**Week 4 – Day1**

**Overview of HTML**

Introduction to HTML

* [HTML](https://en.wikipedia.org/wiki/HTML) stands for Hypertext Markup Language - it is a markup language for creating web pages and applications.
* HTML contains a particular syntax - namely **elements** and **attributes** - that web browsers parse in order to render the content of the webpage.
* With HTML, the structure and content of a webpage is defined. Styling and dynamic behavior are introduced with CSS and JavaScript, respectively.

HTML5

HTML5 introduced a new **DOCTYPE** declaration <!DOCTYPE html> and the character encoding (charset) declaration <meta charset="UTF-8">. The <DOCTYPE> declaration is used to inform the browser about the version of HTML used in the document. It is known as the Document Type Declaration (DTD). It just instructs the browser about the document type. A **character encoding** is an approach of converting bytes into characters. For validating the HTML document, a program must choose a character encoding.

HTML5 also introduced features to allow us to embed audio and video files on the web page and provides the support to run JavaScript in the background.

Take a look at the structure of the HTML5 file given below:

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Title of the document</title>

</head>

<body>

Content of the document......

</body>

</html>

**NOTE:** HTML5 uses UTF-8 as a default character encoding.

<!DOCTYPE html> should be in the beginning of the document before any tags.

**HTML Tags**

# HTML Tags

## Common HTML Tags

There are a vast number of HTML tags you could use on your webpage, but below are listed the most common:

* <div> - defines a "division of the page"
* <p> - defines a paragraph
* <span> - an inline tag for grouping text or elements
* <b> - bold text
* <i> - italicized text
* <h1>, <h2>, ... <h6> - these are headings, h1 is largest and h6 is smallest
* <br> - line break
* <table> - defines a table
* <img src="URL">
* <ol> - an ordered list
* <ul> - an unordered list
* <a href=""> - makes a hyperlink

### **Hyperlinks**

To make a hyperlink in a webpage, use the <a> tag:

<p> Here is a <a href="www.google.com">link to Google!</a></p>

### **Tables**

To create a table, use the following markup. <tr> defines a table row, <td> defines a table cell, and <th> is used for table headers.

<table>

<thead>

<tr>

<th>Id</th>

<th>Name</th>

</tr>

</thead>

<tbody>

<tr>

<td>1</td>

<td>Alice</td>

</tr>

<tr>

<td>2</td>

<td>Bob</td>

</tr>

<tr>

<td>3</td>

<td>Charlie</td>

</tr>

</tbody>

</table>

### **Lists**

There are two options for making lists in HTML - ordered or unordered lists. Ordered lists are defined with <ol>, unordered lists are defined with <ul>, and the list items within either are denoted with <li>:

<ol> <!-- ordered lists render as 1, 2, 3, etc.. -->

<li>First item</li>

<li>Second item</li>

<li>Third item</li>

</ol>

<ul> <!-- list items in here will just be bullet points -->

<li>First item</li>

<li>Second item</li>

<li>Third item</li>

</ul>

# HTML5 Semantic Elements

Semantic Elements defines the meaning for the web page rather than just presentation.

* The HTML elements like <div> and <span> are not releated to the content on the web page. This kind of elements are called as non semantic elements.
* The HTML elements like <form>, <table>, and <article> are used to define the content and on the webpage. This kind of elements are called as semantic elements.

We will disscus about the following semantic elements that helps to define the element's purpose on the webpage:

<section>

<article>

<header>

<footer>

<nav>

<aside>

<figure>

<figcaption>

<details>

<mark>

<summary>

<time>

### **Section**

The HTML5 <section> tag defines a thematic grouping of content. In a document, we have sections like chapters, headers, footers, introduction, content, contact information, etc.

Example

<section>

<h1>Protocol</h1>

<p>A protocol is a standard set of rules that allow electronic devices to communicate with each other.</p>

</section>

### **Article**

The <article> element represents a section of content that forms an independent part of a document or site such as Forum post, Blog post, Newspaper article, etc.,

Example

<article>

<h1>What WHO do?</h1>

<p>WHO works worldwide to promote health, keep the world safe, and serve the vulnerable. Our goal is to ensure that a billion more people have universal health coverage, to protect a billion more people from health emergencies, and provide a further billion people with better health and well-being..</p>

</article>

### **Header**

The <header> element specifies a header for a document or section.

Example

<!-- example defines a header for an article -->

<article>

<header>

<h1>World Health Organisation</h1>

<p>What we do?</p>

</header>

<p>WHO works worldwide to promote health, keep the world safe, and serve the vulnerable. Our goal is to ensure that a billion more people have universal health coverage, to protect a billion more people from health emergencies, and provide a further billion people with better health and well-being..</p>

</article>

### **Footer**

The <footer> element used to define the footer for a document or section. It contains information about the author of the document, copyright information, links to terms of use, contact information, etc.

Example

<footer>

<p>Posted by: someone </p>

<p>Contact information: <a href="mailto:someone@example.com">

someone@example.com</a>.</p>

</footer>

### **Navigation**

The <nav> element is for major navigation blocks that specify a set of navigation links.

Example

<nav>

<a href="/html/">HTML</a> |

<a href="/css/">CSS</a> |

<a href="/js/">JavaScript</a> |

<a href="/jquery/">jQuery</a>

</nav>

### **Aside**

The <aside> element is used to identify content that is related to the primary content of the web page. The content inside the <aside> element does not constitute the primary content of the page. For example, we can have author information, related links, related content, and advertisements.

Example

<p>I went TajMahal for this summer</p>

<aside>

<h4>Taj Mahal</h4>

<p>The Taj Mahal is an ivory-white marble mausoleum on the south bank of the Yamuna river in the Indian city of Agra.</p>

</aside>

### **HTML figure and figcaption Elements**

The <figure> element describes some flow content, optionally with a caption, that is self-contained and referenced as a single unit from the main flow of the document. The <img> and <figcaption> elements are grouped in a <figure> element. We use the <img> element to insert an image on the web page. To add the visual explanation of the image, we need a caption for that image. This can be achieved in the HTML5 by using <figcaption> element.

Example

<figure>

<img src="WorldMap.jpg" alt="WorldMap">

<figcaption>Fig1. - World Map </figcaption>

</figure>

### **Other HTML Elements**

* <details> - Used to add details that user can view or hide
* <mark> - Used to highlight or mark the text.
* <summary> - Defines a obvious heading for a <details> element
* <time> - Used to add a date/time.

# HTML5 Audio tag

The HTML5 <audio> element used to embed audio in a web page.

**Example**

<body>

<h1>The audio element</h1>

<p>Click the Play button:</p>

<audio controls>

<source src="River.ogg" type="audio/ogg">

Invalid audio!!! - Browser doesn't support.

</audio>

</body>

In the above example, we have two tags, <audio> and <source>.

The <audio> element defines sound content and it has a controls attribute that adds audio controls, like play, pause, and volume. Any text within the <audio> and </audio> displayed on the browser only if the audio was not supported by the browser.

The <source> element defines the media resources for the audio files and it has attributes such as src and type. The src is used to specify the file format of the audio content and type specifies the media types that <audio> element supports.

The file format supported by <audio> element and respective media types are tabulated below:

| **File Format** | **Media Type** |
| --- | --- |
| MP3 | audio/mpeg |
| OGG | audio/ogg |
| WAV | audio/wav |

# HTML5 Video tag

The <video> element used to embed a video on a web page, such as a movie clip or other video streams.

**Example**

<body>

<h1>The video element</h1>

<video width="320" height="240" controls>

<source src="movie.mp4" type="video/mp4">

Invalid audio!!! - Browser doesn't support.

</video>

</body>

HTML5 Video Attributes used in the above example are described below:

| **Attribute** | **Value** | **Description** |
| --- | --- | --- |
| src | URL | Specifies the URL of the video file |
| controls | controls | Specifies the video controls such as a play/pause button, etc |
| height | pixels | Sets the height of the video player |
| width | pixels | Sets the width of the video player |

Note: Any text between the <video> and </video> tags will be displayed in browsers that do not support the <video> element.

The MIME Types supported by the <video> element is tabulated below:

| **Format** | **MIME-type** |
| --- | --- |
| MP4 | video/mp4 |
| WebM | video/webm |
| Ogg | video/ogg |

**HTML Document Structure & DOM**

## Starting HTML Documents

Every HTML document (ending with .html extension) should begin with a special tag known as the DOCTYPE declaration - this lets browsers know what kind of document it is using (HTML, in our case) as well as which version of the markup language is being used. For HTML5, the newest version and the one which we'll be using, the DOCTYPE syntax is:

<!DOCTYPE html>

The doctype declaration tag does not have a closing tag and it is not self-closing either.

Next, the tag which begins the root of our HTML document is the <html> tag. Everything about our webpage will be nested within this tag.

Within the html element we have two important tags - the <head> and the <body> tags. The head element will contain all the metadata associated with this page, including the title, character encodingg, and references to external stylesheets. The body element contains the actual content of our page that will be rendered on the screen by the browser.

### **Hello World Example**

Let's write a simple webpage that will show off our knowledge of HTML thus far. Open up a new file in a simple text editor and save the file as hello-world.html. Then write the following HTML markup and save it:

<!DOCTYPE html>

<html>

<head>

<title>Hello World!</title>

<meta charset="utf-8">

</head>

<body>

<div>

<!-- THIS IS A COMMENT! -->

<p>This is my first paragraph written in HTML</p>

</div>

</body>

</html>

The only tag you may be unfamiliar with is the meta tag above - this defines the character encoding that the file will be using. Also, the <!-- ... --> syntax denotes a comment.

If you now open your hello-world.html file in your browser of choice, you'll see your webpage rendered. It may look ugly now, but we'll address that when we talk about CSS.

**Elements and Attributes**

## Elements

HTML is composed of many different **elements** - these provide the structure of the document. Elements are defined within HTML files using **tags** - for example, one very common tag is the <div> tag. The tag is enclosed within angle brackets. Most elements have a closing tag which define the end of the element, using the backslack notation - for example, a closing "div" tag would be </div>. HTML elements may be nested within other elements, like so:

<div> text inside the div but outside the paragraph

<p> this text is inside the paragraph </p>

</div>

In order for HTML to be **valid**, tags must be properly nested - an outer tag cannot be closed before an inner one. For example, the following markup would not be considered valid:

<div> invalid!

<p> cannot close the div here => </div>

</p>

Not all elements have closing tags, some are self-closing. For example, the <img /> tag, which defines an image.

## Attributes

HTML elements can also have **attributes** defined within the tag - these are key/value pairs that give metadata about the tag that are important for the browser to know. For example, image elements must have a URL which the browser can call to retrieve the image file to display on the page - we use the src attribute to do this: <img src="/URL/to/get/cat.png" alt="cool cat!" />. As you may have guessed, the alt attribute specifies an alternative text to show when the image cannot be displayed.

### **Global Attributes**

**Global** attributes are those that can be applied to any element on the page. Some important global attributes are:

* class
* id
* hidden
* lang
* style
* tabindex
* title

There are also many attributes that should be applied to only certain elements, including the src and alt attributes shown above. We'll discuss more about these when relevant.

**Inline and Block Elements**

Inline and Block Elements

Before listing all the HTML elements available to use, it's important to know the difference between inline and block-level elements.

Block-level elements are those that will render on new lines in blocks by default, instead of rendering within the line itself like inline elements do.

One example of a block element is

, and a common inline element is . Try them out on your webpage and notice the difference.

**Common Tags**

# HTML Tags

## Common HTML Tags

There are a vast number of HTML tags you could use on your webpage, but below are listed the most common:

* <div> - defines a "division of the page"
* <p> - defines a paragraph
* <span> - an inline tag for grouping text or elements
* <b> - bold text
* <i> - italicized text
* <h1>, <h2>, ... <h6> - these are headings, h1 is largest and h6 is smallest
* <br> - line break
* <table> - defines a table
* <img src="URL">
* <ol> - an ordered list
* <ul> - an unordered list
* <a href=""> - makes a hyperlink

### **Hyperlinks**

To make a hyperlink in a webpage, use the <a> tag:

<p> Here is a <a href="www.google.com">link to Google!</a></p>

### **Tables**

To create a table, use the following markup. <tr> defines a table row, <td> defines a table cell, and <th> is used for table headers.

<table>

<thead>

<tr>

<th>Id</th>

<th>Name</th>

</tr>

</thead>

<tbody>

<tr>

<td>1</td>

<td>Alice</td>

</tr>

<tr>

<td>2</td>

<td>Bob</td>

</tr>

<tr>

<td>3</td>

<td>Charlie</td>

</tr>

</tbody>

</table>

### **Lists**

There are two options for making lists in HTML - ordered or unordered lists. Ordered lists are defined with <ol>, unordered lists are defined with <ul>, and the list items within either are denoted with <li>:

<ol> <!-- ordered lists render as 1, 2, 3, etc.. -->

<li>First item</li>

<li>Second item</li>

<li>Third item</li>

</ol>

<ul> <!-- list items in here will just be bullet points -->

<li>First item</li>

<li>Second item</li>

<li>Third item</li>

</ul>

# HTML5 Semantic Elements

Semantic Elements defines the meaning for the web page rather than just presentation.

* The HTML elements like <div> and <span> are not releated to the content on the web page. This kind of elements are called as non semantic elements.
* The HTML elements like <form>, <table>, and <article> are used to define the content and on the webpage. This kind of elements are called as semantic elements.

We will disscus about the following semantic elements that helps to define the element's purpose on the webpage:

<section>

<article>

<header>

<footer>

<nav>

<aside>

<figure>

<figcaption>

<details>

<mark>

<summary>

<time>

### **Section**

The HTML5 <section> tag defines a thematic grouping of content. In a document, we have sections like chapters, headers, footers, introduction, content, contact information, etc.

Example

<section>

<h1>Protocol</h1>

<p>A protocol is a standard set of rules that allow electronic devices to communicate with each other.</p>

</section>

### **Article**

The <article> element represents a section of content that forms an independent part of a document or site such as Forum post, Blog post, Newspaper article, etc.,

Example

<article>

<h1>What WHO do?</h1>

<p>WHO works worldwide to promote health, keep the world safe, and serve the vulnerable. Our goal is to ensure that a billion more people have universal health coverage, to protect a billion more people from health emergencies, and provide a further billion people with better health and well-being..</p>

</article>

### **Header**

The <header> element specifies a header for a document or section.

Example

<!-- example defines a header for an article -->

<article>

<header>

<h1>World Health Organisation</h1>

<p>What we do?</p>

</header>

<p>WHO works worldwide to promote health, keep the world safe, and serve the vulnerable. Our goal is to ensure that a billion more people have universal health coverage, to protect a billion more people from health emergencies, and provide a further billion people with better health and well-being..</p>

</article>

### **Footer**

The <footer> element used to define the footer for a document or section. It contains information about the author of the document, copyright information, links to terms of use, contact information, etc.

Example

<footer>

<p>Posted by: someone </p>

<p>Contact information: <a href="mailto:someone@example.com">

someone@example.com</a>.</p>

</footer>

### **Navigation**

The <nav> element is for major navigation blocks that specify a set of navigation links.

Example

<nav>

<a href="/html/">HTML</a> |

<a href="/css/">CSS</a> |

<a href="/js/">JavaScript</a> |

<a href="/jquery/">jQuery</a>

</nav>

### **Aside**

The <aside> element is used to identify content that is related to the primary content of the web page. The content inside the <aside> element does not constitute the primary content of the page. For example, we can have author information, related links, related content, and advertisements.

Example

<p>I went TajMahal for this summer</p>

<aside>

<h4>Taj Mahal</h4>

<p>The Taj Mahal is an ivory-white marble mausoleum on the south bank of the Yamuna river in the Indian city of Agra.</p>

</aside>

### **HTML figure and figcaption Elements**

The <figure> element describes some flow content, optionally with a caption, that is self-contained and referenced as a single unit from the main flow of the document. The <img> and <figcaption> elements are grouped in a <figure> element. We use the <img> element to insert an image on the web page. To add the visual explanation of the image, we need a caption for that image. This can be achieved in the HTML5 by using <figcaption> element.

Example

<figure>

<img src="WorldMap.jpg" alt="WorldMap">

<figcaption>Fig1. - World Map </figcaption>

</figure>

### **Other HTML Elements**

* <details> - Used to add details that user can view or hide
* <mark> - Used to highlight or mark the text.
* <summary> - Defines a obvious heading for a <details> element
* <time> - Used to add a date/time.

# HTML5 Audio tag

The HTML5 <audio> element used to embed audio in a web page.

**Example**

<body>

<h1>The audio element</h1>

<p>Click the Play button:</p>

<audio controls>

<source src="River.ogg" type="audio/ogg">

Invalid audio!!! - Browser doesn't support.

</audio>

</body>

In the above example, we have two tags, <audio> and <source>.

The <audio> element defines sound content and it has a controls attribute that adds audio controls, like play, pause, and volume. Any text within the <audio> and </audio> displayed on the browser only if the audio was not supported by the browser.

The <source> element defines the media resources for the audio files and it has attributes such as src and type. The src is used to specify the file format of the audio content and type specifies the media types that <audio> element supports.

The file format supported by <audio> element and respective media types are tabulated below:

| **File Format** | **Media Type** |
| --- | --- |
| MP3 | audio/mpeg |
| OGG | audio/ogg |
| WAV | audio/wav |

# HTML5 Video tag

The <video> element used to embed a video on a web page, such as a movie clip or other video streams.

**Example**

<body>

<h1>The video element</h1>

<video width="320" height="240" controls>

<source src="movie.mp4" type="video/mp4">

Invalid audio!!! - Browser doesn't support.

</video>

</body>

HTML5 Video Attributes used in the above example are described below:

| **Attribute** | **Value** | **Description** |
| --- | --- | --- |
| src | URL | Specifies the URL of the video file |
| controls | controls | Specifies the video controls such as a play/pause button, etc |
| height | pixels | Sets the height of the video player |
| width | pixels | Sets the width of the video player |

Note: Any text between the <video> and </video> tags will be displayed in browsers that do not support the <video> element.

The MIME Types supported by the <video> element is tabulated below:

| **Format** | **MIME-type** |
| --- | --- |
| MP4 | video/mp4 |
| WebM | video/webm |
| Ogg | video/ogg |

**Form Elements and Its Attributes**

# HTML Forms

An **HTML form** is a section of a document that contains controls such as text fields, password fields, checkboxes, radio buttons, submit button, menus, etc. Using these elements the page can collect information from a user which is typically submitted to a web server. To create a form, you would use the <form> tag.

Why use an HTML Form?

* We use forms to collect some information/data form the user.
  + For example: If a user wants to purchase some items on the internet, he or she must fill out the form which will collect information such as the shipping address and payment details so that the item can be sent to the given address.

# Attributes Used in HTML Forms

There are several attributes that you can use on the <form> tag and on <input> elements. We will cover:

* action
* target
* name
* method
* value
* placeholder
* required

Take a look at the following example and find the different attributes:

<!DOCTYPE html>

<html>

<body>

<form action="/test.php" target="\_blank" method="GET">

Username:<br />

<input type ="text" name="username" placeholder="Username" required/>

<br />

Password:<br />

<input type ="password" name="password" />

<br /><br />

<input type ="submit" value ="Submit" />

</form>

</body>

</html>

### **The Action Attribute**

The action attribute indicates where the form data will be processed. Typically the value is a URL of a server. Generally, the form data is sent to a webpage on the webserver after the user clicks on the submit button.

In the above code, after clicking on the submit button, the form data would be sent to a page called test.php.

### **The Target Attribute**

The Target attribute is used to specify whether the submitted result will open in the current window, a new tab or on a new frame. The default value used is "self" which results in the form submission in the same window. To make the result display in a new browser tab, the value should be set to "blank".

In the above code, after clicking on the submit button, the result will open in a new browser tab. Most often this attribute is not present and the default value of "self" is used.

### **Name Attribute**

The name attribute should be provided for each input element. It is not required, but the value provides a label for the data once the form is submitted. If the name attribute is not specified in an input field then the data of that field will not be sent.

In the above code, after clicking the submit button, the form data will be sent to a page called /test.php. The data sent will include the username and password fields.

### **The Method Attribute**

The method attribute is used to specify the HTTP method used to send data while submitting the form. There are only two options available: GET and POST.

**GET** - When using the GET method, after the submission of the form, the form values will be visible in the address bar of the browser.

**POST** – When using the POST method, after the submission of the form, the form values will NOT be visible in the address bar of the browser.

### **The value Attribute**

The value attribute specifies an initial value for an input field. It also serves as the attribute to use when providing a button label for submit and reset input elements.

In the above example, there are no default values.

### **The placeholder Attribute**

The placeholder attribute specifies a hint that describes the expected value of the input field (a sample value or a short description of the expected format). The short hint is displayed in the input field before the user enters a value. The placeholder attribute works with the following input types: text, search, url, tel, email, and password.

In the above example, the text field has a placeholder of "Username".

### **The required Attribute**

The required attribute indicates an input field that must be filled out before submitting the form. In most modern browsers, it will prevent the user from submitting the form until an acceptable value is entered. The required attribute works with the following input types: text, search, url, tel, email, password, date pickers, number, checkbox, radio, and file.

In the above example, only the text field is required.  
**NOTE**: The required attribute doesn't have a value portion. You only need to specify the word 'required'.

### **The min and max Attributes**

The min and max attributes specify the minimum and maximum values for an input field. The min and max attributes work with the following input types: number, range, date, datetime-local, month, time and week.

**Tip**: Use the max and min attributes together to create a range of legal values. (For example: Set a maximum date or a minimum date)

Example

<form>

<label for="datemax">Enter a date before 1980-01-01:</label>

<input type="date" id="datemax" name="datemax" max="1979-12-31"><br><br>

<label for="datemin">Enter a date after 2000-01-01:</label>

<input type="date" id="datemin" name="datemin" min="2000-01-02"><br><br>

<label for="quantity">Quantity (between 1 and 5):</label>

<input type="number" id="quantity" name="quantity" min="1" max="5">

</form>>

**Input Elements and Input Types**

Input Element in HTML Forms

An HTML form collects information from **** elements. You will specify an addition **type** attribute to indicate which field to display. Various fields can be created such as a text field, checkbox, password field, or radio button.

Text Field

A text field is a one-line input field that allows the user to input a line of text. Text Fields are created by specifying the type attribute value as "text".

The below example will display a text field with the label *Email Id:*

**Example**

<form>

<label for="email-input">Email Id:</label><br>

<input type="text" name="email-input" />

</form>

Password Field

Password fields are a type of text field in which the text entered is masked using asterisk or dots. This prevents others form viewing the screen to see what is typed in. Also, its created by specifying the type attribute value as "password".

**Example**

<form>

<label for="user-password">Password:</label><br/>

<input type="password" name="user-password"/>

</form>

**NOTE**: Although a password field is masked, it is NOT encrypted. You will have to use other measures such as HTTPS to encrypt data once the HTML form is submitted.

Radio Buttons

Radio Buttons are used to let the user select exactly one value from a list of predefined options. It is created by specifying the type attribute value as "radio".

**Example**

<form>

SELECT GENDER

<br/>

<input type="radio" name="gender" id="male">

<label for="male">Male</label><br>

<input type="radio" name="gender" id="female">

<label for="female">Female</label>

</form>

**NOTE**: A form may have multiple sets of radio buttons. In order to make sure the user only selects one option from a given set, each radio  element must have matching name attributes. In the example above, both buttons have a name attribute value as **gender**.

Checkboxes

Checkboxes are used to let the user select one or more values from a pre-defined set of options. The type attribute value for checkboxes input control is "checkbox".

**Example**

<form>

<b>SELECT SUBJECTS</b>

<br>

<input type="checkbox" name="subject" id="math" />

<label for="math">Math</label>

<input type="checkbox" name="subject" id="science" />

<label for="sceince">Science</label>

<input type="checkbox" name="subject" id="english" />

<label for="english">English</label>

</form>

**NOTE**: Just like radio buttons, a form may have multiple sets of checkboxes. In order to make sure the user selects options related to a given set, each checkbox  element must have matching name attributes. In the example above, each checkbox has a name attribute value as **subject**.

File select boxes

File select boxes are used to allow the user to select a local file on their computer and send it as an attachment to the webserver. It is similar to a text box with a button that allows the user to browse for a file. Instead of browsing for the file, the path and the name of the file can also be written. They are created by specifying a type attribute value as "file".

