- Prevent mapping GET to get all courses.

Step 4: Refine resource definitions by adding new resources in the resource model if required
- Introduce courses and departments (plural versions).

Step 5: Introduce resource representations
-

Step 6: Re-map actions by establishing resource hierarchies
- Map GET to get all courses, departments, etc.

Step 7: Revisit above steps if required

**JAX-RS**

- **Types:**
  - Jersey, implementation by SUN
  - RESTEasy, from Redhat Jboss
  - Restlet
  - Spring built-in support
- **Concepts**
  - **Resources**
    - Implemented as **resource class** and requests are handled by **resource methods**
    - **Resource Class**
      - Java class that uses JAX-RS annotations to implement a web resource.
      - Must be annotated with @Path or a request method designator (@GET, @POST, etc.).
      - A resource class instance is created for every request.
    - **Resource Constructors**
      - Instantiated on runtime (public constructor)
      - May include parameters like @Context, @Header-Param, @CookieParam, @MatrixParam, @QueryParam, @PathParam.
  - **Resource Methods**

        - Methods of a resource class annotated with a request method designator (@GET, @POST, etc.)
        - Allows annotated parameters with optional default value.
        - May return void, Response, Java type, GenericEntity.
    - **Annotations**
        - @Path
            - Path grows from previous @Path parameters and @PathParam links to the final @Path
        - @Consumes
        - @Produces
        - @MatrixParam("author") String author
            - Allows the assignment of parameters as: "/books/2011;author=mkyong"
        - @FormParam("name") String name
            - Binds HTML form parameters to values
    - **Application Class**
        - The javax.ws.rs.core.Application class is a standard JAX-RS class that you may implement to provide information on your deployment. It is simply a class the lists all JAX-RS root resources and providers.
        - Tells the application server which JAX-RS components to register with JAX-RS runtime.
        - Has 2 methods:
            - Public Set<Class<?>> get Classes()
                - Get a set of root resource classes.
            - Public Set<Object> getSingletones()
                - Get a set of objects that are singletons within the applications, and are shared across requests.
- **JAX-RS Applications**
    - Packaged as a servlet in a .war file
    - Application subclass and resource classes packaged in WEB-INF/classes
    - Required libraries packaged WEB-INF/lib
    - Using JAX-RS aware servlet
        - Servlet-class element of web.xml should name the application-supplied subclass of Application class.
    - Using non-JAX-RS aware servlet
        - Servlet-class element of web.xml should name the JAX-RS implementation-supplied Servlet class.

**Spring vs. JAX-RS**

| - | Spring | JAX-RS |
|---|---|---|
| Annotations | @RequestMapping, @Controller, @Component, @PathVariable, @RequestHeader, @RequestBody, @ResponseBody, etc.<br>- Tied to spring framework<br>- supports"user-in-the-loop" model | @Path, @PathParam, @MatrixParam, @HeaderParam, @FormParam, @CookieParam, etc.<br>- Part of a specification<br>- Portable across JAX-RS implementations. |

**Advantages:**

- Cleaner web API, easier to use by others, and allows scalability.
- Allows data synchronization.

**Disadvantages:**

- Certain resource actions could not map easily to CRUD.
- Cannot perform multiple actions in one request

# Unit Testing

**What is it?**

- Software development process in which a program is broken down into the smallest testable “unit” and tested. These are usually functions or methods.

**Why do we use it?**

- Automates testing

**How does it work?**

- Using junit we set up individual tests that operate using mockito to set up the environment and asserts to verify the results.
- Steps for unit testing:
    1. Identity “code under test”
    2. Identify the dependencies
    3. Create mock dependencies
    4. Set expectations
    5. Invoke the “code under test”
    6. Assert output is as expected
    7. Verify that certain methods were called/not called

**Advantages:**

- Finding bugs is easier; Broken down segments allows for more scrutinized checks
- Facilitates refactoring.
- Can be used as a “living documentation”

**Disadvantages:**

- Cannot catch integration problems or broader design problems.
- Difficult to set up realistic tests.