**Example**

<form>

<label for="fileselect">Upload:</label>

<input type="file" />

</form>

Text area

A text area is a multi-line text input control which allows users to provide a paragraph or multiple lines of text. It is created by using the "textarea" element.

This is one of the few input controls that DO NOT use the <input> element.

You can control the size of a text area by adding attributes "rows" and "cols" to specify the number of rows and columns of text it supports. Most often text area elements are resizable, but the default size is managed by those two attributes.

**Example**

<form>

<label for="Description">Description:</label>

<textarea rows="5" cols="50" name="Description"

id="Description"></textarea>

</form>

Select Boxes (Drop-downs)

Select boxes are used to allow users to select one or more options from a drop-down list. Select boxes are created by using two elements: <select> and <option>. The <select> element defines a drop-down while list items are defined within the select element using <option> elements.

**Example**

<form>

<label for="country">Country:</label>

<select name="country" id="country">

<option value="United States">United States</option>

<option value="Canada">Canada</option>

<option value="Mexico">Mexico</option>

</select>

</form>

Reset And Submit Buttons

The **submit** button allows the user to send the form data to the web server. You can define a submit button by specifying the type attribute as "submit".

The **reset** button is used to reset the form data and will display any default values. You can define a reset button by specifying the type attribute as "reset".

**Example**

<form action="test.php" method="post" id="users">

<label for="username">Username:</label>

<input type="text" name="username" id="Username" />

<input type="submit" value="Submit" />

<input type="reset" value="Reset" />

</form>

**Select and Multi-select**

Input Element in HTML Forms

An HTML form collects information from **** elements. You will specify an addition **type** attribute to indicate which field to display. Various fields can be created such as a text field, checkbox, password field, or radio button.

Text Field

A text field is a one-line input field that allows the user to input a line of text. Text Fields are created by specifying the type attribute value as "text".

The below example will display a text field with the label *Email Id:*

**Example**

<form>

<label for="email-input">Email Id:</label><br>

<input type="text" name="email-input" />

</form>

Password Field

Password fields are a type of text field in which the text entered is masked using asterisk or dots. This prevents others form viewing the screen to see what is typed in. Also, its created by specifying the type attribute value as "password".

**Example**

<form>

<label for="user-password">Password:</label><br/>

<input type="password" name="user-password"/>

</form>

**NOTE**: Although a password field is masked, it is NOT encrypted. You will have to use other measures such as HTTPS to encrypt data once the HTML form is submitted.

Radio Buttons

Radio Buttons are used to let the user select exactly one value from a list of predefined options. It is created by specifying the type attribute value as "radio".

**Example**

<form>

SELECT GENDER

<br/>

<input type="radio" name="gender" id="male">

<label for="male">Male</label><br>

<input type="radio" name="gender" id="female">

<label for="female">Female</label>

</form>

**NOTE**: A form may have multiple sets of radio buttons. In order to make sure the user only selects one option from a given set, each radio  element must have matching name attributes. In the example above, both buttons have a name attribute value as **gender**.

Checkboxes

Checkboxes are used to let the user select one or more values from a pre-defined set of options. The type attribute value for checkboxes input control is "checkbox".

**Example**

<form>

<b>SELECT SUBJECTS</b>

<br>

<input type="checkbox" name="subject" id="math" />

<label for="math">Math</label>

<input type="checkbox" name="subject" id="science" />

<label for="sceince">Science</label>

<input type="checkbox" name="subject" id="english" />

<label for="english">English</label>

</form>

**NOTE**: Just like radio buttons, a form may have multiple sets of checkboxes. In order to make sure the user selects options related to a given set, each checkbox  element must have matching name attributes. In the example above, each checkbox has a name attribute value as **subject**.

File select boxes

File select boxes are used to allow the user to select a local file on their computer and send it as an attachment to the webserver. It is similar to a text box with a button that allows the user to browse for a file. Instead of browsing for the file, the path and the name of the file can also be written. They are created by specifying a type attribute value as "file".

**Example**

<form>

<label for="fileselect">Upload:</label>

<input type="file" />

</form>

Text area

A text area is a multi-line text input control which allows users to provide a paragraph or multiple lines of text. It is created by using the "textarea" element.

This is one of the few input controls that DO NOT use the <input> element.

You can control the size of a text area by adding attributes "rows" and "cols" to specify the number of rows and columns of text it supports. Most often text area elements are resizable, but the default size is managed by those two attributes.

**Example**

<form>

<label for="Description">Description:</label>

<textarea rows="5" cols="50" name="Description"

id="Description"></textarea>

</form>

Select Boxes (Drop-downs)

Select boxes are used to allow users to select one or more options from a drop-down list. Select boxes are created by using two elements: <select> and <option>. The <select> element defines a drop-down while list items are defined within the select element using <option> elements.

**Example**

<form>

<label for="country">Country:</label>

<select name="country" id="country">

<option value="United States">United States</option>

<option value="Canada">Canada</option>

<option value="Mexico">Mexico</option>

</select>

</form>

Reset And Submit Buttons

The **submit** button allows the user to send the form data to the web server. You can define a submit button by specifying the type attribute as "submit".

The **reset** button is used to reset the form data and will display any default values. You can define a reset button by specifying the type attribute as "reset".

**Example**

<form action="test.php" method="post" id="users">

<label for="username">Username:</label>

<input type="text" name="username" id="Username" />

<input type="submit" value="Submit" />

<input type="reset" value="Reset" />

</form>

**Submitting Forms**

# HTML Forms

An **HTML form** is a section of a document that contains controls such as text fields, password fields, checkboxes, radio buttons, submit button, menus, etc. Using these elements the page can collect information from a user which is typically submitted to a web server. To create a form, you would use the <form> tag.

Why use an HTML Form?

* We use forms to collect some information/data form the user.
  + For example: If a user wants to purchase some items on the internet, he or she must fill out the form which will collect information such as the shipping address and payment details so that the item can be sent to the given address.

# Attributes Used in HTML Forms

There are several attributes that you can use on the <form> tag and on <input> elements. We will cover:

* action
* target
* name
* method
* value
* placeholder
* required

Take a look at the following example and find the different attributes:

<!DOCTYPE html>

<html>

<body>

<form action="/test.php" target="\_blank" method="GET">

Username:<br />

<input type ="text" name="username" placeholder="Username" required/>

<br />

Password:<br />

<input type ="password" name="password" />

<br /><br />

<input type ="submit" value ="Submit" />

</form>

</body>

</html>

### **The Action Attribute**

The action attribute indicates where the form data will be processed. Typically the value is a URL of a server. Generally, the form data is sent to a webpage on the webserver after the user clicks on the submit button.

In the above code, after clicking on the submit button, the form data would be sent to a page called test.php.

### **The Target Attribute**

The Target attribute is used to specify whether the submitted result will open in the current window, a new tab or on a new frame. The default value used is "self" which results in the form submission in the same window. To make the result display in a new browser tab, the value should be set to "blank".

In the above code, after clicking on the submit button, the result will open in a new browser tab. Most often this attribute is not present and the default value of "self" is used.

### **Name Attribute**

The name attribute should be provided for each input element. It is not required, but the value provides a label for the data once the form is submitted. If the name attribute is not specified in an input field then the data of that field will not be sent.

In the above code, after clicking the submit button, the form data will be sent to a page called /test.php. The data sent will include the username and password fields.

### **The Method Attribute**

The method attribute is used to specify the HTTP method used to send data while submitting the form. There are only two options available: GET and POST.

**GET** - When using the GET method, after the submission of the form, the form values will be visible in the address bar of the browser.

**POST** – When using the POST method, after the submission of the form, the form values will NOT be visible in the address bar of the browser.

### **The value Attribute**

The value attribute specifies an initial value for an input field. It also serves as the attribute to use when providing a button label for submit and reset input elements.

In the above example, there are no default values.

### **The placeholder Attribute**

The placeholder attribute specifies a hint that describes the expected value of the input field (a sample value or a short description of the expected format). The short hint is displayed in the input field before the user enters a value. The placeholder attribute works with the following input types: text, search, url, tel, email, and password.

In the above example, the text field has a placeholder of "Username".

### **The required Attribute**

The required attribute indicates an input field that must be filled out before submitting the form. In most modern browsers, it will prevent the user from submitting the form until an acceptable value is entered. The required attribute works with the following input types: text, search, url, tel, email, password, date pickers, number, checkbox, radio, and file.

In the above example, only the text field is required.  
**NOTE**: The required attribute doesn't have a value portion. You only need to specify the word 'required'.

### **The min and max Attributes**

The min and max attributes specify the minimum and maximum values for an input field. The min and max attributes work with the following input types: number, range, date, datetime-local, month, time and week.

**Tip**: Use the max and min attributes together to create a range of legal values. (For example: Set a maximum date or a minimum date)

Example

<form>

<label for="datemax">Enter a date before 1980-01-01:</label>

<input type="date" id="datemax" name="datemax" max="1979-12-31"><br><br>

<label for="datemin">Enter a date after 2000-01-01:</label>

<input type="date" id="datemin" name="datemin" min="2000-01-02"><br><br>

<label for="quantity">Quantity (between 1 and 5):</label>

<input type="number" id="quantity" name="quantity" min="1" max="5">

</form>>

**HTML5 Validation**

HTML5 Validation

When we submit a form, the data in the form will be sent to the server, before that we need to make sure that all the required details are filled out also in the correct format. The process of ensuring or validating the data before submitting to the server is called Client-side form validation.

Let us discuss form validation. For example, we submit any registration form, we may come across such messages listed below :

* "This field is required" (it can't be blank).
* "Please enter the valid phone number" (it should contain the only number).
* "Invalid email address" (it should be in "[lmn@asd.com](https://app.revature.com/)" format).
* "Your password must include one number, one uppercase letter, one lowercase letter, and one special character".

The browser displays the above messages by validating the data entered the registration form. It checks whether the data is in the correct format and satisfies the constraints set by the application or not. If the browser validates the data, then it is called as *client-side validation*. Validation done by the server is called as *server-side validation*.

There are two different types of client-side validation.

1. **Built-in form validation** - It uses HTML5 form validation features.
2. **JavaScript validation** - It is coded using JavaScript. This validation is completely customizable.

Built-in form validation

We have attributes that can be used with the form elements for validation. Some of the attributes are listed below:

* **required**: Used when the user must fill the field before submitting the form.
* **minlength and maxlength**: Used to specify the minimum and maximum length of the text.
* **min and max**: Used to specify the min and max values for the numerical fields.
* **type**: Defines the data should be a number or an email address or other predefined type.
* **pattern**: Defines a pattern (regular expression) the entered data needs to follow.

If the data entered in a form satisfies the constraints are considered **valid**. If not, it is considered **invalid**. We use : valid and : invalid CSS pseudo-class to differentiate between valid and invalid input fields. The:invalid CSS pseudo-class used to select and style form <input> elements whose value is *invalid* according to the validation attributes specified in the <input> element. Similarly, the :valid CSS pseudo-class selects and styles the vaild form input elements.

For example, the email inputs (<input type="email">) whose value does not match a valid email address pattern then the style defined by the : invalid CSS pseudo-class. Below, we have HTML and CSS code for this example.

HTML:

<form>

<label>

<span>Email:</span>

<input type="email" id="email" placeholder="name@domain.com">

</label>

</form>

CSS:

//:valid

.valid {

input[type=email]:valid { border-color: $g; }

}

//:invalid

.invalid {

input[type=email]:invalid { border-color: $r; }

**Overview of CSS**

Overview of CSS

CSS stands for **Cascading Style Sheets** - it is a language for styling HTML documents by specifying certain rules for layout and display in key/value pairs. Style Sheets are a simple and powerful method of allowing attachment of rendering information to HTML documents. It used to style the webpages by setting background-color, font color, font size, font family, etc.

A CSS consists of a *set of rules* that defines the styles for a web page. A CSS style rule composed of **selectors** and **declarations**. The selector is an HTML Element *like h3 used in the below example*. The declaration is comprised of a property and a value surrounded by curly braces. In the below example font-family, font-style and color were properties of the selector h3. Arial, italic and red were the values assigned, respectively, to the properties.

*Example:*

h3 {

font-family: Arial;

font-style: italic;

color: red

}

**CSS Box Model**

CSS Box Model

The CSS box model used to determine how our web page is rendered by browser. It considers each element on the page as a box, CSS allows you to apply different properties that define where and how that element appears. Web pages are made up of rectangular boxes arranged and related to each other.

Every box has 4 parts - **margin**, **border**, **padding** and **content**. The margin is an outermost box, inside that the border, then padding, then the content is innermost. All box sizes/formatting can be styled with CSS.

**Margin** - It is a space between border and margin. It is useful to separate the element from its neighbors. The dimensions are given by the margin-box width and the margin-box height.

**Border** - It is the area between the box’s padding and margin. Its dimensions are given by the width and height of the border.

**Padding** - It is a space around the content area and within the border-box. Its dimensions are given by the width of the padding-box and the height of the padding-box.

**Content** - It consists of content like text, image, or other media content. It is bounded by the content edge and its dimensions are given by content-box width and height.

**Inline, Internal, and External Stylesheets**

# Types of CSS

There are three types of CSS which are given below:

* Inline CSS
* Internal or Embedded CSS
* External CSS

### **Inline CSS**

Inline CSS contains the CSS property in the body section attached with element is known as inline CSS. This kind of style is specified within an **HTML tag using style attribute**.

**Example:**

<!DOCTYPE html>

<html>

<head>

<title>Inline CSS</title>

</head>

<body>

<p style = "color:#009900;

font-size:50px;

font-style:italic;

text-align:center;">

Hello World

</p>

</body>

</html>

### **Internal or Embedded CSS**

This can be used when a single HTML document must be styled **uniquely**. The CSS rule set should be within the HTML file in the head section i.e the CSS is embedded within the HTML file.

**Example:**

<!DOCTYPE html>

<html>

<head>

<title>Internal CSS</title>

<style>

.main {

text-align:center;

}

.hi{

color:#009900;

font-size:50px;

font-weight:bold;

}

.greeting {

font-style:bold;

font-size:20px;

}

</style>

</head>

<body>

<div class = "main">

<div class ="hi"> Hi, Everyone!!</div>

<div class ="greeting">

Hello World!!

</div>

</div>

</body>

</html>

### **External CSS**

External CSS contains separate CSS file which contains only style property with the help of tag attributes (For example class, id, heading, … etc). CSS property written in a separate file with .css extension and should be linked to the HTML document using **link** tag. This means that for each element, style can be set only once and that will be applied across web pages.

**Example:** The file given below contains CSS property. This file should be saved with an .css extension. For Ex: **style.css**

body {

background-color:powderblue;

}

.main {

text-align:center;

}

.hi {

color:#009900;

font-size:50px;

font-weight:bold;

}

#greeting {

font-style:bold;

font-size:20px;

}

Below, we have HTML file that makes use of the above created external style sheet (style.css). This can be achieved by using <link> tag. The <link> element has rel and href attributes. The rel specifices the relationship between the current document and the linked document. In this case, rel attribute value will be stylesheet because we going to add the external style sheet to the HTML document. The href attribute is used to specify the location of the external style sheet file.

<html>

<head>

<link rel="stylesheet" href="style.css"/>

</head>

<body>

<div class = "main">

<div class ="hi"> Hi, Everyone !</div>

<div id = "greeting ">

Hello world !!

</div>

</div>

</body>

</html>

**CSS Properties**

# CSS Properties

There are a number of CSS properties that you can use to style our webpage. Here we'll discuss some of the CSS properties such as Border, Padding, Margin, display, position, color,and text-align.

### **CSS Border Property**

The CSS border property allows to style the border area of a box. The properites and corresponding vlaues with examples covered under the CSS border are tabulated below:

| **Property** | **values** | **Usage** | **Example** |
| --- | --- | --- | --- |
| border-width | medium, thin, thick, length | Used to define the border area of a box | div { border-width: medium 10px thick 15px; } |
| border-style | none, hidden, dashed, dotted, double, groove, inset, outset, ridge and solid | sets the style of a box's border | p { border-style: dotted; } |
| border-color | hex-value for colors | specify the color of a box's border | p { border-style: solid; border-color: #ff0000; } |

### **CSS Padding Property**

The CSS padding property allow you to set the padding area for an element that separates its border from its content. The padding property can take one, two, three, or four values separated by white spaces as listed in the below table. Depending on the list of property values, the HTML element has the padding area on the top, bottom, right, and left.

| **Examples** | **Explanation** |
| --- | --- |
| p { padding: 70px; } | Sets the padding for an <p> element to 70 pixels for all four sides |
| p { padding: 35px 70px; } | Sets the padding for an <p> element to 35 pixels for top and bottom and 70 pixels for right and left sides. |
| p { padding: 35px 70px 40px; } | Sets the padding for an <p> element to 35 pixels for the top, 70 pixels for the left and right side and 40 pixels for the bottom. |
| p { padding: 35px 70px 40px 80px; } | Sets the padding for an <p> element to 35 pixels for the top, 70 pixels for the right side, 40 pixels for the bottom and 80 pixels for the left side. |

The padding property is a shorthand property for the padding-top, padding-right, padding-bottom, and padding-left properties. The below examples set padding on a specific side for the HTML element.

Examples:

h1 {

padding-bottom: 10px;

}

p {

padding-top: 20px;

padding-left: 50px;

}

### **CSS Margin Property**

The CSS margin property is similar to the CSS border property, but it sets the margins around the sides of an element's box instead of the border. It also takes one, two, three, or four values separated by white spaces. The shorthand properties are margin-top, margin-right, margin-bottom, and margin-left to set a margin on respective sides.

Example:

p {

margin-left: 10px;

margin-right: 30px;

}

h1{

margin: 25px 50px;

}

### **CSS Display Property**

The display property controls the display behaviour of an element. The CSS display property sets whether an element is treated as a block or inline elements and the layout used for its children, such as flow layout, flex or grid.

There are two types of HTML elements: **inline-level elements** and **block-level elements**. The differences between these elements affect how you use the box model. Both Inline and block-level elements appear within the body of an HTML page. But, inline-level elements are used to create a short structure that can have data and other inline elements. Inline level elements include <b>, <big>,< i>, <small>, <tt>, <abbr>, <acronym>, <code>, <strong>, etc.  
Block-level elements used to create larger structures than inline elements also it starts on new lines by default whereas inline-level elements not. Block elements include <p>, <h1>, <h2>, <h3>, <h4>, <h5>,<h6>, <ol>, <ul>, <pre>, <address>, <blockquote>, <dl>, <div>, <fieldset>, <form>, <hr>, <table>, etc .

The Syntax for the display property is selector {display: value}. The property values and description with examples are tabulated below:

| **Property value** | **Description** | **Example** |
| --- | --- | --- |
| block | behaves likes block-level elements | a {display: block;} |
| inline | behaves like inline-level elements | ul li { display: inline; } |
| none | elements doesn't generate boxes | h1 { display: none;} |

### **CSS Position Property**

The position property defines how an element will be arranged on a page. The Syntax for the position property is selector {position: value}. The property values are static, relative, absolute, fixed, or inherit.

static - The element's box is arranged automatically consistent with the normal flow.

relative - The element's box position is relative to its normal flow position. You can adjust the normal flow position by using the top, bottom, left and right properties.

absolute - The element's box arranged to an absolute position with reference to its containing block. Its containing block is that the nearest ancestor element that has its position property set to relative, absolute, or fixed. The top, right, bottom, and left properties are used to set the offset of the element's box with reference to its containing block.

fixed - The element's box position is offset from its browser window by using the top, right, bottom, and left properties. The element's box won't move when the browser window is scrolled.

inherit - The inherit keyword is employed to specify that the value for this property should be taken from the parent element. If inherit is used with the root element, then the initial value for this property is going to be used.

Example:

a {position: static;}

div {position: relative; top: 20px; left: 50px;}

h1 {position: absolute; top: 30px; left: 20px;}

div {position: fixed; top: 325px; left: 60px;}

### **CSS Color property**

The color property is used to specify the foreground color of text.The color properties are set using 5 different color notation types which is listed below:

a {color: red;}

div {color: #3c5;}

h1 {color: #ffa500;}

div {color: rgb(100,20,255);}

#id1 {color: rgb(30%,50%,70%);}

### **CSS text-align property**

The text-align property is used to align the content inside the element. The text inside the element can be aligned in 4 ways - left, right, center and justify.

Example: The text-align properties are set to left, right, justify, and center.

div {text-align:left;}

h1 {text-align: right;}

p {text-align: justify;}

div {text-align: center;}

**Element Selectors**

Element Selector

The element selector selects HTML elements by their name / tag name *like a, h1, div, p etc*.

*Example:* Here, we use <p> as an element selector. The text inside the <p> will be center-aligned also blue color.

<!DOCTYPE html>

<html>

<head>

<style>

p {

text-align: center;

color: blue;

}

</style>

</head>

<body>

<p>This style will be applied on every paragraph.</p>

<p> Here also</p>

</body>

</html>

**Class and Id Selectors**

Class Selector

In the CSS, the class selector is a name preceded by a period (“.”). It uses the class attribute of an HTML element to match the specific HTML element. We can have a Class selector specific to an HTML element *like we have p.class in the below example*.

In the below example, we have two class selectors inside the <style> element. The class selector .intro is applied to the element which has an attribute called class, whose value is intro and the p.intro class selector is applied to the <p> element which has an attribute called class, whose value is intro. Also, the <p> element without the class attribute doesn't get affected.

<!DOCTYPE html>

<html>

<head>

<style>

.intro {

text-align: center;

color: red;

}

p.intro {

text-align: center;

color: blue;

}

</style>

</head>

<body>

<h1 class="intro">Red and center-aligned heading</h1>

<p class="intro">blue and center-aligned paragraph.</p>

<p> this will not be affected </p>

</body>

</html>

ID Selector

In the CSS, the ID selector is a name preceded by a hash character (“#”). It uses the id attribute of an HTML element to match the specific HTML element. The **id** of an element should be unique within a page, so the id selector is used to select one unique element.

*Example:* Here, we use #para1 as an ID selector. Inside the body, we have two <p> elements. The CSS style rule applied to the element which has an attribute called id, whose value is para1. Therefore, Hello World! will be center-aligned also blue color.

<!DOCTYPE html>

<html>

<head>

<style>

#para1 {

text-align: center;

color: blue;

}

</style>

</head>

<body>

<p id="para1">Hello World!</p>

<p>This paragraph is not affected by the style.</p>

</body>

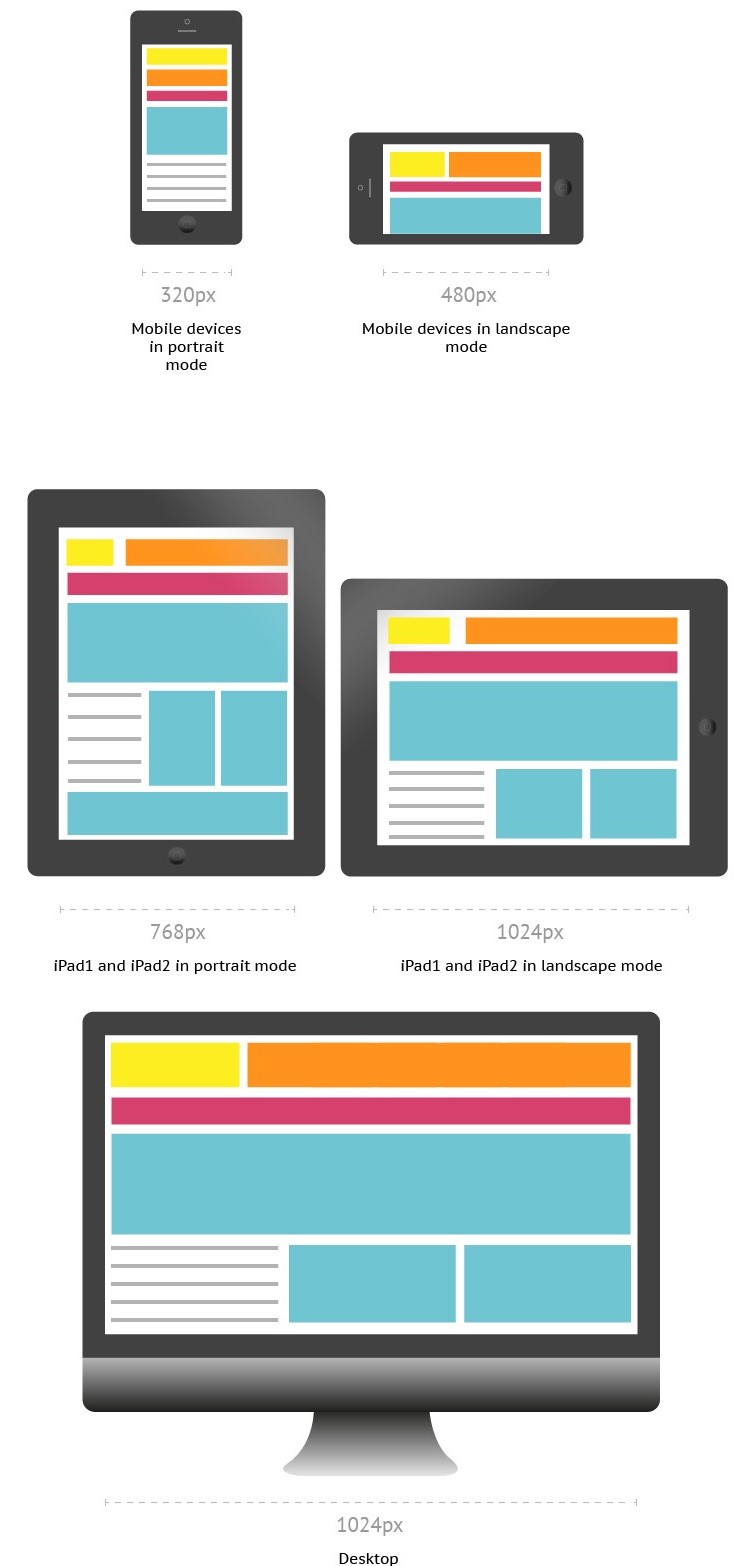
</html>

*NOTE:* The id name should start with the alphabet, not with numbers. Also, the HTML element without the 'id' attribute doesn't get affected.

**Responsive Web Design Overview**

# Responsive Web design

Responsive Web design is the approach that allows websites and pages to render (or display) on all devices and screen sizes by automatically adapting to the screen, whether it’s a desktop, laptop, tablet, or smartphone. Responsive web design works through CSS, using various settings to serve different style properties depending on the screen size, orientation, resolution, color capability, and other characteristics of the user’s device. It is a combination of flexible grids, flex boxes, flexible images, and media queries.



# CSS3 Media Queries

CSS3 supports responsive web design, all kinds of transitions, transformations, and animations and provides box-sizing tools that enable the user to adjust the size of any element without changing the dimensions or padding of the element.

Media queries allow you to customize the presentation of your web pages for a specific range of devices like mobile phones, tablets, desktops, etc. without any change in markups. It composed of a media type and expressions that check for the conditions of particular media features. It is a logical expression that is either true or false.

**Syntax:** A media query consists of an optional **media type** and any number of **media feature** expressions. Multiple queries are often combined in various ways by using **logical operators**. Media queries are case-insensitive. A media query is true if the media sort of the media query matches the media sort of the device and every one expression within the media query are true. It uses the @media rule to incorporate a block of CSS properties as long as a particular condition is true. Queries involving unknown media types are always false.

@media not|only mediatype and (mediafeature and|or|not mediafeature) {

CSS-Code;

}

Media Types - It describes the category of a device.

* all - used for all media type devices
* print - used for printers
* screen - used primarily for screens like computer screens, tablets, smart-phones, etc.
* speech - used for screenreaders that "reads" the page aloud

Media features - It describe specific characteristics of the user agent, output device, or environment. Some of the media features are grid, height, width, hover, max-aspect-ratio, max-color,max-color-index, max-height,etc.

Logical Operators - It used to compose a media query . Logical Operators used in media queries are not, and, and only.

Example: It changes the background color of the <body> element to "red" and the font style to "Arial" when the browser window is 600px wide or less.

@media only screen and (max-width: 600px) {

body {

background-color: red;

font-family: Arial;

}

}

# Bootstrap

Bootstrap is an open-source framework and mobile-first approach for developing responsive websites. It is a front-end framework programmed to support both HTML5 and CSS3. It comprises the list of components such as Typography, Code, Table, Forms, Button, Images, Icons, etc. A responsive website is a website that automatically adjusts the screen size and looks good on all devices, from smartphones to a desktop. It is easy to use, saves time and customizable. Bootstrap 4 is the newest version of Bootstrap.

### **How to Download Bootstrap?**

There are two ways to download and start using Bootstrap for our website:

1. Download Bootstrap 4 - You can download the bootstrap from [getbootstrap.com](https://getbootstrap.com/).
2. Include Bootstrap from a CDN - You can skip the download with [BootstrapCDN](https://www.bootstrapcdn.com/) by copying the links and paste it in the head section of the html code.

## Bootstrap grid systems

Bootstrap grid system consists of series of containers, rows, and columns to layout and align content. It creates a responsive layout and built with grid and flexboxes. Bootstrap classifies the screen sizes ranging from extra small to extra large based on the pixels. The transition between the various screen sizes is known as breakpoints. Bootstrap grid system provides a set of responsive classes to specify the screen size.

### **Bootstrap containers**

Containers are the basic layout elements used to wrap the content in the website. The container is the root of the Bootstrap 4 grid system which consists of all the elements and controls the layout width of a page.

Bootstrap provides two container classes:

* .container - used to provide the responsive fixed width container
* .container-fluid - used to provides a full-width container that spans the entire width of the viewport.

<div class="container">

Hello! I am in a Fixed container.

</div>

<div class="container-fluid">

Hello! I am in a Fluid container.

</div>

### **Bootstrap rows**

Bootstrap rows are horizontal slices of the screen. They are only used for containing columns or a wrapper for columns. They have to be placed in containers to avoid the horizontal scroll on the page. The bootstrap columns should be children for the row, to align properly. .row class is used to create the rows inside the container.

<div class="row">

...

</div>

### **Bootstrap columns**

.col class sets the width for the column dynamically that means we can set the width of each column in a row. Grid system supports a maximum of 12 columns in a row and anything after that will be shifted to a next row.

You can set the size for the column (ranging from 1 to 12). For example: The .col-4 class creates 3 equal columns (because 12/4= 3 columns). The col-6 class creates 2 equal columns. We can also set different sizes for them.

<div class="row">

<div class="col-5">

This is col-5

</div>

<div class="col-7">

This is col-7

</div>

</div>

Also, You can set the Breakpoints for columns and used to specify the screen resolution. .col-[breakpoint] class is used to deine the behaviour for the columns in the displayed devices. There are 4 different breakpoint class listed below:

* .col-sm - used for small devices where the screen width is equal to or greater than 576px
* .col-md - used for medium devices where the screen width equal to or greater than 768px
* .col-lg- used for large devices where the screen width equal to or greater than 992px
* .col-xl - used for extra-large devices where the screen width equal to or greater than 1200px

<div class="row">

<div class="col-lg">

This is a column

</div>

</div>

We can combine the sizes and breakpoints and use a single class with the format .col-[breakpoint]-[size]. Below we have a simple example for Bootstrap Grid System.

<!-- Using ootstrap container class -->

<html>

<head>

<title>Bootstrap Grid System </title>

<!-- Add Bootstrap Links -->

<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">

<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>

<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"></script>

</head>

<body>

<div class="container" >

<div class="row" >

<div class="col-4 border" >

This is a col-lg-4

</div>

<div class="col-3 border">

This is a col-lg-3

</div>

</div>

<div class="row">

<div class="col-sm-6 border">

This is a col-lg-6

</div>

<div class="col-sm-6 border">

This is a col-lg-6

</div>

</div>

</div>

</body>

</html>

## Bootstrap Tables

The bootstrap provides classes to style of HTML tables. To change the default style of HTML table, add bootstrap .table class to the <table> element. Bootstrap table classes used to create striped table rouws, add borders to the cell and the tables, add the colors to the table rows, table headers and also allow us to differentiate handover rows, etc.

<div class="container">

<table class="table ">

...

</table>

</div>

The following classes are used with .table class like <table class="table [class-name]"> to add styles to the table:

* .table-dark - used to get a dark background table with the light text.
* .table-bordered - adds the border on all sides of the table and cells.
* .table-striped - used to add zebra-stripes table rows.
* .table-hover - applies the hover effect to the table rows and cells, i.e, you will get grey background when a cursor rolls over a cell or a row.
* .thead-light - used within <thead>, it makes the table header to appear light.
* .thead- dark - used within <thead>, it makes the table header to appear dark gray.

Contextual classes is used to color the table rows <tr> and cells <td> individually. We can used these contextual classes in <tr> and <tr> of the <table> element: .table-warning, .table-primary, .table-info, .table-success, .table-danger, .table-active, .table-light, .table-secondary and .table-dark. (for example: <tr class="table-primary">).

## Bootstrap Alerts

Bootstrap alerts provide the contextual feedback messages for user actions on the page. Alert boxes contain text that needs the user's attention. The .alert class is display the alert message on website. The .alert class is used as base class then followed by contextual classes . The contextual alert classes are: .alert-success, .alert-info, .alert-warning, .alert-danger, .alert-primary, .alert-secondary, .alert-light and .alert-dark. Also, we can dismiss/close any alert messages by using .alert-dismissible class. We can animate alerts when dismissing by using .fade and .show classes. Just add the data-dismiss="alert" attribute and class = "close" attribute to the <button> element to close the alert message boxes.

In the below example, we are creating a success alert message box by adding the contextual class .alert-success to the .alert base class.

<div class="alert alert-success alert-dismissible fade show"> <strong>Success!</strong>

Your details submitted successfully.

<button type="button" class="close" data-dismiss="alert">&times;</button>

</div>

## Bootstrap forms

Bootstrap forms allow us to create elegant forms on the website. We can style the all textual form controls like input, select, textarea, etc., by using .form-control class. There are 3 different types of form layout - Vertical Form layout, Horizontal Form layout, Inline Form layout.

Vertical Form Layout - This is a default default form layout provided by Bootstrap.

Horizontal Form Layout - In this layout, labels and form controls are aligned side-by-side by using the Bootstrap grid classes. The .row class and the .col-\*-\* grid classes used on the form groups to define the width of the layout. Also, to center them vertically we use .col-form-label on the <label>elements.

Inline Form layout - In this layout, a series of labels, form controls, and buttons are displayed in a single horizontal row. The .form-inline class is used within &lt;form&gt; element to create inline form layout.

We can stack the checkboxes or radio buttons vertically (line by line) , by appling the class .d-block on each <label> element.

Below, we have a simple example for Horizontal Form Layout.

<div class="form-group row">

<label for="inputEmail" class="col-sm-2 col-form-label">Email</label>

<div class="col-sm-10">

<input type="email" class="form-control" id="inputEmail" placeholder="Email">

</div>

</div>

## Bootstrap buttons

Bootstrap includes several button styles in which each styles serves a semantic purpose.The .btn classes are designed to be used with the <button>, <a> and <input> element. The contextual classes used with the .btn classes are btn-primary, btn-secondary, btn-success, btn-danger, btn-warning, btn-info, btn-dark, btn-light, and btn-link. When using button classes on <a> elements, we should use role="button" attribute to convey the purpose.The .btn-outline-\* used to remove all background colors on any button. The .btn-lg or.btn-sm classes are used to create larger and smaller buttons. The .active and .disabled class are used to represent the active and disabled state programmatically.

Example:

<button type="button" class="btn btn-primary btn-lg">Large button</button>

<button type="button" class="btn btn-outline-secondary btn-lg">Large button</button>

## Bootstrap Navbar

Bootstrap allows us to create a responsive navigation header that includes support for navigation, branding, collapse plugin and more. Navbars require a wrapping .navbar with .navbar-expand{-sm|-md|-lg|-xl} for responsive collapsing and color scheme classes. Some of the navbars sub-components are listed below with their purposes.

* .navbar-brand - used with most elements that contain the name of the company, product, or project.
* navbar-toggler - used for collapse plugin and other navigation toggling behaviors (allows to change the position on mobile devices)
* .form-inline - used with any form-controls and actions.
* .navbar-text - used to align the text vertically also makes it centered.
* .collapse.navbar-collapse - used for grouping and hiding navbar contents depending upon the screen resolution.

## Bootstrap Colors

Bootstrap provides color utility classes that support for styling links with hover states.

Text color - used to set the color for text of an element. The contextual text color classes are .text-muted, .text-primary, .text-success, .text-info, .text-warning, .text-danger, .text-secondary, .text-white, .text-dark, .text-body, and .text-light.

Background color - used to set the background color for an element. The contextual background color classes are similar to the contextual text color classes, here we use .bg-\* instead of text-\*.

Example:

<!-- setting text color-->

<p class="text-success">.text-success</p>

<p class="text-danger">.text-danger</p>

<p class="text-warning">.text-warning</p>

<!--setting background and text color-->

<div class="bg-light text-dark">.bg-light</div>

<div class="bg-dark text-white">.bg-dark</div>

<div class="bg-white text-dark">.bg-white</div>

<div class="bg-transparent text-dark">.bg-transparent</div>

**Intro to AWS S3**

## Intro to Amazon Simple Storage Service (Amazon S3)

Amazon Simple Storage Service (Amazon S3) is an **object storage service** that offers industry-leading scalability, data availability, security, and performance. This means customers of all sizes and industries can use it to store and protect any amount of data for a range of use cases, such as data lakes, websites, mobile applications, backup and restore, archive, enterprise applications, IoT devices, and big data analytics.

### **S3 as an Object Storage System**

Amazon S3 is an object storage system which means the data is stored as individual objects rather than in some kind of hierarchy like you would see in a file system or directory structure.

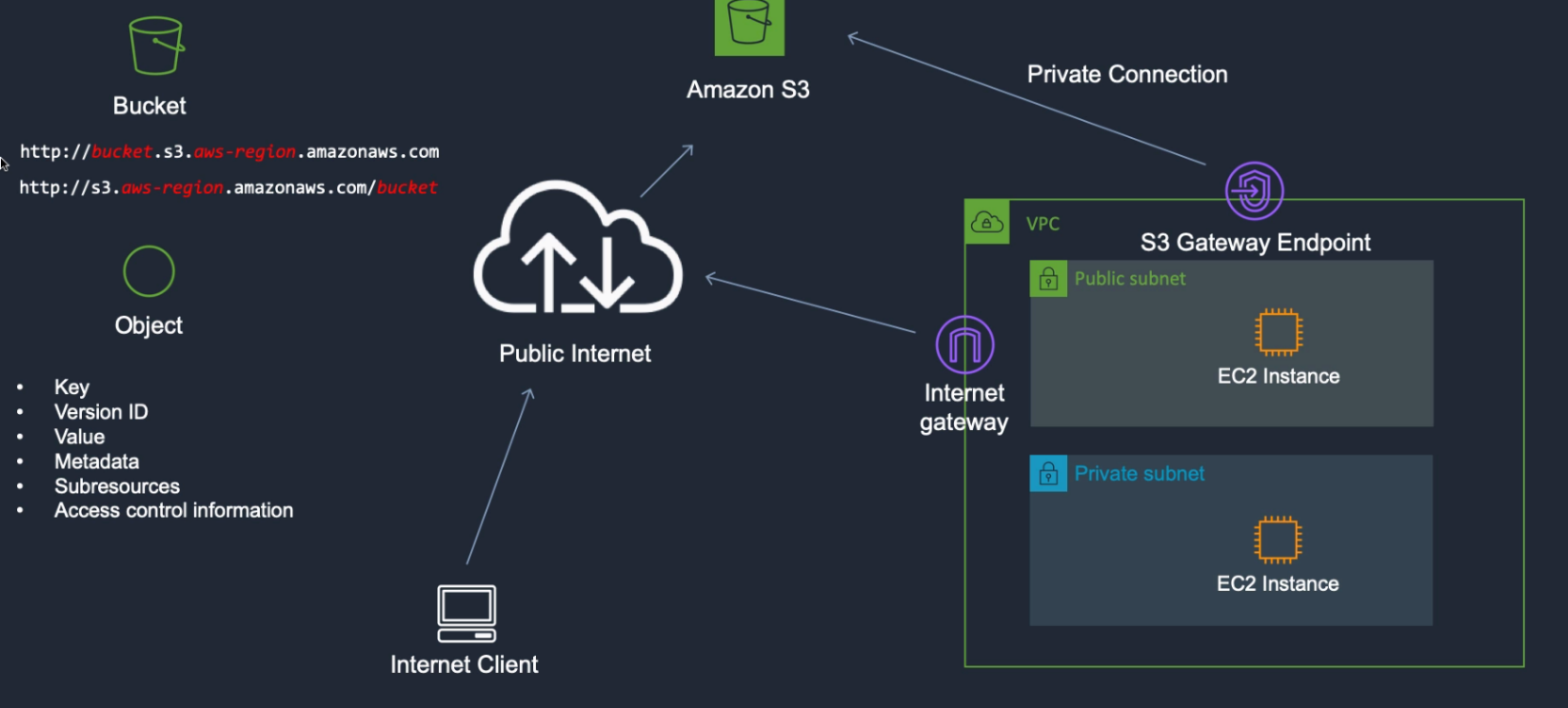
* Each individual object is put into a **bucket** and you connect to Amazon S3 using a URL.
* The URL will have the name of your object and the name of your bucket. The bucket is just the container in which you put your objects.
* You use what's called a REST API to connect to S3 using a URL.
* Your browser does a HTTP PUT request and it puts the objects in the bucket.



### **Use Cases for S3**

You can store any type of file in S3.

* Backup and Storage
* Application Hosting
* Media Hosting
* Software Delivery - host software apps that your customers can download
* [Static Website Hosting](https://app.revature.com/) - You can configure a static website to run from an S3 bucket.



### **References**

* [S3 Documentation](https://docs.aws.amazon.com/AmazonS3/latest/dev/Welcome.html)

**AWS S3 Bucket Configuration**

## AWS S3 Bucket Configuration

* To upload your data (photos, videos, documents etc.) to Amazon S3, you must first create an S3 bucket in one of the AWS Regions. You can then upload your data objects to the bucket.
* Every object you store in Amazon S3 resides in a bucket. You can use buckets to group related objects in the same way that you use a directory to group files in a file system.
* Amazon S3 creates buckets in the AWS Region that you specify. You can choose any AWS Region that is geographically close to you to optimize latency, minimize costs, or address regulatory requirements.

### **To Create a Bucket**

1. Sign in to the AWS Management Console and open the Amazon S3 console at <https://console.aws.amazon.com/s3/>.
2. Choose Create bucket.
3. In Bucket name, enter a DNS-compliant name for your bucket.

The bucket name must:

* + Be unique across all of Amazon S3.
  + Be between 3 and 63 characters long.
  + Not contain uppercase characters.
  + Start with a lowercase letter or number.

4. In `Region`, choose the AWS Region where you want the bucket to reside. 5. In `Bucket settings for Block Public Access`, uncheck the `Block Public Access` settings that you want to apply to the bucket. 6. Choose `Create bucket`.

### **Deleting a Bucket**

Follow the steps provided in the [AWS User Guide for deleting a bucket](https://docs.aws.amazon.com/AmazonS3/latest/user-guide/delete-bucket.html)

**Once you have created a bucket, you can use it to**[**host static websites**](https://app.revature.com/)**.**

**Hosting Static Sites on S3**

## Hosting Static Sites on S3

You can use Amazon S3 to host a static website. On a static website, individual webpages include static content.

By contrast, a dynamic website relies on server-side processing, including server-side scripts such as PHP, JSP, or ASP.NET. Amazon S3 does not support server-side scripting, but AWS has other resources for hosting dynamic websites. To learn > more about website hosting on AWS, see [Web Hosting](https://aws.amazon.com/websites/).

* To configure your bucket for static website hosting, you can use the AWS Management Console without writing any code.
* You can also create, update, and delete the website configuration programmatically by using the AWS SDKs. The SDKs provide wrapper classes around the Amazon S3 REST API. If your application requires it, you can send REST API requests directly from your application.
* To host a static website on Amazon S3, you [configure an Amazon S3 bucket for website hosting](https://app.revature.com/) and then upload your website content to the bucket.

### **Steps to Host a Static Website on your Previously Created S3 Bucket**

1. Click on your bucket in your S3 Management Console
2. Select Overview > Create Folder - this is where you will upload static HTML files from your computer to be hosted on the S3 bucket.
3. Copy and Paste the following dummy HTML text into a file titled index.html on your computer.

<html>

<head>

<title>Here's a title, displayed at the top of the window.</title>

</head>

<body>

<h1>A main heading, usually the same as the title.</h1>

<p>Be <b>bold</b> in stating your key points. Put them in a list: </p>

<ul>

<li>The first item in your list</li>

<li>The second item; <i>italicize</i> key words</li>

</ul>

<p>Add a link to your favorite <a href="https://revature.com/learn-to-code/">Web site</a>.

Break up your page with a horizontal rule or two. </p>

<hr>

<p>Finally, link to <a href="page2.html">another page</a> in your own Web site.</p>

<!-- A copyright notice.-->

<p>&#169; Revature 2020</p>

</body>

</html>

1. Select Upload and upload the index.html file you just saved to your computer.
2. Select Properties
3. Select Static website hosting
4. Select Use this bucket to host a website > enter the name of the file we're hosting: index.html.
5. Return to your bucket > click on the uploaded index.html file > click Make public.
6. Visit the Object URL and you will see your static webpage hosted on your configured S3 bucket!

### **References**

* [Configuring a bucket as a static website using the AWS Management Console](https://docs.aws.amazon.com/AmazonS3/latest/dev/HowDoIWebsiteConfiguration.html)

Day – 2

**Client and Server Architecture**

### **Client and Server Architecture**

A client-server architecture is a networking model in which the server provides services to clients to perform user-based tasks. A client and a server are two pieces of software that might be on the same computer, or two different computers that might be separated by miles but connected by the Internet.

**Server** - A server is software designed to process requests and deliver responses to another computer over the internet.

**Client** - A client is a program that runs on a local machine requesting service from the server.



A Client and a Server establish a connection according to a set of rules called a protocol. There are quite a few protocols for different purposes, but one of the most popular is the **HTTP protocol**. Once the connection is established, the Client sends **HTTP Requests** to the server in the form of XML or JSON, which both entities (Client and Server) understand. After parsing the request, the Server responds with appropriate data by sending back an **HTTP Response**.

### **Types of Client-Server Architecture**

**2 tier architecture** - The user interface stored at the client machine and the database stored on the server. If Business Logic & Data Logic collected at a client-side, then it is known as a fat client thin server architecture. If Business Logic & Data Logic handled on the server, then it is known as a thin client fat server architecture. 2 tier architecture has some limitations in performance, security, and portability.

**3 tier architecture** - Three-tier architecture has a middleware between the user interface and database. The 3 tiers are named the presentation tier, application tier, and data tier. The presentation tier is the front end layer and consists of the user interface. The application tier contains the functional business logic which drives an application’s core capabilities. The data tier consists of a database system and the data access layer.

**n-tier architecture** - In n-tier architecture, there are multiple Business Logic & Data Logic layers. It increases the flexibility and reusability of applications but can be difficult to implement.

**What is a servlet?**

## Servlets

A website can consist of both static and dynamic webpages. A static webpage is a pre-built HTML page with the content explicitly written into the code, and stored in the webserver. Static web pages display the same content each time we visit. A dynamic webpage loads dynamic content such as stock prices, weather information, news, and sports updates at different points of time. In Java, there exists a way to generate static webpages with dynamic data, and that's with **Java Servlets**.

A **Servlet** is a Java class that takes incoming requests, processes them, and generates a response to send back to the user. For example, an HttpServlet takes an HTTP request, processes its headers and content, and uses that information to write HTML, CSS, and JavaScript code into an HTTP response that can be sent back to the user's browser. The **Servlet container** is the component of an **application server** that interacts with Java servlets and is responsible for managing the execution of servlets and JSP pages for Java applications.