1. Given a method definition, identify all its dependencies from unit testing point of view
    -
2. Given a method definition, write unit test(s) for it

```
@Before
public void setup() {
        mockJsoupHandler = mock(JSoupHandlerService.class);
        when(mockJsoupHandler.getElements(any(String.class))).thenCallRealMethod();
        openStackMeetings = spy(new OpenStackMeetingsImpl(mockJsoupHandler));
}

@Test // 1. Test when project and year are valid
public void testCalculateWhenBothOperandsNull() {
        String ret = openStackMeetings.getMeetingsCounts("heat", 2016);
        assertTrue(ret.contains("Number of meeting files: 145"));
}
```

3. Given a piece of code, refactor it to enable writing of unit tests for it
    - Follow the steps given above.

# Functional testing

**What is it?**

- Known as “black-box testing”, it only checks that the final output of a program is correct. Doesn’t care for smaller units or functions.

**Why do we use it?**

- Allows

**How does it work?**

- Design and implement your REST Service
- Create tests that programmatically exercise the API
- For each test:
    - Save the response
    - Write assertions

**Advantages:**

- Changes in architecture doesn’t affect the tests.

**Disadvantages:**

- Bugs are harder to identify

**What is difference between unit testing and functional testing?**

- Unit testing is concerned with testing individual layers of your application.
- Functional testing is concerned with testing whether the entire application is working as expected or not.
- Unit testing involves mocking out the dependencies of the code under test.
- Functional tests do not involve mocking.

Write a functional test for a particular REST resource

```
@Test
public void nonExistentProjectTest() throws Exception{
        Output output = new Output();
        output.setError("Project non-existent-project does not exist");

        URL getUrl = new
             URL("http://localhost:8080/assignment3/myeavesdrop/projects/non-existent-project/meetings");
        JAXBContext jc = JAXBContext.newInstance(Output.class);
        Unmarshaller u = jc.createUnmarshaller();
        Object o = u.unmarshal(getUrl);
        Assert.assertEquals(output.getError(), ((Output) o).getError());
}
```

# Marshalling/Unmarshalling

**What is it?**

- Converting data into objects

**Why do we use it?**

- Helps us convert incoming XML documents into usable java class objects

**How does it work?**

- You add anchor text to the class so that JAXB can link XML elements with their respective variables within the class.

```
@XmlRootElement(name = "projects")
@XmlAccessorType(XmlAccessType.FIELD)
public class Projects {

    private List<String> project = new ArrayList<String>();

    public List<String> getProjects() {
        return project;
    }

    public void setProjects(List<String> projects) {
        this.project = project;
    }

    public void addProject(String p) {
        this.project.add(p);
    }
}
```

**Advantages:**

- Access data non sequentially.
- Memory management is more efficient that DOM trees.

**Disadvantages:**

- None found, people literally call it the best there is.

**Marshalling (aka Serializing)**

- Converting Java objects to JSON/XML
- is the process of translating data structures or object state into a format that can be stored and reconstructed later in the same or another computer environment.
- Serialization of object-oriented objects does not include any of their associated methods with which they were previously inextricably linked.

**Unmarshalling (aka Deserializing)**

- Converting JSON/XML strings to Java objects
- extracting a data structure from a series of bytes

## XML/HTML Parsing

**What is it?**

- XML is a “generic” markup language for data
- HTML is a markup language for browser display

**XML Parsers**

- **Tree(DOM) parsing:**
  - Def.
  - Advantages:
    - Provides fine grained control over parsing.
  - Disadvantages:
    - Entire tree must be built in memory before parsing can begin.
- **Callback (SAX) parsing:**
  - Def.
  - Advantages:
    - No memory issues.
  - Disadvantages:
    - State between callback invocations needs to be maintained by the program.
- **XPath parsing (current):**
  - Declarative model for querying XML documents

**HTML Parsers**

- Java RegexParser
- Jsoup
- Disadvantages
  - Parsing presentation logic instead of working with domain objects.
  - Brittle, will break if the HTML page is changed
  - No formal contract defined, unable to validate

XML/HTML Parsing Given a XML/HTML response, write code to parse it using <a_parsing_method>