User --> HTTP Request --> Application Server --> Servlet Container

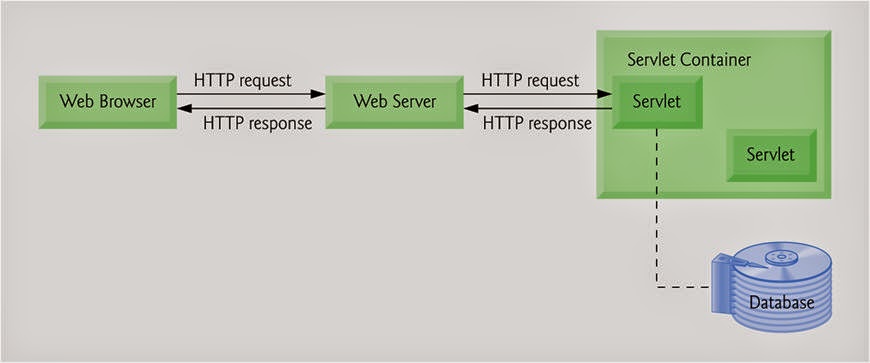
|

V

User <-- HTTP Response <-- Application Server <-- Servlet(s)

### **How do servlets work?**

When a client sends a request to the application server, the application server receives and passes the request to the appropriate servlet. The servlet processes the request, generates the response, and sends the response back to the application server. The application server sends the response back to the client. Most servlets are HTTPServlets, which receive HTTP requests and generate HTTP Responses out of HTML, CSS, and JavaScript code.



### **References**

* [Java Servlet Technology Overview](https://www.oracle.com/java/technologies/servlet-technology.html)
* [Introduction to Servlets](https://www.edureka.co/blog/java-servlets) - [Video Tutorial](https://youtu.be/ewiOaDitBBw)
* [Servlet Documentation - Oracle](https://javaee.github.io/servlet-spec/downloads/servlet-3.1/Final/servlet-3_1-final.pdf)

**Web Server and Servlet Container**

## Servlets

A website can consist of both static and dynamic webpages. A static webpage is a pre-built HTML page with the content explicitly written into the code, and stored in the webserver. Static web pages display the same content each time we visit. A dynamic webpage loads dynamic content such as stock prices, weather information, news, and sports updates at different points of time. In Java, there exists a way to generate static webpages with dynamic data, and that's with **Java Servlets**.

A **Servlet** is a Java class that takes incoming requests, processes them, and generates a response to send back to the user. For example, an HttpServlet takes an HTTP request, processes its headers and content, and uses that information to write HTML, CSS, and JavaScript code into an HTTP response that can be sent back to the user's browser. The **Servlet container** is the component of an **application server** that interacts with Java servlets and is responsible for managing the execution of servlets and JSP pages for Java applications.

User --> HTTP Request --> Application Server --> Servlet Container

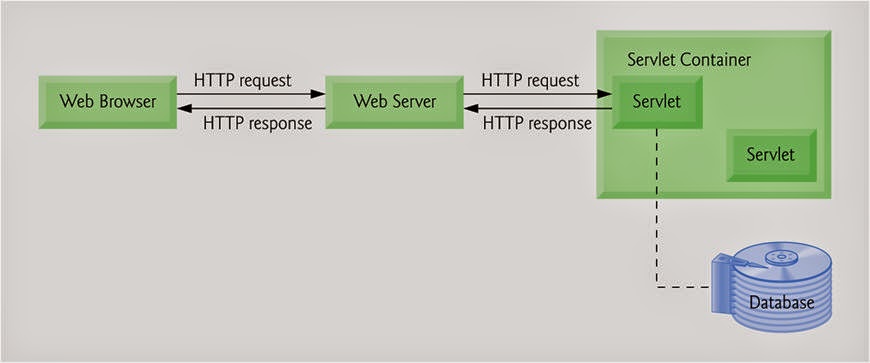
|

V

User <-- HTTP Response <-- Application Server <-- Servlet(s)

### **How do servlets work?**

When a client sends a request to the application server, the application server receives and passes the request to the appropriate servlet. The servlet processes the request, generates the response, and sends the response back to the application server. The application server sends the response back to the client. Most servlets are HTTPServlets, which receive HTTP requests and generate HTTP Responses out of HTML, CSS, and JavaScript code.



### **References**

* [Java Servlet Technology Overview](https://www.oracle.com/java/technologies/servlet-technology.html)
* [Introduction to Servlets](https://www.edureka.co/blog/java-servlets) - [Video Tutorial](https://youtu.be/ewiOaDitBBw)
* [Servlet Documentation - Oracle](https://javaee.github.io/servlet-spec/downloads/servlet-3.1/Final/servlet-3_1-final.pdf)

**Installation: Tomcat server**

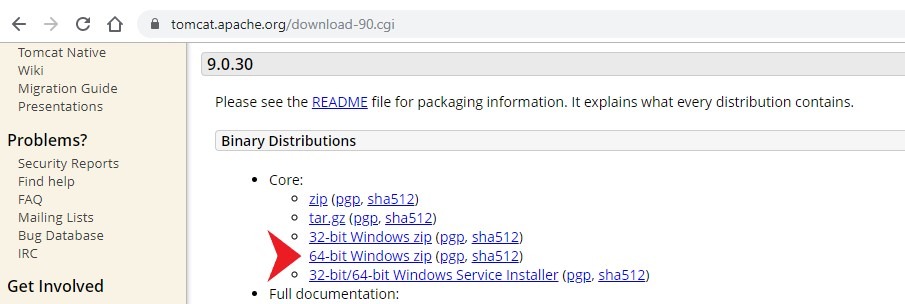
## Apache Tomcat

[Apache Tomcat](https://en.wikipedia.org/wiki/Apache_Tomcat) is an **open-source** implementation of the Java Servlet, Java Server Pages, Java Expression Language, and WebSocket technologies. Apache Tomcat is an **application server** that allows us to run servlet and JavaServer Pages (JSP) based web applications. The default port for the Apache Tomcat service is **8080**.

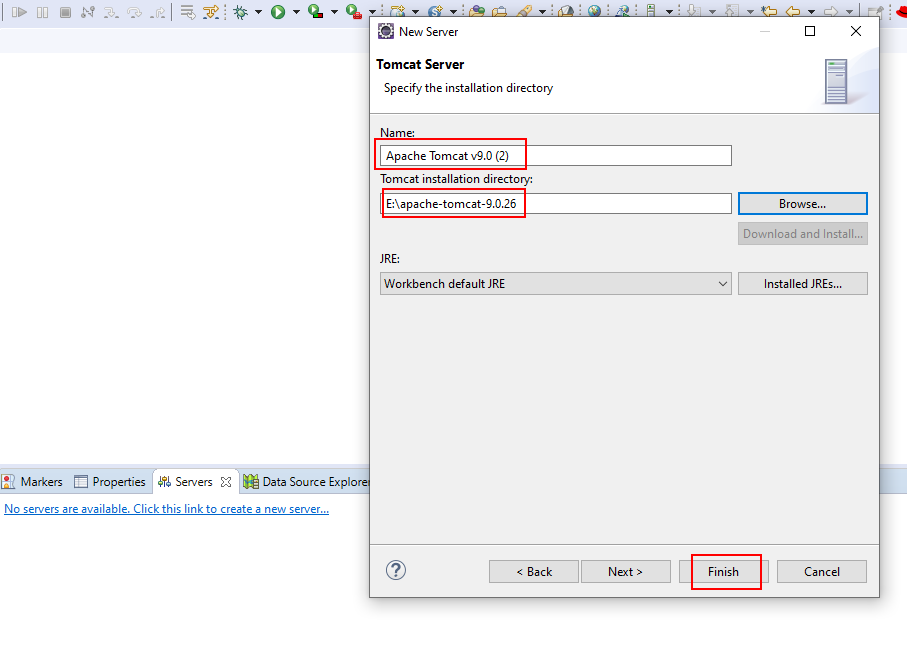
## Installing Apache Tomcat web server in Eclipse IDE

Before installing Apache Tomcat, ensure that the JDK installed and the Java environment variable configured.

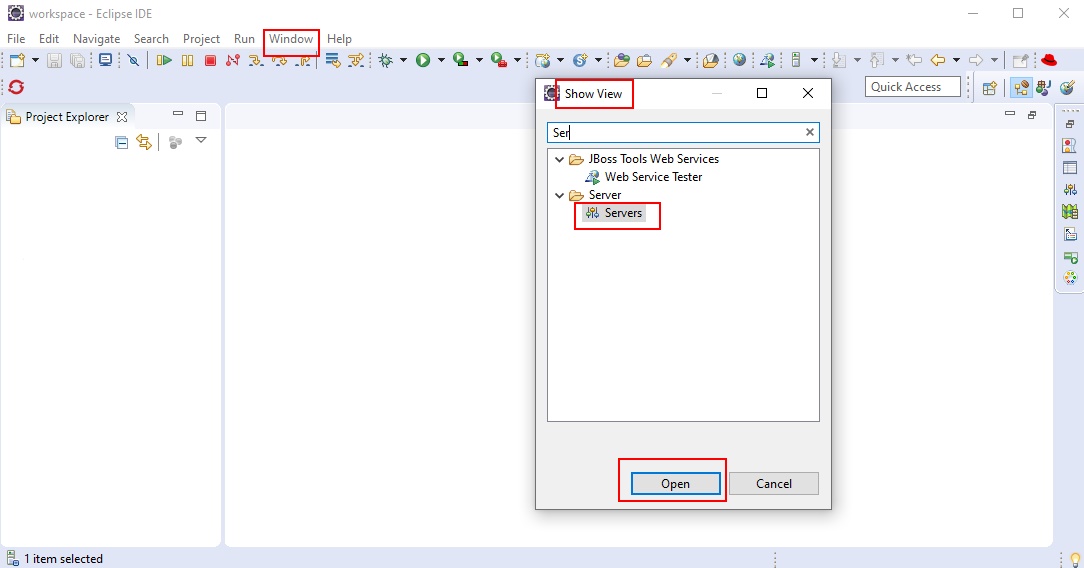
**Step 1:** Download Apache Tomcat server from [tomcat.apache.org](https://tomcat.apache.org/download-90.cgi). Extract files from the downloaded zip file.



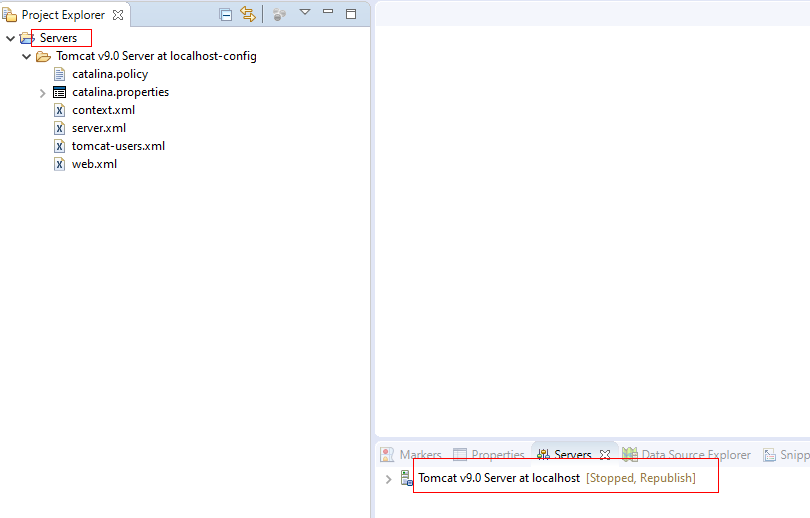
**Step 2:** In the Eclipse IDE, click on the **Servers** Tab. You'll find a link to create a new server. Click on that will open a New Server dialog box. In that dialog box, select the server type. Select **Apache -> Tomcat v9.0 Server** and click Next.



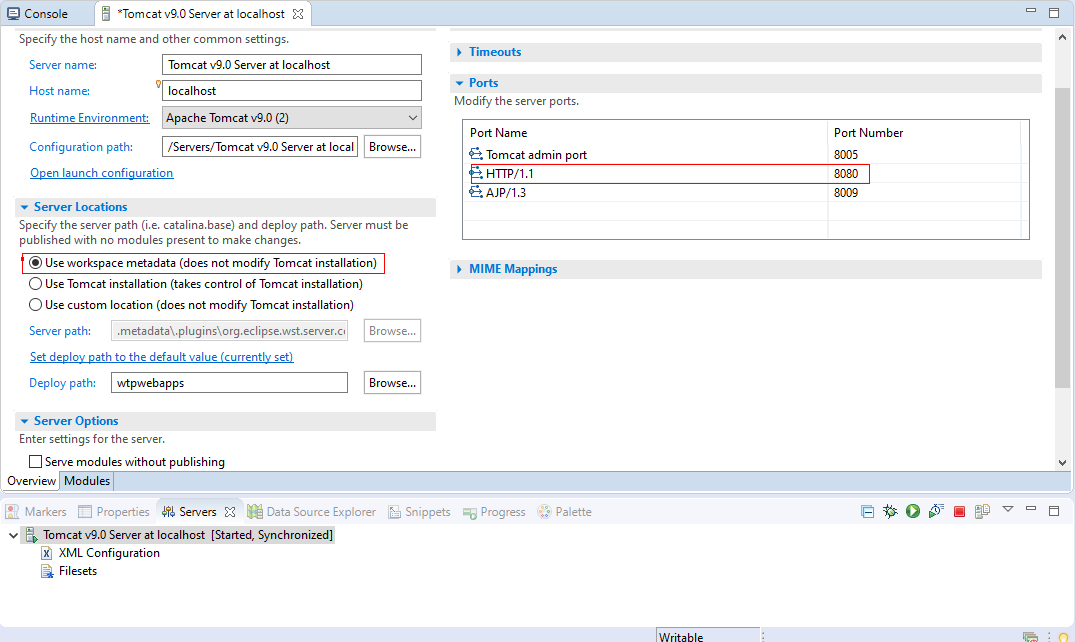
If the **Servers** tab is not visible, it can be configured by navigating to the Window tab in the Eclipse navbar. Click on the **Window** tab and select **Show View** -> **Other**. In the Show View dialog box, select ***Servers*** under the **Server** Folder and click **Open**. Now, you able see the Servers tab on the Eclipse window.



**Step 3:** Then, specify the Tomcat installation directory. Just browse the location where you extracted the apache-tomcat-9.0.26 files and then click FINISH. Under Server Tab, you should see Tomcat v9.0 Server at localhost [Stopped, Republish].



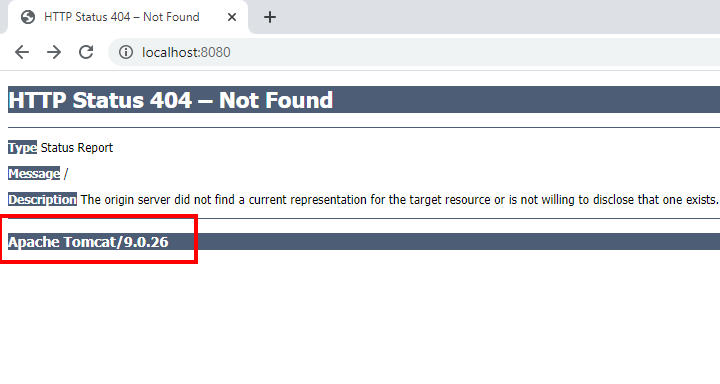
**Step 4:** Right-click on the server runtime and click Start to start the Tomcat server. Double click on the Tomcat v9.0 Server at localhost [Started, Synchronized] to view the HTTP port on which the tomcat server is running, and its configurations.



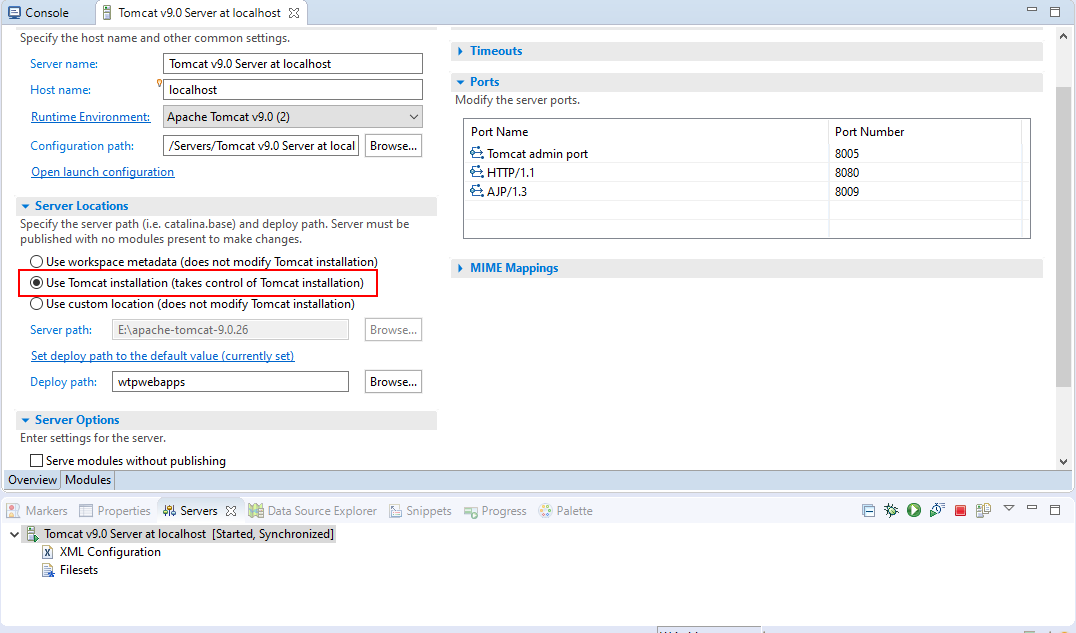
Here, we can configure:

* Server Name - The name of the server which appears under the server tab.
* Configuration Path - used to specify the location of the server configuration.
* Server Locations – used to specify the server path (location of the Apache Tomcat server installed) and the deploy path.
* Publishing – used to configure how the modules are publishing.
* Timeouts - used to specify the time limit to complete the server operations. We can set the timeouts for starting/stopping the server. If the server timeout is very low, the server might fail to start.
* Ports – used to set the server ports.
* MIME mappings – used to set the various MIME type mappings.
* Server Launch Configuration – used to configure the VM arguments, classpath, etc.
* Server Options – used to enable/disable features like security, auto-reload of modules by default, etc.

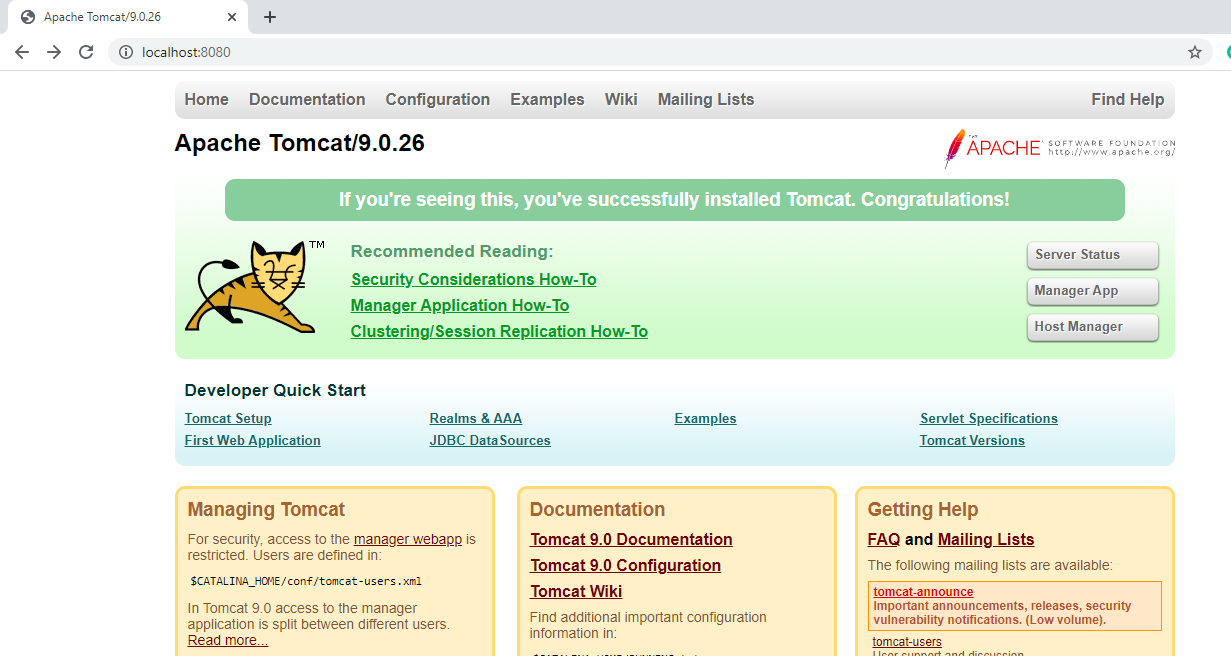
**Step 5:** The Tomcat server is up and running on port 8080. To verify, open your browser and visit <http://localhost:8080/>. You may see the HTTP-404 error page, but that response should come from the Apache Tomcat server.



You can also view the Apache Tomcat/9.0.26 home page by selecting the Use Tomcat installation option under the Server Locations.



Restart the Tomcat server and visit <http://localhost:8080/>, you'll see the Apache Tomcat/9.0.26 home page.



### **References**

* [Apache Tomcat 9 Docs](https://tomcat.apache.org/tomcat-9.0-doc/)
* [Apache Tomcat 9 Configuration Reference](https://tomcat.apache.org/tomcat-9.0-doc/config/index.html)
* [Video Tutorial - Install Apache Tomcat in Eclipse IDE](https://www.youtube.com/watch?v=Qd3iwJ3pMgo&feature=youtu.be)

**Life Cycle of a Servlet**

### **Life Cycle of a Servlet**

A **servlet container** manages the life cycle of a servlet. [Servlet](https://docs.oracle.com/javaee/1.4/api/javax/servlet/Servlet.html) is an interface defined in **javax.servlet** package. A servlet container uses the Servlet interface to understand a specific Servlet object and manage it.

There are three life cycle methods of a Servlet :

* init()
* service()
* destroy()

The steps involved in the servlet life cycle are listed below:

**Step-1 : Loading of Servlet**

When the application server (e.g. Apache Tomcat) starts up, the servlet container deploys and loads all the servlet classes.

**Step-2 : Creating an instance of Servlet**

Once all the Servlet classes are loaded, the servlet container creates only one instance for each servlet class. All requests to the servlet are executed on that same servlet instance. Some application servers can create multiple instances of a servlet to handle a high volume of incoming requests, but that is not the default behavior.

**Step-3 : Invoke init() method once**

Once all the servlet classes are instantiated, the init() method is invoked for each instantiated servlet. The init() method is used to initialize the servlet. The init() method is called only once.

**The init() method signature:**

**public** **void** **init**() **throws** **ServletException** {

}

**Step-4 : Invoke service() method repeatedly for each client request**

The servlet container calls the service method each time a request for the servlet is received. The service() method determines the type of Http request (GET, POST, PUT, DELETE, etc.) also calls doGet(), doPost(), doPut(), doDelete(), etc. methods as appropriate.

**The service() method signature:**

**public** **void** **service**(**ServletRequest** req, **ServletResponse** resp) **throws** **ServletException**, **IOException** {

}

**Step-5 : Invoke destroy() method once**

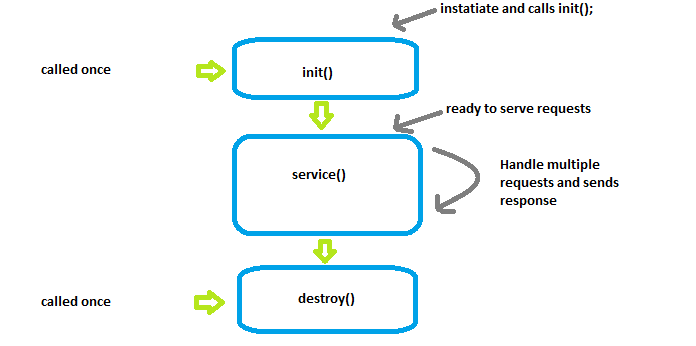
The destroy() method is called only once at the end of the a servlet's life. The servlet container calls this method before removing the servlet instance from the service.

**The destroy() method signature:**

**public** **void** **destroy**() {

}

**Life Cycle of a Servlet:**



**Servlet API**

Servlet API

The [Servlet API](https://docs.oracle.com/javaee/6/api/javax/servlet/package-tree.html) provides interfaces and classes that are required to build servlets. These interfaces and classes represented in two packages:

1. **java.servlet** package - used by the servlet or web container.
2. **javax.servlet.http** package - used for handling http requests.

Servlet Class Hierarchy:

The Servlet interface is the root interface of the servlet class hierarchy. The GenericServlet class implements Servlet, ServletConfig, and Serializable interfaces. The HttpServlet class extends the GenericServlet class and implements the Serializable interface. It provides HTTP methods such as doGet, doPost, doHead, doTrace, etc.

Java Servlet Class Hiearchy:

Servlet ServletConfig Serializable {Built-in Interfaces}

| | |

-----------------------------------

|

V

GenericServlet {Built-in Class}

|

V

HttpServlet {Built-in Class}

|

V

UserDefinedServlet {User defined servlet Class}

The user defined servlet class is created by implementing the *Servlet* interface, usually by extending the *GenericServlet* class or (more commonly) the *HttpServlet* class.

In order to initialize a Servlet, a server application loads the user-defined servlet class and creates an instance. Then it calls the Servlet’s init (ServletConfig config) method. Since the init() method is run once, it stores the initial parameters or configuration information in the *ServletConfig* object. This information can be retrieved later by calling the Servlet’s getServletConfig() method. This is implemented in the *GenericServlet* class definition. The *ServletConfig* object contains Servlet parameters and a reference to the Servlet’s *ServletContext*. The **ServletContext** is an interface which helps to communicate with other servlets. Then, the service (ServletRequest request, ServletResponse response) method is called for every request to the Servlet. When the Servlet needs to be unloaded the destroy() method is called.

**Servlet Declarations and Mapping**

Declaring and mapping servlets

To configure a servlet in the web.xml file,

1. Configure the servlet using the <servlet> element.
2. Map the servlet to a URL or URL pattern using the <servlet-mapping> element.

The <servlet> element is used to declare the servlet name, the fully-qualified class name of the servlet, and any initialization parameters. The name for every servlet must be unique across the deployment descriptor.

The <servlet-mapping> element is used to specify a URL pattern, and the name of the servlet which handles requests whose URL matches the given pattern. The URL pattern uses an asterisk (\*) at the beginning or end of the pattern to indicate zero or more of any character.

**Example:** A simple *web.xml* file which has <servlet> and <servlet-mapping> for *servlet1*.

<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee

http://xmlns.jcp.org/xml/ns/javaee/web-app\_3\_1.xsd"

version="3.1">

<servlet>

<servlet-name>servlet1</servlet-name>

<servlet-class>com.revature.MyFirstServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>servlet1</servlet-name>

<url-pattern>/\*</url-pattern>

</servlet-mapping>

</web-app>

**Deployment descriptor (web.xml)**

Deployment descriptor

Java web applications use a **deployment descriptor file** to define the URLs that map to servlets, and to determine which URLs require authentication and additional information.

A deployment descriptor file specifies the classes, resources, and configuration of the application and how the web server uses them to serve HTTP requests.

The deployment descriptor is a file named **web.xml**. It resides within the app's WAR beneath the WEB-INF/ directory.The root element of the web.xml file is <web-app>.

The **web.xml** file defines mappings between URL paths and the servlets that will handle requests with those paths. The application server uses this configuration to find the servlet that handles a given request, and calls the servlet method that corresponds to the HTTP request method used.

To map a URL to a servlet, you declare the servlet with the <servlet> element, then define a mapping from a URL path to a servlet declaration with the <servlet-mapping> element.

Below we have simple **web.xml** file that maps all URL paths (/\*) to the servlet class mysite.server.MyFirstServlet.

<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee

http://xmlns.jcp.org/xml/ns/javaee/web-app\_3\_1.xsd"

version="3.1">

<servlet>

<servlet-name>servlet1</servlet-name>

<servlet-class>com.revature.MyFirstServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>servlet1</servlet-name>

<url-pattern>/\*</url-pattern>

</servlet-mapping>

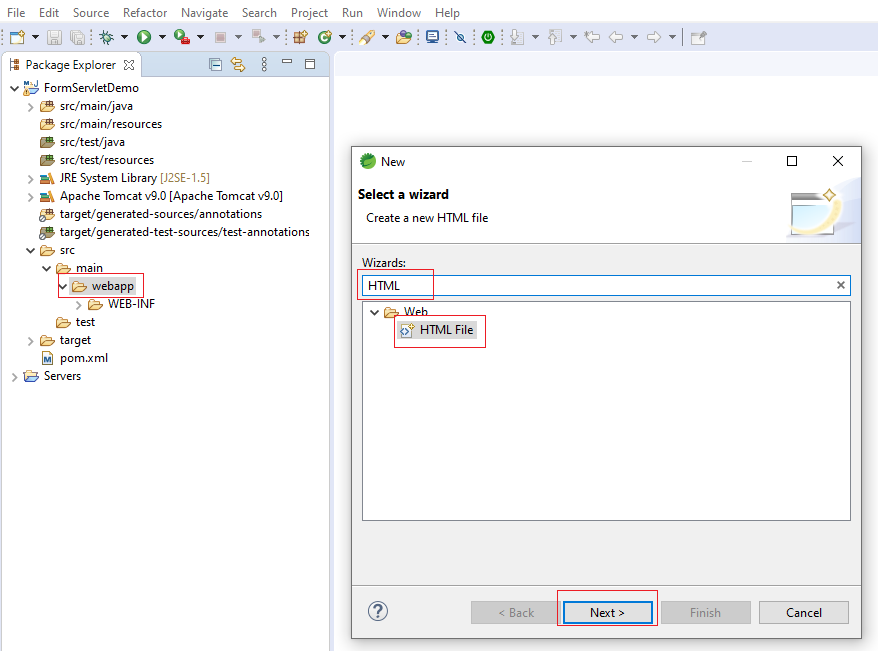
</web-app>

**Form Processing**

Handling form data from the request

In this section, we'll discuss how to handle HTML form data from on the server-side with a Java Servlet. When a user fills in the fields of a form and submits it, the servlet processes the request based on the submitted data and sends a response back to the client.

* Right click on the **webapp** folder and click New -> HTML File. If you do not see the option to create a new HTML file, click Other and search for HTML.



* Name the new file add.html and click Finish.

To make the form work with a Java servlet, we need to specify the following attributes for the <form> tag:

* **method="post" or method="get"** : to send the form data as an HTTP POST or HTTP GET request to the server.
* **action= "URL of the servlet"**: specifies URL of the servlet which is responsible for handling form data.

**For example**,'*add.html*' uses a <form> element for adding two numbers:

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Addition</title>

</head>

<body>

<form method="get" action="add">

Enter Number 1: <input name="num1" /> <br/>

Enter Number 2: <input name="num2" /> <br/>

<input type="submit" value="Add">

</form>

</body>

</html>

We have to configure the **web.xml** file, so that the servlet container sends the form's submit request to the corresponding servlet.

**For example**,'*web.xml*' file maps the '*/add*' URL pattern to the '*AddServlet*' class:

<servlet>

<servlet-name>servlet1</servlet-name>

<servlet-class>AddServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>servlet1</servlet-name>

<url-pattern>/add</url-pattern>

</servlet-mapping>

The servlet handles **GET** requests using the doGet() method and **POST** requests using the doPost() method. The servlet uses the HttpServletRequest.getParameter() method to get the value of form parameters like *num1* and *num2*. It does so by using the **name** attribute of the HTML <input> elements in the form.

**For example**, '*AddServlet.class*' handles the form data:

**public** **class** **AddServlet** **extends** **HttpServlet** {

**public** **void** **doGet**(**HttpServletRequest** req,**HttpServletResponse** res) **throws** **ServletException**,**IOException**

{

**int** i = **Interger**.**parseInt**(request.**getParameter**("num1"));

**int** j = **Interger**.**parseInt**(request.**getParameter**("num2"));

**int** k = i + j;

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

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

out.**println**("The sum is: " + k);

}

}

Retrieving request parameters from the request

Servlets use the following methods for retrieving request/form parameters from the HTTP request:

* getParameter() method - used to get the value of a specified parameter.
* getParameterValues() method - used to get the multiple values of a specified parameter.(for example checkbox)
* getParameterNames() method - used to get complete list of all parameters.

Example *HTML form*:

<p> Enter the following details</p>

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

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

Age: <input type="password" name="age"/>

Gender:

<input type="radio" name="gender" value="male" />Male

<input type="radio" name="gender" value="female" />Female

Speaking language:

<input type="checkbox" name="language" value="english" />English

<input type="checkbox" name="language" value="french" />French

<input type="submit" />

</form>

The *web.xml* file:

<servlet>

<servlet-name>servlet2</servlet-name>

<servlet-class>MyServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>servlet2</servlet-name>

<url-pattern>/details</url-pattern>

</servlet-mapping>

*Servlet* example for retrieving request parameters from the request using the getParameter() and getParameterValues() methods:

**public** **class** **MyServlet** **extends** **HttpServlet** {

**public** **void** **doGet**(**HttpServletRequest** req,**HttpServletResponse** res) **throws** **ServletException**,**IOException**

{

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

**int** age = **Interger**.**parseInt**(request.**getParameter**("age"));

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

**String** languages[] = request.**getParameterValues**("language");

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

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

out.**println**("Your details:")

out.**println**("Name: " + name);

out.**println**("Age: "+ age);

out.**println**("Gender: "+ gender);

**if** (languages != null) {

out.**println**("Languages are: ");

**for** (**String** lang : languages) {

out.**println**(lang);

}

}

}

}

**ServletConfig and ServletContext parameters**

### **ServletConfig**

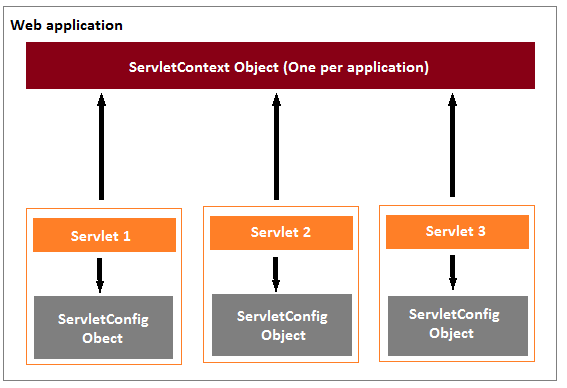
ServletConfig is an object created by the Servlet Container, used to pass initial parameters or configuration information to a particular servlet during initialization. The <servlet> XML element in the deployment descriptor (web.xml) has a subelement called <init-param> used to pass parameters to a servlet from the web.xml file. The ServletConfig object is returned by the getServletConfig() method of HttpServlet, and so the XML properties added to ServletConfig are only shared with the single servlet indicated.

### **ServletContext**

ServletContext is the object created by the Servlet Container to share initial parameters or configuration information to all servlets and other components. The <context-param> element used to declare the parameters of ServletContext. It present outside the <servlet> element and inside the <web-app> element. This object is returned by the getServletContext() method of HttpServlet.

The <param-name> and <param-value> used to declare the parameter name and its value.

**ServletConfig vs ServletContext**



### **Example**

Here '**message**' parameter can be accessed only by servlet1. The '**username**' and '**password**' parameters can be accessed by both servlet1 and servlet2.

<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee

http://xmlns.jcp.org/xml/ns/javaee/web-app\_3\_1.xsd"

version="3.1">

<servlet>

<servlet-name>servlet1</servlet-name>

<servlet-class>com.revature.MyFirstServlet</servlet-class>

<init-param>

<param-name>message</param-name>

<param-value>Hello World</param-value>

</init-param>

</servlet>

<servlet-mapping>

<servlet-name>servlet1</servlet-name>

<url-pattern>/FirstServlet</url-pattern>

</servlet-mapping>

<servlet>

<servlet-name>servlet2</servlet-name>

<servlet-class>com.revature.MySecondServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>servlet2</servlet-name>

<url-pattern>/secondServlet</url-pattern>

</servlet-mapping>

<context-param>

<param-name>username</param-name>

<param-value>system</param-value>

</context-param>

<context-param>

<param-name>password</param-name>

<param-value>pass123</param-value>

</context-param>

</web-app>

**Hidden Form Fields**

Hidden Form Fields

Hidden fields can be inserted into webpages by the server for session tracking.  
These fields are not visible directly to the user but can still be viewed using the *view source* option from the browser. Therefore, hidden fields should *not* be used as a form of security.

Hidden fields may be used to send information that is only pertinent to the server, and not the client.

A web server can send a hidden HTML form field along with a unique session ID:

<input type = "hidden" name = "session\_id" value = "65349">

This hidden field is not displayed on the browser but the value is sent to the server when the parent <form> element is submitted.

The server retrieves this hidden form field value using the request.getParameter("session\_id") method in a servlet.

**Request Dispatcher**

## Request Dispatcher

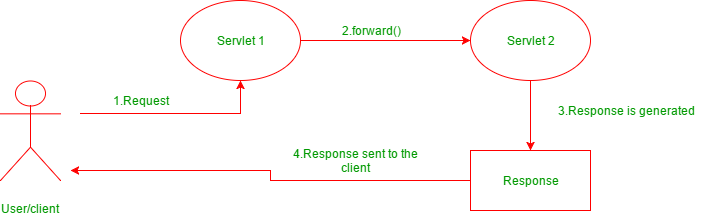
The Request Dispatcher interface defines an object that passes along the client's request to any other resources (servlet, JSP file, or HTML file) on the server.

The servlet container creates the RequestDispatcher object. The getRequestDispatcher(String) method of the **ServletRequest** interface returns the Request Dispatcher object.

[RequestDispatcher interface](https://docs.oracle.com/javaee/7/api/javax/servlet/RequestDispatcher.html) defined in the javax.servlet package provides two methods:

### **1. forward(ServletRequest request, ServletResponse response)**

forward() passes a request from one servlet to another resource on the server. The contents of the request and response are preserved and forwarded to the next resource which will process the data and return the response to the client.



**Example for forward() method:**

//"request" is a HttpServletRequest Object and "Welcome.html" is a resoure name

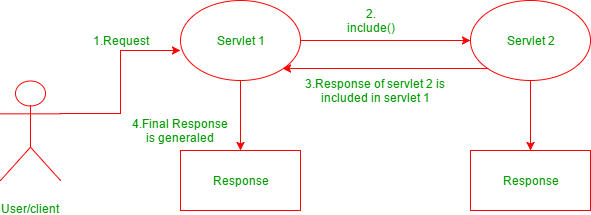
**RequestDispatcher** rd = request.**getRequestDispatcher**("Welcome.html");

//forward the request and response to "Welcome.html" page

rd.**forward**(request,response);

### **2. include(ServletRequest request, ServletResponse response)**

include() does not entirely transfer control over the request and reponse object to the next resource. Instead, this method includes the content of the original resource in the reponse returned to the client. If you include() a servlet or JSP document, the included resource may not change the response code or HTTP headers.



**Example for include() method:**

//"request" is a HttpServletRequest Object and "hello.html" is a resoure name

**RequestDispatcher** rd = request.**getRequestDispatcher**("hello.html");

//includes the response of "hello.html" page in current servlet response

rd.**include**(request,response);

### **References**

* [RequestDispatcher Documentation](https://docs.oracle.com/javaee/7/api/javax/servlet/RequestDispatcher.html)

**Writing plain text to the response Object**

## Writing plain text to the response Object

The Servlet API provides HttpServletResponse interface which extends the ServletResponse interface to assist in sending a response to the client.

The Servlet Container informs the client browser about the type of data in the response before sending it. The Servlet container uses the setContentType() method to set the type of data in the response object. The response data can be in simple plain text format, HTML format, XML format, an image format, etc.

The setContentType(String type) method **sets the content type** of the response being sent to the client before sending the response. Theresponse.getWriter() method returns a **PrintWriter** object, which sends character text to the client.

**Example:**

**public** **class** **HelloWorld** **extends** **HttpServlet** {

**public** **void** **doGet**(**HttpServletRequest** req,**HttpServletResponse** res) **throws** **ServletException**,**IOException**

{

// set the content type as plain text

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

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

out.**println**("Hello World"); // writes a plain text to the response object

}

}

In the above code, the MIME type "text/plain" can be broken down as follows: "text" is known as the **type** and "plain" is known as a **subtype**. A type contains many subtypes. Some of the types used in the setContentType() method are "text/html" used for writing HTML content to the response object, "text/css" for writing CSS content to the response object, etc.

## SendRedirect in servlets

**sendRedirect(String URL)** - This method defined in **HttpServletResponse** interface and used to redirect a response to another resource. It uses the URL to make another request. Therefore, it works at the client-side also can work inside and outside the server.

The RequestDispatcher.forward() method is used to pass the same request to a new destination resource, but the Response.sendRedirect() method is used to send an entirely new request for the destination resource. Any request attributes or parameters from the original request are lost.

**Example for sendRedirect():**

//"response" is a HttpServletRequest Object redirected to the google server.

response.**sendRedirect**("http://www.google.com");

### **Resources**

* [ServletResponse Oracle Documentation](https://docs.oracle.com/javaee/7/api/javax/servlet/ServletResponse.html)
* [PrintWriter Oracle Documentation](https://docs.oracle.com/javase/7/docs/api/java/io/PrintWriter.html)
* [HttpResponse Object Examples](http://tutorials.jenkov.com/java-servlets/httpresponse.html)

Day -3

**Status Codes**

Status Codes

* Informational responses (100–199),
* Successful responses (200–299),
  + 200 OK
  + 201 Created
  + 202 Accepted
  + etc. ...
* Redirects (300–399),
  + 300 Multiple Choices
    - There is more than one valid option the server could respond with. Best to let the client decide in these circumstances and respond with multiple links
  + 301 Moved Permanently
    - URL has changed and the response ought to include the new address
  + 302 Found
    - Resource has temporarily relocated
  + 303 See Other
    - tells the client to send a get request to another URL
* Client errors (400–499),
  + 400 Bad Request
    - the request couldn't be understood something was wrong with the syntax
  + 401 Unauthorized
    - the client needs to be authenticated
  + 403 Forbidden
    - client doesn't have rights to the content -- it's not just a matter of logging in
  + 404 Not Found
    - Can't find the resource requested
  + 405 Method Not Allowed
    - occurs when you use an HTTP method that the server is not configured to support at that particular endpoint
  + 406 Not Acceptable
    - This will occur if the resource the server finds does not fit the criteria of what is acceptable to the user
    - HTTP accept headers are restrictive, so if you say you will only Accept a JPEG and the server has an HTML document that would be considered Not Acceptable.
  + 410 Gone
    - The resource has been removed.
  + 418 I'm a teapot
    - "The server refuses the attempt to brew coffee with a teapot."
  + etc.
* Server errors (500–599).
  + 500 Internal Server Error
  + 503 Service Unavailable
    - service might be down for maintenance
    - The response should indicate when to retry
  + etc.

References

* [HTTP MDN overview](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview)
* [HTTP status-codes](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status)
* **Introduction to REST**
* REST
* Stands for **Representational State Transfer**.
* Defined originally by Roy Fielding in his dissertation in 2000, REST is an *architectural style* that outlines communication between a client and server over the web.
* Essentially for a web service to be RESTful it has to adhere to a set of guidelines or constraints.
* A RESTful server should not retain information about the state of the client.
* Clients communicate with the server through an interface that is standard in that it too follows another set of constraints: "defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state." - Roy Fielding
* In a RESTful system, the server creates an object or resource and returns the *state* of that object (the values contained within the object) when requested by the client.
* **Exposing and Consuming RESTful API endpoints**
* API - Application Programming Interface
* An API is a software intermediary that allows two applications to talk to each other.
* For example, all airline ticket booking applications will use an API exposed by the airline company. Anytime a customer uses any such application and books a flight ticket, the application passes the passenger and flight booking information to the API. The booking API will process the data and book the ticket for the customer and the application will get a success response with booking details in return. The booking applications do not know and need not to know how the API works internally. All they are required to do is pass the booking information in a well-defined format to the API and wait for the response. Similarly, any application/mobile app can use an API or expose an API to other software.
* A RESTful API (also known as a RESTful WebService) is a web service implemented using HTTP protocol and the [REST Architectural Constraints](https://app.revature.com/).
* RESTful APIs typically are built on top of HTTP.
* Example - Exposing / Consuming REST API endpoints
* Here, we setup a JSON server and publish a REST API. [JSON Server](https://www.npmjs.com/package/json-server) is a simple project that helps you to setup a REST API with CRUD operations very fast. Following steps demonstrate how we can interact with the REST API through the HTTP endpoints.
* **Step 1** - Install the JSON Server using the npm install -g json-server command. JSON Server is available as a NPM package.
* **Step 2** - Create a new JSON file with name db.json. This file contains the JSON data which will be exposed by the REST API.
* Below, the db.json file consists of one employee object which has three data sets assigned. Each employee object is consisting of four properties: id, first\_name, last\_name and email.
* {
* "employees": [
* {
* "id": 1,
* "first\_name": "Sebastian",
* "last\_name": "Eschweiler",
* "email": "sebastianE@gmail.com"
* },
* {
* "id": 2,
* "first\_name": "Steve",
* "last\_name": "Palmer",
* "email": "steveP@gmail.com"
* },
* {
* "id": 3,
* "first\_name": "Ann",
* "last\_name": "Smith",
* "email": "annS@gmai;.com"
* }
* ]
* }
* **Step 3** - Running the JSON Server using the json-server --watch db.json command. We need to pass over the file name containing our JSON structure - db.json- as a parameter. By using watch parameter we’re making sure that the server is started in watch mode which means that it watches for file changes and updates the exposed API accordingly. Now, our JSON server is up and running in the port 3000.
* The following HTTP endpoints are created automatically by the JSON server:
* GET /employees
* GET /employees/{id}
* POST /employees
* PUT /employees/{id}
* PATCH /employees/{id}
* DELETE /employees/{id}
* NOTE: You have to download the [Postman](https://www.postman.com/downloads/) before testing the REST API Endpoints With Postman.
* **Step 4** - Now, we are going to test the REST API Endpoints exposed by the JSON Server with Postman.
* Send the HTTP GET request to <http://localhost:3000/employees> which shows the following result:
* When you send the HTTP GET request with id parameter to <http://localhost:3000/employees/1>, you able see the following result:
* When you send the HTTP POST request to <http://localhost:3000/employees> with JSON data on the Body, you able see the following result:
* .
* Similarly, you can try the remaining HTTP request methods. For all the POST, PUT, PATCH or DELETE requests you make, changes will be automatically saved to the db.json file.

**Jackson Library**

## Working with Jackson API

The [Jackson API](https://en.wikipedia.org/wiki/Jackson_(API)) is used to convert Java objects into JSON format to send in an HTML response. The JSON format is commonly used for transmitting data between a server and a client in a web application.

In this section, we'll discuss how we convert java objects into JSON using the Jackson API and send it in a servlet response.

### **Jackson ObjectMapper**

The ObjectMapper API is used for data-binding. It uses reader/writer methods to perform the conversion between Java objects and JSON.

* **writeValue() method** - used to convert Java object to JSON.

Example:

**ObjectMapper** mapper = **new** **ObjectMapper**();

**User** user = **new** **User**();

//Object to JSON in String

**String** jsonInString = mapper.**writeValueAsString**(user);

* **readValue() method** - used to convert JSON to Java object.

Example:

**ObjectMapper** mapper = **new** **ObjectMapper**();

**String** jsonInString = "{'name' : 'Adam'}";

//JSON from String to Object

**User** user = mapper.**readValue**(jsonInString, **User**.**class**);

### **Example**

* First, we need to add the Jackson dependency in our pom.xml file.

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-core</artifactId>

<version>2.11.0</version>

</dependency>

* Then, we create a POJO class for converting this class object into JSON.

**public** **class** **User**

{

**private** **Integer** id;

**private** **String** firstName;

**private** **String** lastName;

//Getters and setters

}

* Servlets sends the JSON output as a response to the client using **ObjectMapper**.

**public** **class** **HomeServlet** **extends** **HttpServlet** {

**protected** **void** **doGet**(**HttpServletRequest** request, **HttpServletResponse** response) **throws** **ServletException**, **IOException** {

//converts the User object into JSON and sends the JSON data to the Client as a response

response.**getWriter**().**write**(

**new** **ObjectMapper**().**writeValueAsString**(

**new** **User**("1234", "John", "Adams")));

}

}

### **References**

* [ObjectMapper Documentation](https://fasterxml.github.io/jackson-databind/javadoc/2.7/com/fasterxml/jackson/databind/ObjectMapper.html)

**Exception and Error Handling**

## Exception and Error Handling

The Servlet API allows us to handle exceptions and errors caused during the execution of a servlet and still send a useful response to the user. The **deployment descriptor** file just needs to be configured to handle the exceptions/errors thrown by a servlet.