# Logging

# JDBC

# Hibernate

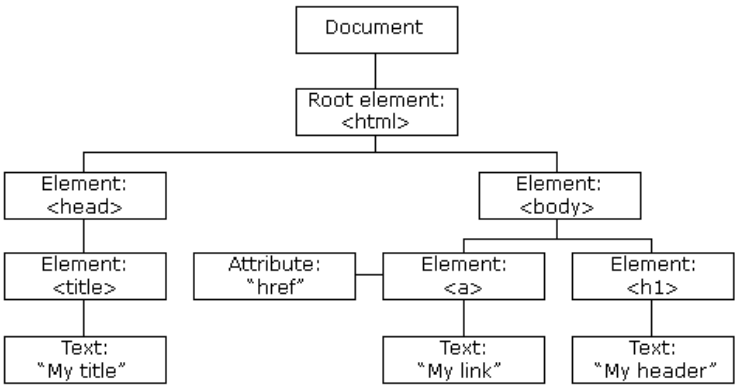
- How to get Hibernate to Create a Table
    - So the key to getting Hibernate to create a table for us is to understand how Hibernate makes the decision to create a table. It works by scanning a particular package that you've specified in your configuration file. Hibernate will scan that package for any Java objects annotated with the @Entity annotation. If it finds any, then it will begin the process of looking through that particular Java object to recreate it as a table in your database!
    - We still need to specify a couple of things:
        - The primary key for the table
        - How to generate the primary keys
    - For both items above, we'll use annotations.
    - The first annotation (@Id) is simple and it's used to "mark" the primary key for the table, the second annotation is @GeneratedValue and it will be used to specify HOW to generate primary keys.
    -
```
@Table(name="address_book")
@Entity
public class AddressBook
{
  private Long id;
  private String name;

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
        public Long getId()
        {
                return id;
        }
        public void setId(Long id)
        {
                this.id = id;
        }
        public String getName()
        {
                return name;
        }
        public void setName(String name)
        {
                this.name = name;
        }
```
    - The @Entity annotation is used at the class level, but the @Id and @GeneratedValue annotations are used at the method level.
    - It's important to note the exact location of the @Id and @GeneratedValue annotations: just above the getter method for the id instance variable. If you were to put these annotations on the setter method, it won't work as Hibernate only looks at the getter methods for these annotations.

- Hibernate detects that the @Id annotation is on a field and assumes that it should access properties on an object directly through fields at runtime. If you placed the @Id annotation on the getId() method, you would enable access to properties through getter and setter methods by default. Hence, all other annotations are also placed on either fields or getter methods, following the selected strategy. Following section will explain the annotations used in the above class.
    - @Entity Annotation:
        - marks the class as an entity bean, so it must have a no-argument constructor that is visible with at least protected scope.
    - @Table Annotation:

        - allows you to specify the details of the table that will be used to persist the entity in the database.
        - provides four attributes, allowing you to override the name of the table, its catalogue, and its schema, and enforce unique constraints on columns in the table.
    - @Id and @GeneratedValue Annotations:
        - Each entity bean will have a primary key, which you annotate on the class with the @Id annotation. The primary key can be a single field or a combination of multiple fields depending on your table structure.
        - By default, the @Id annotation will automatically determine the most appropriate primary key generation strategy to be used but you can override this by applying the @GeneratedValue annotation which takes two parameters strategy and generator which I'm not going to discuss here, so let us use only default the default key generation strategy. Letting Hibernate determine which generator type to use makes your code portable between different databases.
    - @Column Annotation:
        - used to specify the details of the column to which a field or property will be mapped. You can use column annotation with the following most commonly used attributes:
            - name attribute permits the name of the column to be explicitly specified.
            - length attribute permits the size of the column used to map a value particularly for a String value.
            - nullable attribute permits the column to be marked NOT NULL when the schema is generated.
            - unique attribute permits the column to be marked as containing only unique values.

# Javascript

- AngularJS, ReactJS, Bootstrap, JQuery
- What is the event bubbling model of JavaScript?
    - Event bubbling and capturing are two ways of event propagation in the HTML DOM API, when an event occurs in an element inside another element, and both elements have registered a handle for that event. The event propagation mode determines in which order the elements receive the event.
    - With **bubbling**, the event is first captured and handled by the innermost element and then propagated to outer elements.
    - With **capturing**, the event is first captured by the outermost element and propagated to the inner elements.
- **DOM**