The **<error-page>** element used to specify the invocation of servlets in response to certain **exceptions** or **HTTP status codes**. The following elements are used within an <error-page> element to handle error or exception:

* <error-code> - used to specify a valid HTTP error code. For example, 404, 403, 500, etc.
* <exception-type> - used to specify a fully-qualified class name of a Java exception type. For example, *javax.servlet.ServletException,java.io.IOException*, *java.lang.RuntimeException*, etc.
* <location> - used to specify the location of the resource which is displayed to the user in case of an error. This might be a servlet, an HTML page, a JSP page, or something else.

**Request Attributes for Errors/Exceptions:**

Before the servlet container invokes the servlet to handle the exception, it sets some attributes in the request to get useful information about the exception. Some of these are:

* javax.servlet.error.status\_code
* javax.servlet.error.servlet\_name
* javax.servlet.error.exception
* javax.servlet.error.request\_uri
* javax.servlet.error.exception\_type
* javax.servlet.error.message

Note that while these appear to be fully-qualified class names, they are neither packages, classes, nor variables. They are the names of attributes, and are treated as Strings.

### **Example**

The web.xml file for mapping an exception to a servlet.

<error-page>

<exception-type>java.lang.ArithmeticException</exception-type>

<location>/errorHandler</location>

</error-page>

The Servlet class which throws an exception:

@WebServlet(name = "testServlet", urlPatterns = {"/test"}, loadOnStartup = 1)

**public** **class** **TestServlet** **extends** **HttpServlet** {

**protected** **void** doGet (**HttpServletRequest** request, **HttpServletResponse** resp)**throws** **ServletException**, **IOException** {

**int** i = 1 / 0;

}

}

Here, we use [@WebServlet](https://docs.oracle.com/javaee/7/api/javax/servlet/annotation/WebServlet.html) annotation to declare a servlet.

The servlet class that handles the error caused by the above servlet:

@WebServlet(name = "errorHandlerServlet", urlPatterns = {"/errorHandler"}, loadOnStartup = 1)

**public** **class** **ErrorHandlerServlet** **extends** **HttpServlet** {

@Override

**protected** **void** doGet (**HttpServletRequest** request, **HttpServletResponse** response) **throws** **ServletException**, **IOException** {

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

**Exception** exception = (**Exception**) req.**getAttribute**("javax.servlet.error.exception");

**Class** exceptionClass = (**Class**) req.**getAttribute**("javax.servlet.error.exception\_type");

**Integer** status\_code = (**Integer**) req.**getAttribute**("javax.servlet.error.status\_code");

**String** errorMessage = (**String**) req.**getAttribute**("javax.servlet.error.message");

**String** requestUri = (**String**) req.**getAttribute**("javax.servlet.error.request\_uri"

**String** servletName = (**String**) req.**getAttribute**("javax.servlet.error.servlet\_name");

out.**println**("Exception: " + exception);

out.**println**("Exception Type: " + exceptionClass);

out.**println**("HttpError Status code: " + status\_code);

out.**println**("ErrorMessage: " + errorMessage);

out.**println**("Request URI: " + requestUri);

out.**println**("Servlet Name: " + servletName);

}

}

### **References**

* [Further Reading on Servlet Exception Handling](https://www.tutorialspoint.com/servlets/servlets-exception-handling.htm#:~:text=If%20the%20web%20application%20throws,explain%20you%20the%20basic%20concept.)

**HttpSession API**

## HttpSession API

The Servlet API provides **HttpSession Interface**, which provides a way to identify a user and to store information about that user. For the client's first request, the Servlet Container generates a **unique session ID** and gives it back to the client with a response. Thereafter, the client sends the session ID with each request to the server.

The **getSession()** method of the HttpServletRequest object returns a user's session. Any servlet can access the HttpSession object using getSession() method.

Example for creating the HttpSession object:

**protected** **void** **doPost**(**HttpServletRequest** request,**HttpServletResponse** response)**throws** **ServletException**, **IOException** {

**HttpSession** session = request.**getSession**();

}

The commonly used HttpSession Interface methods are listed below:

* **setAttribute(key,object)** - used to bind an object to the session, using the key specified.
* **getAttribute(String)** - used to retrieve a specific saved object from the session object, using its key.
* **removeAttribute(key)** - used to remove the object bound with the specified key from the session.
* **invalidate()** - destorys the session.
* **getId()** - returns the unique ID assigned to the session.
* **getCreationTime()**- returns the time when the session was created
* **getLastAccessedTime()** - returns the last time the client sent a request associated with the session
* **getMaxInactiveInterval()** - returns the maximum time interval, in seconds.
* **setMaxInactiveInterval(int interval)** - Specifies the time, in seconds,after servlet container will invalidate the session.

## ****Example:****

Create two servlets: a **SourceServlet** and a **TargetServlet** which will process data submit through a form in an HTML file called **user.html**.

1. Upon submitting the userName in the **user.html** form, the \*\*SourceServlet will create a session object.
2. The **SourceServlet** will use the session object to setAttribute to the user, and then use the PrintWriter to produce a hyperlink to the **TargetServlet**.
3. The **TargetServlet** can retrieve the user by once again instantiating a session and then getting the "user" attribute.
4. By using PrintWriter, the **TargetServlet** will print username that the client initially input in the **user.html** form as well as more information about the session.

### **Create the HTML file user.html**

<!DOCTYPE html>

<html>

<head>

<meta charset="ISO-8859-1">

<title>Insert title here</title>

</head>

<body>

<h1>Enter User Name:</h1>

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

User Name :<input name="userName">

<input type="submit" value="send" name="submitButton">

</form>

</body>

</html>

### **Create SourceServlet.java**

@WebServlet("/sourceServlet")

**public** **class** **SourceServlet** **extends** **HttpServlet** {

**private** **static** **final** **long** serialVersionUID = 1L;

**protected** **void** **doPost**(**HttpServletRequest** request, **HttpServletResponse** response) **throws** **ServletException**, **IOException** {

**String** username = request.**getParameter**("userName");

// instantiate a session objection

**HttpSession** session = request.**getSession**();

// set an attribute that can be retrieved by the next servlet

session.**setAttribute**("user", username);

// create a hyperlink to go to the next servlet which will process the request

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

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

out.**println**("<a href='targetServlet'>Click Here to get the UserName</a>");

}

}

Here we are using @WebServlet annotations to map our servlets.

Notice that the annotation for our sourceServlet correlates with the action= attribute of our user.html form.

### **Create TargetServlet.java**

@WebServlet("/targetServlet")

**public** **class** **TargetServlet** **extends** **HttpServlet** {

**private** **static** **final** **long** serialVersionUID = 1L;

// this is called when the hyperlink from SourceServlet is clicked

**protected** **void** **doGet**(**HttpServletRequest** request, **HttpServletResponse** response) **throws** **ServletException**, **IOException** {

// instantiate the session object and use the .getSession() method on the request

**HttpSession** session = request.**getSession**();

// retrieve the attribute from the session

**String** username = **(String)**(session.**getAttribute**("user"));

**String** sessionId = session.**getId**();

**long** creationTime = session.**getCreationTime**();

**long** lastAccessedTime = session.**getLastAccessedTime**();

**Date** createDate= **new** **Date**(creationTime);

**Date** lastAccessedDate= **new** **Date**(lastAccessedTime);

// print the retrieved attribute using the PrintWriter

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

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

out.**println**("<h1>Username is: " + username + " </h1>");

out.**println**(" Your Session Infomation: <br/>");

out.**println**("ID: " + sessionId + "<br/>");

out.**println**("Session Created Date: " + createDate + "<br/>");

out.**println**("Session Created Time: " + creationTime + "<br/>");

out.**println**("Last Accessed Date : " + lastAccessedDate + "<br/>");

out.**println**("Last Accessed Time: " + lastAccessedTime + "<br/>");

}

}

Right click on user.html and click Run On Server.

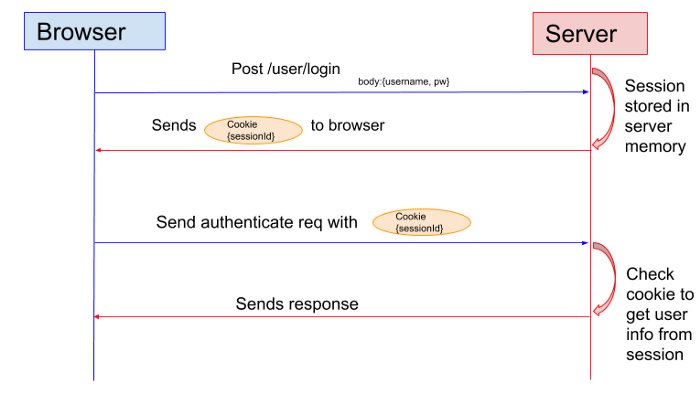
You should be able to enter a username and retrieve that information in your **TargetServlet** by accessing the data stored in the associated session.

**Session Management in Servlets**

## Session Management in servlets

The HTTP protocol is a **stateless protocol**, which means no client information stored in the server. The server considers every request form the same client as a new, independent request. However, this means that a server cannot keep a user "logged in" naturally.

Instead, the client will have to re-identify itself with each request it sends. One workaround is to have the server create a **session** for each client request, which the client can re-associate itself with in each request by sending the **session ID**.



**For example**, When a user logs in to a website, a token proving a successful login will be associated with their session, and the session ID will be returned to the client. Whatever web page user visits after logging in, their browser will send that session ID with each request, and the server can check that session to validate that the user was successfully authenticated.

Session tracking is a mechanism that **servlets use to maintain state** about a series of requests from the same user across a period of time.

A session stores the **unique identification information about the client** that we can get for all requests that client makes. There are four different techniques used by the Servlet application for session management.

* Cookies
* Hidden form fields
* URL Rewriting
* The HttpSession API

### **References**

* [Session Management Further Reading](https://www.geeksforgeeks.org/session-management-in-http/)

**URL Rewriting**

## URL Rewriting

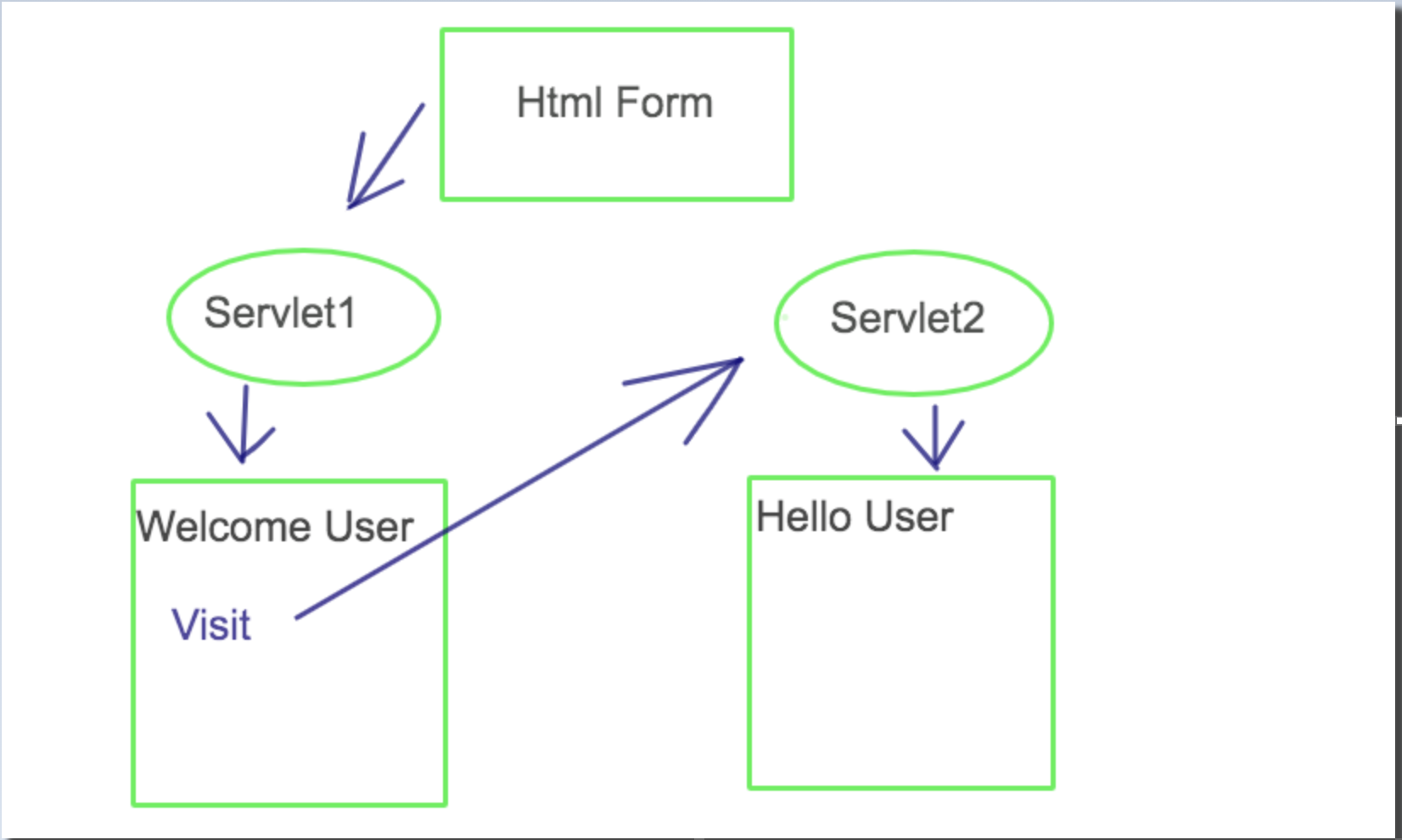
URL Rewriting is a process by which a **unique session ID gets appended to each request URL**, so the server can identify the user session.

URL Rewriting maintains the session and works even the user's browser doesn't support cookies. This makes it one of the ways in which we can provide a unique id in request and response, alongside [implementing the Session Interface](https://gitlab.com/revature_training/servlet-team/-/blob/master/modules/session-management/HttpSession.md) and [Cookies Management](https://gitlab.com/revature_training/servlet-team/-/blob/master/modules/session-management/cookies.md).

## Example

We will create an **HTML page** to capture a username value from the client, a **FirstServlet** which will print the username, and then provide the url pass control to **SecondServlet** using url rewriting.

#### **Program Flow**



### **Create index.html Form**

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>HTML Form</title>

</head>

<body>

<form action="FirstServlet" method="get">

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

<input type="submit" value="Submit" />

</form>

</body>

</html>

### **Create FirstServlet.java**

@WebServlet("/FirstServlet")

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

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

{

**try** {

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

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

// request.getParameter takes the value from index.html file where name is "userName"

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

out.**print**("Welcome " + n);

/\*\* url rewriting is used for creating a session - it will redirect you to SecondServlet page

Notice that we have set the query parameter ?uname equal to the userName parameter we retrieved from the request

\*\*/

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

out.**close**();

}

**catch** (**Exception** e) {

**System**.out.**println**(e);

}

}

}

We appply URL rewriting in the FirstServlet when we pass control to the SecondServlet.

out.**print**("<a href='SecondServlet?uname=" + n + "'>pass control to secondServlet</a>");

### **Create SecondServlet.java**

@WebServlet("/SecondServlet")

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

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

{

**try** {

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

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

// use request.getParameter() to get the value from the url that we have rewritten in FirstServlet

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

out.**print**("Hello " + n);

out.**close**();

}

**catch** (**Exception** e) {

**System**.out.**println**(e);

}

}

}

The benfit of URL rewriting is that it doesn’t depend upon cookies and will work whether cookies are enabled or disabled. Extra form submission is not required on all pages.

### **References**

* [Further reading on URL Rewriting](https://www.geeksforgeeks.org/url-rewriting-using-java-servlet/)

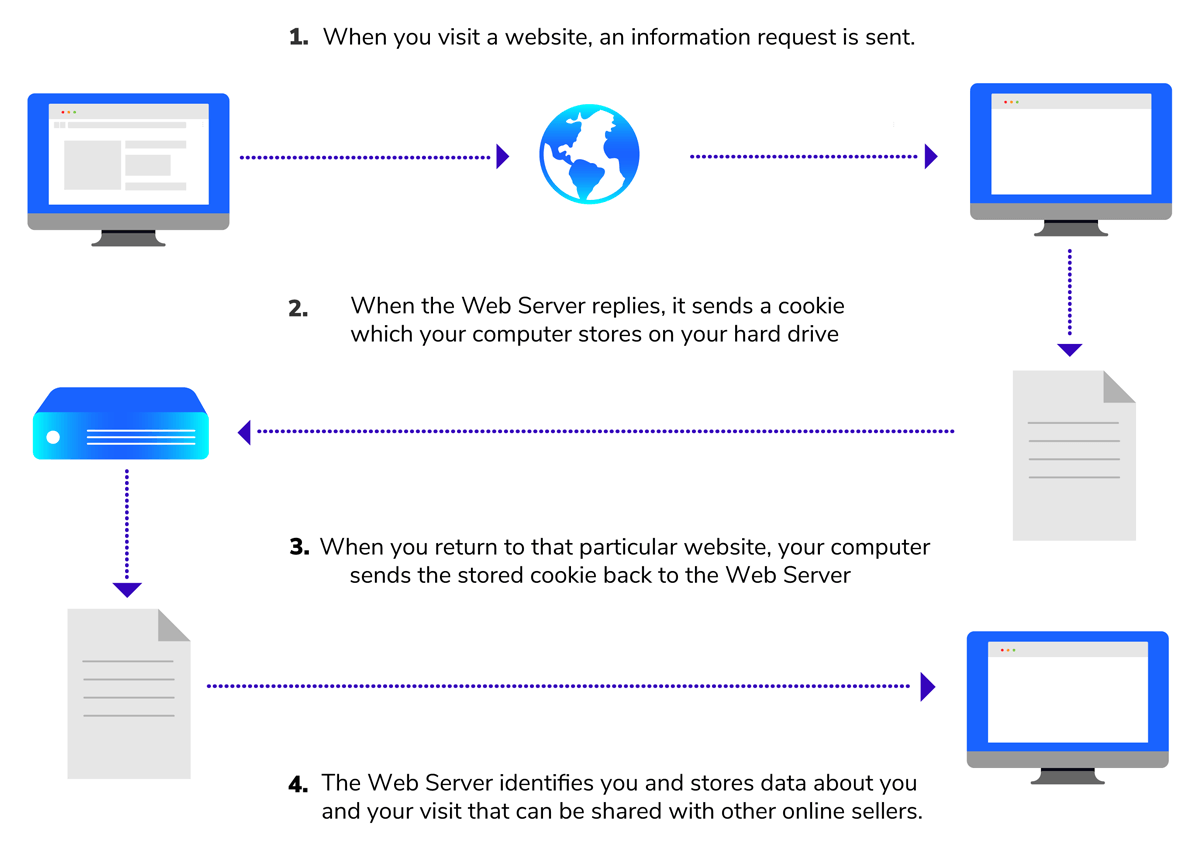
**Cookies**

## Cookies

A cookie is a key-value pair of information sent by the server to the client, which the client will store. The client (usually a web browser) can send this cookie in the HTTP request header for all subsequent requests until the cookie becomes invalid.

The Servlet container checks the request header for cookies, most commonly to get the session information from the cookie, which it uses it to retrieve the associated session data stored in the server.

We can classify the cookies into two types based on their expiry time:



* **Non-persistent cookie** - Cookie becomes expired when the user closes the browser.
* **Persistent Cookies** - Cookie expires only if the user logs out of the website. The cookie is stored on the browser even the user closes the browser each time.

### **Creating Cookies with Servlets**

To send cookies to the client, we need to create a Cookie object, set the maximum age for the cookie, and place the cookie in the HTTP response header. The Cookie(String name, String value) constructor defined in the **javax.servlet.http.Cookie** class can be used to create a cookie with a specified name and value. We can use the setMaxAge() method to set the maximum age for the particular cookie in seconds. We can use the response.addCookie() method to place the cookie in the HTTP response header.

**Example:**

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

//Creating a Cookie object

**Cookie** cookie = **new** **Cookie**("name","Adam");

// Set expiry date after 24 Hrs (86,400seconds)

cookie.**setMaxAge**(86400);

// Add the cookie in the response header

response.**addCookie**(cookie);

}

### **Reading Cookies with Servlets**

To read cookies, We need to create an array of javax.servlet.http.Cookie objects by calling the getCookies() method of HttpServletRequest. Then the getName() and getValue() methods used to access each cookie and associated value.

**Example:**

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

//getting all the cookies

**Cookie** cookies[]=request.**getCookies**();

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

**for**(**Cookie** c : cookies){

out.**println**("Name: "+c.**getName**()+" & Value: "+c.**getValue**());

}

}

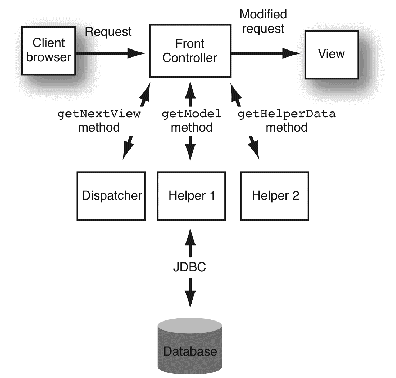
### **References**

* [Further Reading on Cookies](https://www.geeksforgeeks.org/javax-servlet-http-cookie-class-java/)

**Front Controller Design Pattern**

## Front Controller Design Pattern

The front controller design pattern provides a **single handler** for all the incoming requests for a resource in an application, and then **dispatches** the requests to the appropriate secondary handler for that type of request. The front controller may use other **helper** APIs to achieve the dispatching mechanism.



**Front Controller** - The Front controller is a single entry point for all requests, and routes incoming user requests. It delegates to a dispatcher to perform action and view management.

**Dispatcher** - A dispatcher is responsible for the action and view management, including locating and routing to the specific actions that will service a request, and finding the appropriate view.

**Helper** - We use Helper classes to break out specific features and make the application easier to build and maintain. They can be used for the retrieval of content, validation of user-entered information, processing of business logic, and data processing.

**View** - A view represents and displays information to the client - think an HTML/CSS/JS page. The view retrieves information from model objects. These model objects can be passed to the view from a Front Controller servlet through a request attribute, or by placement in the web application's session data.

The **benefits** of using the front controller design pattern:

* It provides centralized control for all requests, which helps for user tracking and security.
* It improves manageability, reusability, and role separation.

**The Program flow of a Front Controller within a Web Application is as follows:** The Front Controller uses the Dispatcher to delegate and identify which Helper can process the request and return the correct View.

### **Example**

We will create a basic web application that allows a Customer to login to a basic login.html page.

The Web Application will have several packages to store our Models, Views, and Controllers.

1. Create a Customer.java class in a package called com.revature.model

**package** com.revature.model;

/\* Main Customer POJO (bean) needed for example \*/

**public** **class** **Customer** {

**private** **int** id;

**private** **String** firstName;

**private** **String** lastName;

**private** **String** username;

**private** **String** password;

**public** **Customer**() {

**this**.firstName = "";

**this**.lastName = "";

**this**.username = "";

**this**.password = "";

}

**public** **Customer**(**int** id) {

**this**();

**this**.id = id;

}

**public** **Customer**(**String** username, **String** password) {

**this**();

**this**.username = username;

**this**.password = password;

}

**public** **Customer**(**int** id, **String** firstName, **String** lastName, **String** username, **String** password) {

**this**.id = id;

**this**.firstName = firstName;

**this**.lastName = lastName;

**this**.username = username;

**this**.password = password;

}

// getters & setters and toString() method

}

1. Create a **Repository** package com.revature.repository which holds your repositories that follow the Data Access Object (DAO) design pattern. The DAO separates your business logic from your persitence layer (database).

**package** com.revature.repository;

**import** java.util.List;

**import** com.revature.model.Customer;

/\* Contract interface that uses DAO design pattern rules that can be implemented

\* by many types of technologies like JDBC, Hibernate, MongoDB, etc.