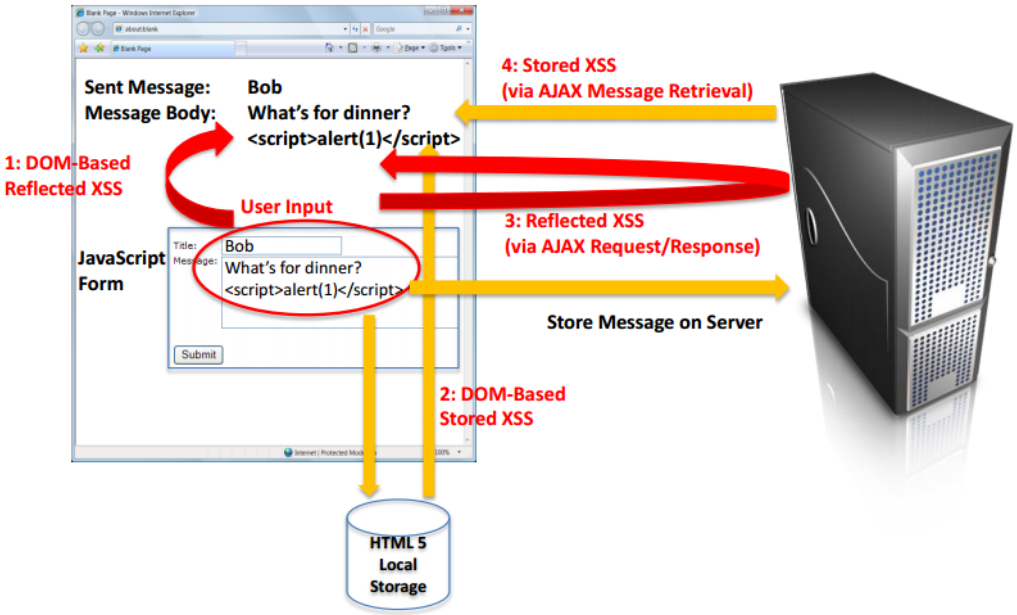


- # Same Origin Policy
  - ## What is the same origin policy
  - The same-origin policy restricts how a document or script loaded from one origin can interact with a resource from another origin. Under the policy, a web browser permits scripts contained in a first web page to access data in a second web page, but only if both web pages have the same origin. An origin is defined as a combination of URI scheme, hostname, and port number. This policy prevents a malicious script on one page from obtaining access to sensitive data on another web page through that page's Document Object Model.
  - ## Identify whether the specified resources have the same origin?
  - Two pages have the same origin if the protocol, port (if one is specified), and host are the same for both pages.

| Compared URL | Outcome | Reason |
|---|---|---|
| **http://www.example.com**/dir/page2.html | Success | Same protocol, host and port |
| **http://www.example.com**/dir2/other.html | Success | Same protocol, host and port |
| **http://**username:password**@www.example.com**/dir2/other.html | Success | Same protocol, host and port |
| http://www.example.com:**81**/dir/other.html | Failure | Same protocol and host but different port |
| **https**://www.example.com/dir/other.html | Failure | Different protocol |
| http://**en.example.com**/dir/other.html | Failure | Different host |
| http://**example.com**/dir/other.html | Failure | Different host (exact match required) |
| http://**v2.www.example.com**/dir/other.html | Failure | Different host (exact match required) |
| http://www.example.com:**80**/dir/other.html | Depends | Port explicit. Depends on implementation in browser. |

- ## http://stackoverflow.com/questions/929677/how-exactly-is-the-same-domain-policy-enforced
- **Let's say I have a domain, js.mydomain.com, and it points to some IP address, and some other domain, requests.mydomain.com, which points to a different IP address. Can a .js file downloaded from js.mydomain.com make Ajax requests to requests.mydomain.com?**
- **How exactly do modern browsers enforce the same-domain policy?**
- The short answer to your question is no: for AJAX calls, you can only access the same hostname (and port / scheme) as your page was loaded from.
- There are a couple of work-arounds: one is to create a URL in foo.example.com that acts as a reverse proxy for bar.example.com. The browser doesn't care where the request is actually fulfilled, as long as the hostname matches. If you already have a front-end Apache webserver, this won't be too difficult.

- # Cross-site scripting
  - **Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into** otherwise benign and trusted web sites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it.
  - An attacker can use XSS to send a malicious script to an unsuspecting user. The end user’s browser has no way to know that the script should not be trusted, and will execute the script. Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by the browser and used with that site. These scripts can even rewrite the content of the HTML page.