\*/

**public** **interface** **CustomerRepository** {

**public** **boolean** **insert**(**Customer** customer);

**public** **boolean** **insertProcedure**(**Customer** customer);

**public** **Customer** **select**(**Customer** customer);

**public** **List**<**Customer**> **selectAll**();

**public** **String** **getCustomerHash**(**Customer** customer);

}

1. Create another class which will implement the CustomerRepository interface and establish connection to your database. Create your ConnectionUtil class in a separate package to follow best practices.

**package** com.revature.repository;

**import** java.sql.CallableStatement;

**import** java.sql.Connection;

**import** java.sql.PreparedStatement;

**import** java.sql.ResultSet;

**import** java.sql.SQLException;

**import** java.util.ArrayList;

**import** java.util.List;

**import** org.apache.log4j.Logger;

**import** com.revature.model.Customer;

**import** com.revature.util.ConnectionUtil;

/\* JDBC implementation for DAO contract for Customers data access \*/

**public** **class** **CustomerRepositoryJdbc** **implements** **CustomerRepository** {

**private** **static** **Logger** logger = **Logger**.**getLogger**(**CustomerRepositoryJdbc**.**class**);

/\*Singleton transformation of JDBC implementation object \*/

**private** **static** **CustomerRepository** customerRepository;

**private** **CustomerRepositoryJdbc**() {}

**public** **static** **CustomerRepository** **getInstance**() {

**if**(customerRepository == null) {

customerRepository = **new** **CustomerRepositoryJdbc**();

}

**return** customerRepository;

}

/\* Regular insert statement for Customer \*/

@Override

**public** **boolean** **insert**(**Customer** customer) {

**try**(**Connection** connection = **ConnectionUtil**.**getConnection**()) {

**int** statementIndex = 0;

**String** command = "INSERT INTO CUSTOMER VALUES(NULL,?,?,?,?)";

**PreparedStatement** statement = connection.**prepareStatement**(command);

//Set attributes to be inserted

statement.**setString**(++statementIndex, customer.**getFirstName**().**toUpperCase**());

statement.**setString**(++statementIndex, customer.**getLastName**().**toUpperCase**());

statement.**setString**(++statementIndex, customer.**getUsername**().**toLowerCase**());

statement.**setString**(++statementIndex, customer.**getPassword**());

**if**(statement.**executeUpdate**() > 0) {

**return** true;

}

} **catch** (**SQLException** e) {

logger.**warn**("Exception creating a new customer", e);

}

**return** false;

}

/\* Insert a customer using the stored procedure we created \*/

@Override

**public** **boolean** **insertProcedure**(**Customer** customer) {

**try**(**Connection** connection = **ConnectionUtil**.**getConnection**()) {

**int** statementIndex = 0;

//Pay attention to this syntax

**String** command = "{CALL INSERT\_CUSTOMER(?,?,?,?)}";

//Notice the CallableStatement

**CallableStatement** statement = connection.**prepareCall**(command);

//Set attributes to be inserted

statement.**setString**(++statementIndex, customer.**getFirstName**().**toUpperCase**());

statement.**setString**(++statementIndex, customer.**getLastName**().**toUpperCase**());

statement.**setString**(++statementIndex, customer.**getUsername**().**toLowerCase**());

statement.**setString**(++statementIndex, customer.**getPassword**());

**if**(statement.**executeUpdate**() > 0) {

**return** true;

}

} **catch** (**SQLException** e) {

logger.**warn**("Exception creating a new customer with stored procedure", e);

}

**return** false;

}

/\* Select Customer based on his username \*/

@Override

**public** **Customer** **select**(**Customer** customer) {

**try**(**Connection** connection = **ConnectionUtil**.**getConnection**()) {

**int** statementIndex = 0;

**String** command = "SELECT \* FROM CUSTOMER WHERE C\_USERNAME = ?";

**PreparedStatement** statement = connection.**prepareStatement**(command);

statement.**setString**(++statementIndex, customer.**getUsername**());

**ResultSet** result = statement.**executeQuery**();

**while**(result.**next**()) {

**return** **new** **Customer**(

result.**getInt**("C\_ID"),

result.**getString**("C\_FIRST\_NAME"),

result.**getString**("C\_LAST\_NAME"),

result.**getString**("C\_USERNAME"),

result.**getString**("C\_PASSWORD")

);

}

} **catch** (**SQLException** e) {

logger.**warn**("Exception selecting a customer", e);

}

**return** **new** **Customer**();

}

/\* Select all customers \*/

**public** **List**<**Customer**> **selectAll**() {

**try**(**Connection** connection = **ConnectionUtil**.**getConnection**()) {

**String** command = "SELECT \* FROM CUSTOMER";

**PreparedStatement** statement = connection.**prepareStatement**(command);

**ResultSet** result = statement.**executeQuery**();

**List**<**Customer**> customerList = **new** **ArrayList<>**();

**while**(result.**next**()) {

customerList.**add**(**new** **Customer**(

result.**getInt**("C\_ID"),

result.**getString**("C\_FIRST\_NAME"),

result.**getString**("C\_LAST\_NAME"),

result.**getString**("C\_USERNAME"),

result.**getString**("C\_PASSWORD")

));

}

**return** customerList;

} **catch** (**SQLException** e) {

logger.**warn**("Exception selecting all customers", e);

}

**return** **new** **ArrayList<>**();

}

/\* Get a customer hash consuming the user defined function we created \*/

@Override

**public** **String** **getCustomerHash**(**Customer** customer) {

**try**(**Connection** connection = **ConnectionUtil**.**getConnection**()) {

**int** statementIndex = 0;

**String** command = "SELECT GET\_CUSTOMER\_HASH(?,?) AS HASH FROM DUAL";

**PreparedStatement** statement = connection.**prepareStatement**(command);

statement.**setString**(++statementIndex, customer.**getUsername**());

statement.**setString**(++statementIndex, customer.**getPassword**());

**ResultSet** result = statement.**executeQuery**();

**if**(result.**next**()) {

**return** result.**getString**("HASH");

}

} **catch** (**SQLException** e) {

logger.**warn**("Exception getting customer hash", e);

}

**return** **new** **String**();

}

}

1. Your ConnectionUtil.java class will look like this:

**package** com.revature.util;

**import** java.sql.Connection;

**import** java.sql.DriverManager;

**import** java.sql.SQLException;

**import** org.apache.log4j.Logger;

/\* Final utility class to obtain connections in a modular way \*/

**public** **final** **class** **ConnectionUtil** {

**private** **static** **Logger** logger = **Logger**.**getLogger**(**ConnectionUtil**.**class**);

/\* Make Tomcat now which database driver to use \*/

**static** {

**try** {

**Class**.**forName**("oracle.jdbc.driver.OracleDriver");

} **catch** (**ClassNotFoundException** e) {

logger.**warn**("Exception thrown adding oracle driver.", e);

}

}

/\* Get connection to JDBC \*/

**public** **static** **Connection** **getConnection**() **throws** **SQLException** {

**String** url = "jdbc:oracle:thin:@myrevaturerds.cgmoq4yzdcov.us-east-1.rds.amazonaws.com:1521:ORCL";

**String** username = "LOGIN\_TEST\_DB";

**String** password = "p4ssw0rd";

**return** **DriverManager**.**getConnection**(url, username, password);

}

}

1. Create a **Service** layer between the controller and the model. Start with a com.revature.service package, and create an Interface called CustomerService.java.

This allows us to encapsulate all of the business logic that could be in the controller. This way, the only purpose of the controller is to forward and control the execution.

**package** com.revature.service;

**import** java.util.List;

**import** com.revature.model.Customer;

**public** **interface** **CustomerService** {

**public** **boolean** **registerCustomer**(**Customer** customer);

**public** **boolean** **registerCustomerSecure**(**Customer** customer);

**public** **List**<**Customer**> **listAllCustomers**();

**public** **Customer** **authenticate**(**Customer** customer);

}

1. Create an implementation of your CustomerService.java Interface in a class called CustomerServiceAlpha.

**package** com.revature.service;

**import** java.util.List;

**import** com.revature.model.Customer;

**import** com.revature.repository.CustomerRepositoryJdbc;

**public** **class** **CustomerServiceAlpha** **implements** **CustomerService** {

/\* Singleton \*/

**private** **static** **CustomerService** customerService = **new** **CustomerServiceAlpha**();

**private** **CustomerServiceAlpha**() { }

**public** **static** **CustomerService** **getInstance**() {

**return** customerService;

}

@Override

**public** **boolean** **registerCustomer**(**Customer** customer) {

**return** **CustomerRepositoryJdbc**.**getInstance**().**insert**(customer);

}

@Override

**public** **boolean** **registerCustomerSecure**(**Customer** customer) {

**return** **CustomerRepositoryJdbc**.**getInstance**().**insertProcedure**(customer);

}

@Override

**public** **List**<**Customer**> **listAllCustomers**() {

**return** **CustomerRepositoryJdbc**.**getInstance**().**selectAll**();

}

@Override

**public** **Customer** **authenticate**(**Customer** customer) {

//Information on the database

**Customer** loggedCustomer = **CustomerRepositoryJdbc**.**getInstance**().**select**(customer);

/\*

\* What we have stored in the database is the Username + Password hash.

\* We can't compare the blank password provided by the user, against the hash.

\* So we have to obtain the hash of the user input.

\*

\* If the hashes are the same, user is authenticated.

\*/

**if**(loggedCustomer.**getPassword**().**equals**(**CustomerRepositoryJdbc**.**getInstance**().**getCustomerHash**(customer))) {

**return** loggedCustomer;

}

**return** null;

}

}

1. Before we create a **Controller** Layer, let's create a package called com.revature.ajax which will hold the a ClientMessage.java that a controller could instantiate and return to the Client.

**package** com.revature.ajax;

**public** **class** **ClientMessage** {

**private** **String** message;

**public** **ClientMessage**() {}

**public** **ClientMessage**(**String** message) {

**this**.message = message;

}

**public** **String** **getMessage**() {

**return** message;

}

**public** **void** **setMessage**(**String** message) {

**this**.message = message;

}

}

1. Next, create your **Controller** layer in a package called com.revature.controller. This will be an interface called CustomerController.java which we will implement later in another controller.

**package** com.revature.controller;

**import** javax.servlet.http.HttpServletRequest;

**public** **interface** **CustomerController** {

/\*\*

\* Registers the user.

\*

\* -> If the method is GET, return the registration view.

\* -> Else, return a message stating that registration was successful, or not.

\*/

**public** **Object** **register**(**HttpServletRequest** request);

/\*\*

\* Get all customers in the database.

\*

\* -> If it's GET with no parameters, then we return the view.

\* -> If it's GET with a paramter, then we return the list of users.

\*/

**public** **Object** **getAllCustomers**(**HttpServletRequest** request);

}

1. Create another controller called CustomerControllerAlpha which implements the CustomerController interface we just creataed.

**package** com.revature.controller;

**import** javax.servlet.http.HttpServletRequest;

**import** com.revature.ajax.ClientMessage;

**import** com.revature.model.Customer;

**import** com.revature.service.CustomerServiceAlpha;

**public** **class** **CustomerControllerAlpha** **implements** **CustomerController** {

**private** **static** **CustomerController** customerController = **new** **CustomerControllerAlpha**();

**private** **CustomerControllerAlpha**() {}

**public** **static** **CustomerController** **getInstance**() {

**return** customerController;

}

@Override

**public** **Object** **register**(**HttpServletRequest** request) {

**if** (request.**getMethod**().**equals**("GET")) {

**return** "register.html";

}

/\* Logic for POST \*/

**Customer** customer = **new** **Customer**(0, request.**getParameter**("firstName"), request.**getParameter**("lastName"),

request.**getParameter**("username"), request.**getParameter**("password"));

**if** (**CustomerServiceAlpha**.**getInstance**().**registerCustomerSecure**(customer)) {

**return** **new** **ClientMessage**("REGISTRATION SUCCESSFUL");

} **else** {

**return** **new** **ClientMessage**("SOMETHING WENT WRONG");

}

}

@Override

**public** **Object** **getAllCustomers**(**HttpServletRequest** request) {

**Customer** loggedCustomer = (**Customer**) request.**getSession**().**getAttribute**("loggedCustomer");

/\* If customer is not logged in \*/

**if**(loggedCustomer == null) {

**return** "login.html";

}

/\* Client is requesting the view. \*/

**if**(request.**getParameter**("fetch") == null) {

**return** "all-customers.html";

} **else** {

/\* Client is requesting the list of customers \*/

**return** **CustomerServiceAlpha**.**getInstance**().**listAllCustomers**();

}

}

}

1. Similarly, we will need to create a LoginController.java interface and implementation to handle login requests.

**package** com.revature.controller;

**import** javax.servlet.http.HttpServletRequest;

**public** **interface** **LoginController** {

/\*\*

\* If the method is GET, it will return the login view.

\*

\* If the method is POST.

\* -> If service layer returns null, we return a message

\* stating that authentication failed.

\*

\* -> Else, it will return the Customer information

\* (and store it in the session).

\*/

**public** **Object** **login**(**HttpServletRequest** request);

/\*\*

\* Invalidates the session and returns the login view.

\*/

**public** **String** **logout**(**HttpServletRequest** request);

}

1. Create the controller implementation in a class called LoginControllerAlpha.

**package** com.revature.controller;

**import** javax.servlet.http.HttpServletRequest;

**import** com.revature.ajax.ClientMessage;

**import** com.revature.model.Customer;

**import** com.revature.service.CustomerServiceAlpha;

**public** **class** **LoginControllerAlpha** **implements** **LoginController** {

**private** **static** **LoginController** loginController = **new** **LoginControllerAlpha**();

**private** **LoginControllerAlpha**() {}

**public** **static** **LoginController** **getInstance**() {

**return** loginController;

}

@Override

**public** **Object** **login**(**HttpServletRequest** request) {

**if**(request.**getMethod**().**equals**("GET")) {

**return** "login.html";

}

**Customer** loggedCustomer = **CustomerServiceAlpha**.**getInstance**().**authenticate**(

**new** **Customer**(request.**getParameter**("username"),

request.**getParameter**("password"))

);

/\* If authentication failed \*/

**if**(loggedCustomer == null) {

**return** **new** **ClientMessage**("AUTHENTICATION FAILED");

}

/\* Store the customer information on the session \*/

request.**getSession**().**setAttribute**("loggedCustomer", loggedCustomer);

**return** loggedCustomer;

}

@Override

**public** **String** **logout**(**HttpServletRequest** request) {

//throw new RuntimeException("Something went wrong");

/\*

\* If session.invalidate() doesn't work for you

\*/

request.**getSession**().**invalidate**();

**return** "login.html";

}

}

1. Create your **Dispatchers** to delegate to particular controllers. This acts as a **Front Controller**. These are **Servlets** that redirect the user with the help of the logic incorporated in your controllers.

First create a RequestHelper.java class which will delegate to your DispatcherServlet (which we will create next).

**package** com.revature.request;

**import** javax.servlet.http.HttpServletRequest;

**import** com.revature.controller.CustomerControllerAlpha;

**import** com.revature.controller.LoginControllerAlpha;

**public** **class** **RequestHelper** {

**private** **RequestHelper**() {}

**public** **static** **Object** **process**(**HttpServletRequest** request) {

**switch**(request.**getRequestURI**()) {

**case** "/FrontController/login.do":

**return** **LoginControllerAlpha**.**getInstance**().**login**(request);

**case** "/FrontController/logout.do":

**return** **LoginControllerAlpha**.**getInstance**().**logout**(request);

**case** "/FrontController/register.do":

**return** **CustomerControllerAlpha**.**getInstance**().**register**(request);

**case** "/FrontController/getAll.do":

**return** **CustomerControllerAlpha**.**getInstance**().**getAllCustomers**(request);

**default**:

**return** "not-implemented.html";

}

}

}

1. Then create your DispatcherServlet.

**package** com.revature.request;

**import** java.io.IOException;

**import** javax.servlet.ServletException;

**import** javax.servlet.http.HttpServlet;

**import** javax.servlet.http.HttpServletRequest;

**import** javax.servlet.http.HttpServletResponse;

**import** com.fasterxml.jackson.databind.ObjectMapper;

**public** **class** **DispatcherServlet** **extends** **HttpServlet** {

**private** **static** **final** **long** serialVersionUID = 5244611926643604805L;

**protected** **void** **doGet**(**HttpServletRequest** request, **HttpServletResponse** response) **throws** **ServletException**, **IOException** {

**Object** data = **RequestHelper**.**process**(request);

/\* If what the controllers return is a String, we forward to an HTML file. \*/

**if**(data **instanceof** **String**) {

**String** URI = (**String**) data;

request.**getRequestDispatcher**(URI).**forward**(request, response);

}

/\* Else, we marshall the given POJO \*/

**else** {

response.**getWriter**().**write**(**new** **ObjectMapper**().**writeValueAsString**(data));

}

}

**protected** **void** **doPost**(**HttpServletRequest** request, **HttpServletResponse** response) **throws** **ServletException**, **IOException** {

**doGet**(request, response);

}

}

1. In your **web.xml**, you can also map to error pages. It will look like this:

<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd" version="2.5">

<display-name>FrontController</display-name>

<welcome-file-list>

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

</welcome-file-list>

<servlet>

<servlet-name>DispatcherServlet</servlet-name>

<servlet-class>com.revature.request.DispatcherServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>DispatcherServlet</servlet-name>

<!-- \* it's a Wildcard (which means "anything") -->

<!-- .something it's called Extended Mapping -->

<url-pattern>\*.do</url-pattern>

</servlet-mapping>

<!-- ERROR PAGES -->

<!-- For 500 (Exceptions Thrown) -->

<error-page>

<exception-type>java.lang.Throwable</exception-type>

<location>/oops.html</location>

</error-page>

<!-- For a specific status code -->

<error-page>

<error-code>404</error-code>

<location>/404.html</location>

</error-page>

<!-- Session Objects will expire in 30 minutes -->

<session-config>

<session-timeout>30</session-timeout>

</session-config>

</web-app>

1. You should have a login.html, register.html, all-customers.html, 404.html, and an error page titled oops.html for any exceptions (as mapped in our **web.xml** page).

### **References**

* [Further Reading on FrontController Design Pattern](https://www.geeksforgeeks.org/front-controller-design-pattern/)

**MVC Design pattern**

## MVC Design pattern

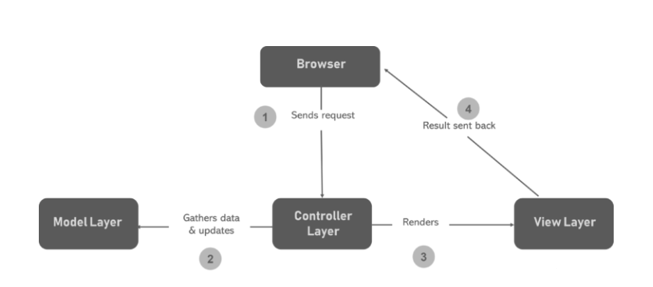
The **M**odel **V**iew **C**ontroller Design pattern is an architectural pattern, used to design and create user interfaces and the structure of an application.

This pattern divides the application into three parts that are dependent and connected.

**Model** - Model used to represent the business layer of the application. It's not involved in the UI or presentation of the application. The model is the data layer which defines the business logic of the system and also represents the state of the application. The model objects retrieve and store the state of the model in a database.

**View** - The view presents the model’s data to the user. The view is a presentation layer that represents the user interface. It displays the data fetched from the model layer by the controller and presents the data to the user whenever requests.

**Controller** - A controller is an intermediary between the model & a view. It receives the user requests from the view layer and processes them. The requests are then sent to model for data processing. Once they are processed, the data is again sent back to the controller and then displayed on the view.



**Advantages** - Multiple developers can simultaneously work on the model, controller and views. The MVC pattern enables a logical grouping of related actions on a controller together. The views for a specific model are also grouped. Models can have multiple views.

### **References**

* [MVC Example](https://www.geeksforgeeks.org/mvc-design-pattern/)
* [Oracle MVC Documentation](https://www.oracle.com/technical-resources/articles/javase/mvc.html)

**Request and Response**

HTTP Background

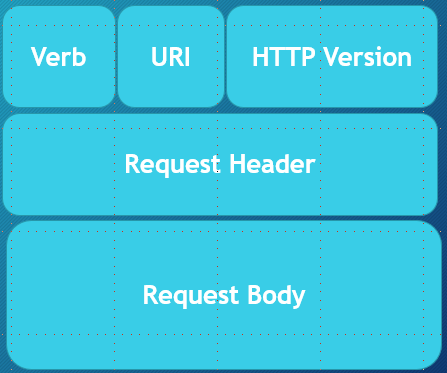
***Hypertext Transfer Protocol (HTTP)*** is a client-server protocol. This means the client or consumer must initiate the communication.

Information is transferred via multiple messages- it's akin to the way we send letters.

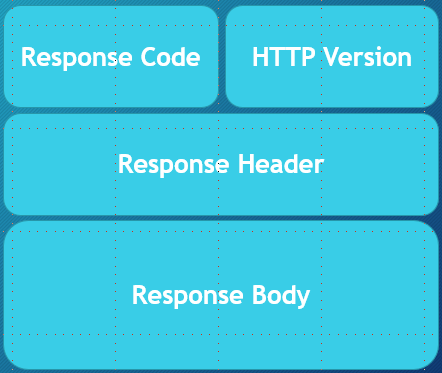
The HTTP protocol is part of the application layer and thus requires some underlying communication protocol to transmit data- that protocol is TCP.

HTTP Requests

HTTP Requests are composed of:

* Verb
  + Indicates the executing HTTP method.
* URI
  + Specifies the endpoint where the resource is located.
* HTTP Version.
* Request Header
  + META-DATA (information) of the Request as key-value pairs such as format supported by the client, browser type, etc.
* Request Body
  + Message content or resource representation.

HTTP Responses

* HTTP Response
  + Response Code
  + 200 (OK), 403 (Forbidden), 404 (Not Found), 500 (Internal Error), etc.
* HTTP Version.
* Response Header
  + META-DATA for the Response such as: content length, content type, date, etc.
* Response Body
  + Some kind of payload in the case where HTTP is used in the context of a RESTful service then the body is a resource representation.

Reference

* [IBM article on SOA gives an overview of HTTP requests and responses](https://www.ibm.com/support/knowledgecenter/SSMQ79_9.5.1/com.ibm.egl.pg.doc/topics/pegl_serv_overview.html)
* [HTTP MDN overview](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview)

**Verbs**

# Verbs

HTTP request verbs or methods indicate the action that a client hopes to perform.

Characteristics of requests: (Basically, a request either is or isn't each of these)

* idempotent
  + "identical request can be made once or several times in a row with the same effect while leaving the server in the same state"- MDN
* safe
  + doesn't alter the server's state
  + read-only
* cacheable
  + Determines if there is a chance the response to the corresponding method can be cached
* allowed in HTML forms
* request has a body

### **GET**

* Used to retrieve data
* NO request body
* safe
* idempotent
* cacheable
* allowed in HTML forms

### **HEAD**

* Essentially the same as GET, but the server's response should not include a body.
* Let's say you are planning to request a large amount of info and you want to test out the response without the risk of wasting bandwidth resources--> HEAD
* Used to retrieve data
* NO request body
* safe
* idempotent
* cacheable
* NOT allowed in HTML forms
* NO response body

### **POST**

* "sends data to the server"
* will often be used to create or update data
* it is NOT idempotent
* technically there are ways it can be cacheable, but typically isn't thought of as cacheable
* NOT safe
* allowed in HTML forms
* request has a body
* response has a body

### **PUT**

* updates a target resource such that it replaces the current representations with the one included in the PUT request
* it is idempotent
* Not safe
* Not cacheable
* Not allowed in HTML forms
* response does NOT have a body
* request does have a body
* not supported by HTML forms

### **DELETE**

* used to delete a resource
* might have a request and/or response body
* Not safe
* Not cacheable
* Not allowed in HTML forms
* is idempotent

### **CONNECT**

* starts a "tunnel"-- two-way communication with the resource
* No request body
* Receives a response body
* Not safe, idempotent, cacheable, or supported by HTML forms

### **OPTIONS**

* Get the options for communication with a particular resource
* safe, idempotent
* request has no body
* response has a body
* Not cacheable or available in HTML forms

### **TRACE**

* message loop-back test between client and resource
* useful for debugging
* no request body or response body
* not safe, cacheable, or allowed in HTML forms
* idempotent

### **PATCH**

* similar to update
* changes partial aspects of a resource
* request and response have a body
* not safe, idempotent, allowed in HTML forms, or cacheable

## References

* [MDN HTTP request methods](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods)

Day 4

**JavaScript Introduction**

## JavaScript

JavaScript is the most commonly used **client-side** scripting language. It is a **high-level, multi-paradigm, interpreted** programming language used to create dynamic webpages. When the browser loads a webpage, the browser interprets javascript code in it and executes it.