- **TYPES:**
  - **Stored XSS (Persistent)** - occurs when user input is stored on the target server, such as in a database, in a message forum, visitor log, comment field, etc. And then a victim is able to retrieve the stored data from the web application without that data being made safe to render in the browser
  - **Reflected XSS (non-Persistent)** - occurs when user input is immediately returned by a web application in an error message, search result, or any other response that includes some or all of the input provided by the user as part of the request, without that data being made safe to render in the browser, and without permanently storing the user provided data. In some cases, the user provided data may never even leave the browser
  - **Server XSS** - occurs when untrusted user supplied data is included in an HTML response generated by the server. The source of this data could be from the request, or from a stored location. As such, you can have both Reflected Server XSS and Stored Server XSS. In this case, the entire vulnerability is in server-side code, and the browser is simply rendering the response and executing any valid script embedded in it.
  - **Client XSS -** occurs when untrusted user supplied data is used to update the DOM with an unsafe JavaScript call. A JavaScript call is considered unsafe if it can be used to introduce valid JavaScript into the DOM. This source of this data could be from the DOM, or it could have been sent by the server (via an AJAX call, or a page load). The ultimate source of the data could have been from a request, or from a stored location on the client or the server. As such, you can have both Reflected Client XSS and Stored Client XSS.

| | Where untrusted data is used | | |
|---|---|---|---|
| Data Persistence | **XSS** | **Server** | **Client** |
| | Stored | Stored Server XSS | Stored Client XSS |
| | Reflected | Reflected Server XSS | Reflected Client XSS |

- ❑ DOM Based XSS is a subset of Client XSS (where the data source is from the DOM only)
- ❑ Stored vs. Reflected only affects the likelihood of successful attack, not the nature of vulnerability or the most effective defense

- **DEFENSES:**
  - **Server XSS** - The easiest and strongest defense against Server XSS in most cases is **Context-sensitive server side output encoding**. Input validation or data sanitization can also be performed to help prevent Server XSS, but it's much more difficult to get correct than context-sensitive output encoding.
  - **Client XSS** - The easiest and strongest defense against Client XSS is **Using safe JavaScript APIs**
  -

# ETL (Extract, Transfer, Load)

- Used by: XML feeds, RESTful API, databases, and websites.
- Issues:
  - Continually updating source
    - Need to periodically poll; maintain "last-queried-pointer"
  - Load from different sources• Merge data
    - Concurrency
  - Upsert
    - Insert or Update
  - Target data representation• Transformations• High throughput
    - Using JDBC vs Hibernate (Object/Relational Mapping)
- **What is it?**
  - ETL is short for extract, transform, load, three database functions that are combined into one tool to pull data out of one database and place it into another database.
  - **Extract** is the process of reading data from a database.
  - **Transform** is the process of converting the extracted data from its previous form into the form it needs to be in so that it can be placed into another database. Transformation occurs by using rules or lookup tables or by combining the data with other data.
  - **Load** is the process of writing the data into the target database.
- **Why care?**
  - ETL is used to migrate data from one database to another, to form data marts and data warehouses and also to convert databases from one format or type to another.
  - ETL saves significant time on data extraction and preparation - time that could be better spent on extracting insights.ETL saves significant time on data extraction and preparation - time that could be better spent on extracting insights.ETL saves significant time on data extraction and preparation - time that could be better spent on extracting insights.
-

# Cloud computing

- On-demand provisioning and management of resources
- Resources
  – Compute servers, database instances, object stores, load balancers
    • Compute servers can be virtual machines and/or dedicated hardware
- Provisioning and management
  – creation, scaling, deletion
- On demand
  – When needed by application developer
  – When needed by running application

**Infrastructure-as-a-Service (IaaS)**

• In this model you provision a VM, ssh into it, install application container such as Tomcat, deploy your code

**– Advantage:**

• As an application developer you have complete control over how to setup your application's environment – Disadvantage:

• You have to deal with issues outside of core application development

**– Setting up Tomcat**

**– Modifying 2rewall rules**

**– Examples:**

• Commercial

– Amazon AWS, Google Compute Engine (GCE), Microsoft Azure, IBM Softlayer, Rackspace cloud servers

• Open source

– OpenStack, CloudStack

**Platform-as-a-Service (PaaS)**

• In this model you develop your code locally and push it to the "platform". The platform takes care of deploying it to an appropriate application container

**– Advantage:**

• You just need to worry about developing your application's code • Appealing model from developers' point of view

**– Disadvantage:**

• Hard to troubleshoot since no access to the VM or the application container

**– Typically, the platforms make logs available via an API**

**– Examples**

• Commercial

– Heroku, Amazon Elastic Beanstalk, Google App Engine (GAE), Redhat OpenShift, Pivotal CloudFoundry, IBM BlueMix

• Open source

– OpenStack Solum, Redhat OpenShift, Pivotal CloudFoundry

# Languages/frameworks/libraries

- Python
  - Django, Flask
- Ruby
  - Rails
- Java
  - Groovy, Scala
- PHP
- C#
- JavaScript