When we say JavaScript is a client-side language, we mean that it runs in the user's web browser and not on a server. However, although JavaScript originated as a scripting language that runs in the browser, the Node.js runtime environment does allow JavaScript code to be run on servers as the backend program for an application.

When we say JavaScript is a high level language, we mean that it abstracts away many implementation details that relate to computer hardware - like allocating memory or garbage collection of objects. When we say it is multi-paradigm, that means it supports many programming paradigms like procedural, object-oriented, and functional programming.

ECMAScript is the official name for JavaScript. It is used for language specification. The versions of a Javascript are defined by ECMAScript (ES). The versions of JavaScript are ES5, ES6, ES7, and so on.

In HTML, JavaScript code is written inside the <script> and </script> tags. You can place any number of scripts in an HTML document. Scripts can be placed in the <body>, or in the <head> section of an HTML page, or in both.

<script>

JavaScript code

</script>

**Internal JavaScript** - When the javascript code placed anywhere within the HTML page using <script> tags, then it called Internal JavaScript.

**External Javascript** - When the same JavaScript code used for many webpages then we place the JavaScript code in a separate file. When JavaScript code placed in a separate file from the HTML code is called External JavaScript. These files are saved with the ". js" extension and imported into the HTML page using the src attribute. The src attribute specifies the URL/path of an external JavaScript file. (Eg: <script src="./script.js"></script>)

## Syntax

JavaScript is case sensitive. Every statement in JavaScript is separated using a semicolon (;). JavaScript ignores multiple spaces and tabs.

**JavaScript Comments**: JavaScript support single-line as well as multi-line comments. Single line comment starts with // and multi-line comments are wrapped between /\* and \*/.

**JavaScript Literals**: Literals are the fixed values, it can be numbers, strings, boolean values, etc. The number type stores integer, float, and hexadecimal value. Strings are text, enclosed within single or double quotes (' or "). If the number is enclosed with single or double quotes, then it is considered as a string.

**JavaScript Keywords**: Keywords are tokens that have special meaning in JavaScript. The keywords supported by JavaScript are break, case, catch, continue, do, else, finally, for, function, if, in, new, return, switch, this, throw, try, typeof, var, void, while, etc.

## JavaScript Variables

Variables are used to store data values. It uses the var keyword to declare variables. An equal sign (=) is used to assign values to variables. Variable names are identifiers that should not be a JavaScript keyword. It starts only with the alphabet, underscores (\_) or dollar ($). It cannot start with a number and also there shouldn't be spaces in-between.

Example:

<script>

var x, y ; //declaring the variable

x **=** 6; y **=** 6; // assigning values

var name **=** 'John';

</script>

## Operators

Javascript Operators performs some operation on single or multiple operands and produces a result. The categories of operators and the operators used in JavaScript are listed below:

* Arithmetic operators - +, -, \*, /, %, ++, --
* Comparison Operators - ==, ===, !=, >, <, >=, <=
* Logical Operators - &&, ||, !
* Assignment Operators - =, +=, -=, \*=, /=, %=
* Ternary Operator - <condition> ? <value1> : <value2>;

### **Control Flow**

JavaScript uses the following control flow statements:

* if/else if/else
* for
* for-in
* for-of
* while
* do-while

The important difference between for-in and for-of is that the first allows iteration over the **keys** of any object, while for-of allows iteration over an array or array-like object. The syntax is demonstrated below.

let person **=** {

name: 'Bob',

age: 25

};

**for** (let prop in person) {

console.log(person[prop]); // prints 'Bob' and then 25

}

let people **=** [

{

name: 'Alice',

age: 30

},

{

name: 'Charlie',

age: 29

}

]

**for** (let pers of people) {

console.log(pers.name); // prints 'Alice' then 'Charlie'

}

### **Compared to Other Languages (Java/C#)**

When comparing to other languages, it's interesting to note that JavaScript was originally developed to be a scripting language with similar syntax to Java. While this means that some of the syntax bears a resemblence to the C-derived languages of Java and C#, JavaScript is very different from them in more meaningful ways.

* As a dynamically-typed language, JavaScript does not declare types and a variable you had previously stored a string in might hold a number. This also means that that there is no static type checking available, leading to potential issues where a function can return a value of multiple types. In Java or C#, this is far less likely as you usually declare a specific type for each variable, and static type checking is a feature leading to less confusion in utilizing an API.
* JavaScript is single-threaded and runs off of an event-loop. Java and C# support multi-threading.
* JavaScript can run on a server or on a browser. Java and C# would have to be compiled to WebAssembly in order to run on modern-day browsers.
* JavaScript is a multi-paradigm language, meaning we can solve our problems in a variety of ways, functionally, object-oriented, or event-driven. With Java and C#, while many of those paradigms have support in the language, Object-oriented solutions are preferred. In JavaScript, functions are first-class variables, meaning that they are treated like any other variable and can be passed as arguments to other functions.
* JavaScript utilizes prototype-based objects. Java and C# utilize class-based objects. In ES6, JavaScript introduced class-based syntax which allows us to interface with our prototypes in a similar manner to Java and C#. However, OOP in JavaScript is accomplished very differently than in these other languages. For example:
  1. An "overridden" method is merely shadowed on the prototype and is therefore still accessible.
  2. JavaScript has no concept of overloading.
  3. Encapsulation is possible in JavaScript but there are no access modifiers and is not as simple as in other languages.

**Data types**

## Data types

JavaScript includes the 7 following data types. The primitive data types used in javascript are:

* string
* number
* boolean
* null
* undefined
* object
* Symbol

Symbol is a new data type introduced in ES6 but is not commonly used.

Also, in this section we'll discuss about Arrays and Date specifically, which are both Objects.

**Loosely- Typed JavaScript** - JavaScript is a dynamic or loosely-typed language because a variable can hold the value of any data type at any point in time.

Example:

var a **=** 100;

a **=** true;

a **=** null;

a **=** undefined;

a **=** "Johnson";

alert(a); // Johnson

**NOTE:** The alert() method displays an alert box in the current browser window with a specified message and an 'OK' button.

Data types supported by JavaScript are listed below:

### **Strings**

Strings are text, enclosed within a single(') or double quotes ("). We can have quotes inside the string if they don't match the enclosing quotes.

Example:

var name **=** "Johnson";

var a **=**'hello';

var a **=** "Let's have a cup of coffee."; // single quote inside double quotes

var b **=** 'He said "Hello" and left.'; // double quotes inside single quotes

var c **=** 'We\'ll never give up.'; // escaping single quote with backslash

### **Numbers**

Numbers can be positive or negative numbers, with or without decimals and exponential representation of numbers. If a number is enclosed within single or double quotes, then it is considered a string. The Number data type also includes special values such as Infinity, NaN (Not-a-Number value). When we divide a number by 0, it results in infinity. When we do undefined mathematical operations such as taking the square root of -1, multiplying, or dividing with String the result is NaN. Example:

var a **=** 51; // integer

var b **=** 15.5; // floating-point number

var c **=** 1.5e-4; // exponential notation, equivalent to 0.0001.5

var d **=** 1.5e+4; // exponential notation, equivalent to 1.5e4 or 15000

var h **=** **-**6/0 // results infinity

var j **=** ("2" / 2); //results NaN

### **BigInt**

In JavaScript, the “number” type cannot represent integer values larger than 253 or less than -253 for negatives. BigInt type used to represent integers of arbitrary length. A BigInt is created by appending n to the end of an integer literal.

Example:

var bigInt **=** 1234567890123456789012345678901234567890n; //// the "n" at the end refers it's a BigInt

### **Boolean**

In JavaScript, we have two Boolean values: true and false.

Example:

var isEmpty **=** true;

var isGreater **=** (5**>**6); //returns false

(5 **==** 5) //returns true;

### **The “null” value**

A null value refers to nothing and is not equivalent to an empty string ("") or 0.

Example:

var a **=** null;

alert(a); // Output: null

### **The “undefined” value**

The undefined refers to value is not initialized. If a variable declared but not initialized with a value, its value is undefined.

Example:

let x;

alert(x); // shows "undefined"

var car **=** ""; // It is an empty string not undefined.

NOTE: An empty string has legal value and type defined.

### **Objects**

An object contains a set of key-value pairs. A key is a name that is of string type. The value can be any data type such as numbers, booleans, strings, function, arrays, and other objects.

The syntax of a javascript object as follows: var <object-name> = { key1: value1, key2: value2,... key: valueN};

Example:

var person1 **=** {

firstName: "John",

lastName: "Wilson",

age: 23,

};

You can access the values of an object's properties using dot notation (.) or bracket ([ ]).

Example:

person1.firstName; // returns John

person1.lastName; // returns Wilson

**or**

person1["firstName"];// returns John

person1["lastName"];// returns Wilson

**Object Constructor** - Another way to create an object is with Object Constructor using the new keyword. You can attach properties and methods to the object using dot (.) or bracket ([]).

Example:

var person2 **=** **new** Object();

// Attach properties to person object

person2.firstName **=** "Ben";

person2["lastName"] **=** "Tennyson";

person2.age **=** 18;

If we try to access properties or call methods that don't exist, then JavaScript will return undefined. The hasOwnProperty() method is used to check whether an object has a particular property or not.

Example:

var person3 **=** **new** Object();

person3.firstName; // returns undefined

**if**(person3.hasOwnProperty("firstName")){

person3.firstName;

}

We can use for..in loop to get the list of all properties and methods of an object.

Example:

**for**(var key in person1){

alert(key);

};

### **Arrays**

Arrays stores a list of values under a single variable.

The Syntax for Arrays in JavaScript as follows. var <array-name> = [element0, element1, element2,... elementN];

Array values are associated with an index starts with 0. The array includes "length" property that returns the number of values stored in it.

Example:

var arr **=** [1, 2.2, "hello", false];

alert(arr[0]); //Outputs 1

alert(arr[1]); //Outputs 2.2

alert(arr[2]); //Outputs hello

alert(arr[3]); //Outputs false

alert(arr.length); //Outputs 4

Arrays are also dynamic - we can add elements to grow the array and remove elements to shrink it. Below are some common array methods:

* .push()
* .pop()
* .filter()
* .map()
* .slice()

**Array Constructor** - We can also initialize an array using a new keyword.

Example:

var names **=** **new** Array();

names[0] **=** "John";

names[1] **=** "Ben";

names[2] **=** "Chris";

var numbers **=** **new** Array (1, 2, "three", 4);

### **Date**

JavaScript provides a Date object to work with date & time. It includes days, months, years, hours, minutes, seconds, and milliseconds in each object.

We can display the current date and time by creating a Date object using Date() or new Date().

Example:

Date(); //current date

var currentDate **=** **new** Date(); //current date

We can create a date object by specifying different parameters in the Date constructor:

var dt **=** **new** Date();

var dt **=** **new** Date(milliseconds);

var dt **=** **new** Date('date string');

var dt **=** **new** Date(year, month[, date, hour, minute, second, millisecond]);

Example for creating Date objects:

var date1 **=** **new** Date("3 march 2015");

var date2 **=** **new** Date("3 February, 2015");

var date3 **=** **new** Date("3rd February, 2015"); // invalid date

var date4 **=** **new** Date("2015 3 February");

### **Typeof Operator**

The typeof operator returns the data type of its operand.

Syntax: typeof operand or typeof (operand)

Example:

typeof 42; // output: "number"

typeof NaN ; // output: "number"

typeof 'hello'; // output: "string"

typeof true; // output: "boolean"

var a;

typeof a; // output: "undefined"

typeof 42n; // output: "bigint"

**function** **func**(){}

typeof func; // output: "function"

// For arrays, date, regular expression,null and objects , typeof operators returns "object"

typeof {a: 1}; // output: "object"

typeof [1, 2, 4]; // output: "object"

typeof **new** Date(); // output: "object"

typeof null; // output: "object"

,animal);

}

func(); //Output: "inside function: Lion"

console.log(animal); //error: animal is not defined

## Block Scope

Block scope is the scope of the variables declared inside the {} (curly brackets). In ES6, const and let keywords allow developers to declare variables in the block scope, which means those variables exist only within the corresponding block.

Example:

**function** **func**(){

**if**(true){

var fruit1 **=** 'apple'; //exist in function scope

const fruit2 **=** 'banana'; //exist in block scope

let fruit3 **=** 'strawberry'; //exist in block scope

}

console.log(fruit1);

console.log(fruit2); // results error - due to it exist in block scope

console.log(fruit3); // results error - due to it exist in block scope

}

foo();

The result will be:

apple

error: fruit2 is not defined

error: fruit3 is not defined

## Lexical Scope

Lexical scope is that a variable defined outside a function can access the inside another function defined after the variable declaration. The inner functions are lexically bound to the execution context of their outer functions.

Example:

**function** **func1**(){

var animal1 **=** "Lion"; //exist in Lexical scope

**function** **func2**(){ //Inner Function

var animal2 **=** "Tiger"; //exist in function scope

console.log(animal1);

console.log(animal2);

}

func2();

console.log(animal2); // results error - due to it exist in function scope

}

foo1();

The result will be:

Lion

Tiger

error: animal2 is not defined

### **Hoisting**

In JavaScript, variable declarations made with var and function declarations made with the function keyword are **hoisted** - or moved - to the top of the scope in which they are declared when the JavaScript interpreter parses the code. This means that variables and functions can be used **before they are even declared** as shown below.

**function** **example**() {

// var a declaration hoisted here

a **=** 4;

var a;

a**++**;

console.log(a); // prints 5

}

// anotherExample declaration hoisted to here

anotherExample(); // no error thrown!

**function** **anotherExample**() {

console.log('it works!');

}

**Arrays**

## Arrays

An array is a variable that allows the programmer to store more than one value. Arrays in JavaScript are objects and thus consist of key/value pairs and inherit from the Array prototype. Like objects, array values can consist of JavaScript primitives, or other JavaScript objects, including arrays and functions.

### **Creating an Array**

In JavaScript, arrays can be created using square brackets, using what is known as an array literal. They can also be created using the new keyword, but it is best practice to use array literals.

// array literal

let cheeses = ['bleu', 'cheddar', 'parmesan', 'brie']

// ["bleu", "cheddar", "parmesan", "brie"]

// new keyword

let primes = new Array(2, 3, 5, 7, 11, 13)

// [2, 3, 5, 7, 11, 13]

Both methods create an array object based on the Array Prototype.

### **Array Structure**

All array objects share a common structure. Each array has a length field that stores the current length of the array. In addition, the prototype of an array is [], giving each array access to certain functions that we will cover later.

let array = [1, 2, 3]

console.log(array.length) // 3

console.log(array.\_\_proto\_\_) // []

#### **Accessing an Array**

Arrays in JavaScript are **zero-indexed**, meaning that the first element in an array is represented by the key 0. In order to access our array, we can use Array Notation:

let cheeses = ['bleu', 'cheddar', 'parmesan', 'brie']

console.log(cheeses[0]) // bleu

Here, we gave the zero-index to obtain the first element in the array, which is the string bleu. The string cheddar would be represented by the key 1, parmesan would be 2, and brie would be 3. The length of the array is 4. The last key of an array is normally array.length - 1, however, be aware that there might not be data at this location.

//access the last element of the array

console.log(cheeses[cheeses.length-1])

#### **Modifying an Array**

You can also assign a different value by using the index and an assignment operator.

cheeses[2] = 'american' // changes parmesan to american

**Changing the Array Length** - JavaScript is a dynamic language and arrays are no different. the length of an array can be changed in several ways.

**Adding an Item** - To add an item to an array you can specify an index

// create an empty array

let arr = []

console.log(arr.length) // 0

// add an element at a specific index

arr[0] = 'duck'

console.log(arr.length) // 1

console.log(arr) // ["duck"]

arr[3] = 'chicken'

console.log(arr.length) // 4

console.log(arr) // [ "duck", <2 empty slots>, "chicken"]

In the above example, we first added the string duck to an empty array, changing the length of that array from 1 to 0. Note however, that this can be dangerous, as we can specify any index we wish. We then added the string chicken at index 3, changing the length of our array to 4, with two empty indexes. One way around this would be to use the length field to determine what the next index should be.

arr[arr.length] = 'pigeon'

console.log(arr.length) // 5

console.log(arr) // [ "duck", <2 empty slots>, "chicken", "pigeon"]

An easier and safer way to add a new element to an array is the push() method inherited from the Array Prototype.

let students = []

students.push('Timothy')

students.push('Zach')

console.log(students.length) // 2

console.log(students) // ["Timothy", "Zach"]

**Removing an Item** - Removing an item from an array in JavaScript can result in unexpected behavior if done incorrectly. When removing a key from an object, you might use the delete keyword. When you do this with an array, the length field will not be updated, resulting in an empty slot.

let arr = [1, 2, 3]

// [1, 2, 3]

console.log(arr.length) // 3

delete arr[0]

// [<empty slot>, 2, 3]

console.log(arr.length) // 3

In order to remove elements we turn to the methods splice(), pop(), and shift from the Array Prototype. The splice() method allows us to remove a number of elements at a specific index. It can also allow us to replace those elements with something else. splice() returns an array containing all the values removed from the array.

**NOTE**: In most browers, if you do not specify the number of elements to remove, it will remove all elements starting at the index specified.

let arr = [1, 2, 3, 4, 5, 6, 7, 8, 9]

arr.splice(2, 1) // returns [3]

console.log(arr)

// [1, 2, 4, 5, 6, 7, 8, 9]

arr.splice(4, 3) // returns [6, 7, 8]

console.log(arr)

// [1, 2, 4, 5, 9]

The pop() method removes and returns the last element in an array and the shift() method removes and returns the first element in an array.

let arr = [1, 2, 3, 4, 5, 6, 7, 8, 9]

arr.pop() // returns [9]

console.log(arr)

// [1, 2, 3, 4, 5, 6, 7, 8]

arr.shift() // returns [1]

console.log(arr)

// [2, 3, 4, 5, 6, 7, 8]

**NOTE:** - The length field can be modified directly. This can lead to arrays behaving a little oddly. If you change the length field to a value less than the actual length of the array, JavaScript will remove elements from the array in most browsers. If you set the field to a greater value, it will "add" empty fields. The fields that are added only exist in the toString method, as no keys are created.

### **Iterating Through an Array**

You can use a for loop to iterate through an array in JavaScript:

let list = [1, 2, 3, 4, 5];

// standard for

for(let i = 0; i < list.length; i++){

console.log(list[i]);

}

Arrays are iterable, and so you can use a for-in or for-of loop to iterate through an array. for-in will iterate through the keys of an array. for-of will iterate through the values of the array.

// for-in

for(key in list) {

// an enhanced for-in loop in JS iterates through

// the keys not the values

console.log(key);

console.log(list[key]);

}

// for-of

for(element of list) {

// if you say of, it will iterate through the values

console.log(element);

}

There is also the forEach() method on the Array Prototype. This function is a functional array method that takes in a callback function and runs that function for each element in the array. The forEach() method returns undefined.

// forEach()

list.forEach(

function(value, index) {

console.log(index + ' ' + value);

}

)

### **Array Methods**

The Array Prototype includes many useful methods. Above, we covered the splice(), shift(), push(), pop, and forEach(). Here we will talk about a few other useful methods.

#### **sort()**

The sort() method will sort an array "in-place". This means that the array will be modified and the original array will be sorted. The sort() method can take in a function that will compare objects and sort them to your preference. If no callback function is provided, each element is converted to a string and sorted in ascending order.

let list = ['tiger', 'panda', 'giraffe', 'cat', 'owl', 'bird']

list.sort()

// this will sort alphabetically

console.log(list)

// [ "bird", "cat", "giraffe", "owl", "panda", "tiger" ]

// This will sort by size of the string

list.sort(function(item1, item2) {return item1.length - item2.length})

console.log(list)

// [ "cat", "owl", "bird", "panda", "tiger", "giraffe" ]

#### **indexOf() & lastIndexOf()**

The indexOf() method returns the first index at which an element is present. lastIndexOf() returns the last index at which an element is present. If the element can't be found in the list, both will return the value -1.

let list = [1, 1, 2, 3, 3, 5, 6, 1]

list.indexOf(3) // returns 3

list.lastIndexOf(3) // returns 4

list.indexOf(1) // returns 0

list.lastIndexOf(1) // returns 7

list.indexOf('cat') // returns -1

list.lastIndexOf('cat') // returns -1

#### **find() & findIndex()**

The find() method returns the first element in an array for which the callback function returns a truthy value. The findIndex() method does the same but returns that element's index. For example, if I were to try to find the first string in an array that had a length greater than or equal to 7, I could write a function that tested for that and pass it to the find() method:

let words = ['the', 'small', 'fox', 'ate', 'a', 'largish', 'breakfast', 'and', 'slept']

words.find(function(word) {return word.length >= 7}) //returns 'largish'

words.findIndex(function(word) {return word.length >= 7}) //returns 5

#### **filter()**

The filter() function takes a callback function and creates a new array that is the made up of elements for which the callback function returns a truthy value. This can be useful in situations where you wish to perform an operation on only a subset of elements in an array. The original array is not modified. For example, I may wish to get only even numbers from an array:

let list = [4, 67, 34, 55, 79, 12]

let evens = list.filter(function(n) { return n%2 === 0 })

console.log(evens)

// [ 4, 34, 12 ]

#### **map()**

The map() function takes a callback function and creates a new array that is the result of calling the function on each element of the array. The original array is not modified. For example, if I were to have an array of numbers and I wished to have an array of the squares of those numbers, I could use the map() function to accomplish this:

let numbers = [1, 2, 3, 4, 5, 6, 7]

let squares = numbers.map(function(n) {return n\*n})

console.log(squares)

// [ 1, 4, 9, 16, 25, 36, 49 ]

#### **reduce()**

The reduce() function takes a callback function and returns a single value that is the result of calling the function for each value in the array. For example, if we wished to have the sum of each element in an array, we could do the following:

let numbers = [1, 2, 3, 4, 5, 6, 7]

let result = numbers.reduce(function(previousValue, currentValue) {return previousValue + currentValue})

console.log(result) // 28

**"this" keyword**

JavaScript this keyword

The this keyword is a reference variable that refers to the current object.

* **this alone**: refers to a global Object.

Example:

var x **=** this;

* **this in function**: refers to the Global object [object Window].

Example:

**function** **myFunction**() {

**return** this;

}

* **this in strict mode**:

Example:

"use strict";

var x **=** this; //Here, this refers to the Global object [object Window]

"use strict";

**function** **myFunction**() {

**return** this; //Here, this is undefined.

}

* **this in Event Handlers**: refers to the HTML element that received the event

Example:

<button onclick="this.style.backgroundColor= 'green'">

Click Me!

</button>

* **this in Object Method Binding**: refers to the object. In the below example, this refers to people object.

Example:

let **people** **=** **function**(name, age) {

this.name **=** name;

this.age **=** age;

**this.displayInfo** **=** **function**() {

document.write(this.name **+** " is " **+** this.age **+** " years old");

}

}

* **The call() and apply() method**: allows us to write a method that can be used on different objects. Here, person1 object writes its fullName function on person2 object using call() method

Example:

var person1 **=** {

**fullName**: **function**() {

**return** this.firstName **+** " " **+** this.lastName;

}

}

var person2 **=** {

firstName:"John",

lastName: "Wilson",

}

document.write("Hello, "**+** person1.fullName.call(person2));

**"strict" mode**

strict mode

JavaScript is a loosely typed scripting language. JavaScript allows strictness of code by using "use strict"; statement at the top of JavaScript code or in a function.

For example, When we expect the compiler to give an error if we have used a variable before defining it, then we apply "strict mode" to the javascript code.

<!DOCTYPE html>

<html>

<body>

<h1>strict mode</h1>

<p id="errorMessage"></p>

<script>

"use strict";

**try**

{

var x **=** 16; // valid in strict mode

y **=** 10; // error

}

**catch**(ex)

{

document.getElementById("errorMessage").innerHTML **=** ex;

}

</script>

</body>

</html>

The output will be:

strict mode

ReferenceError: y is not defined

The strict mode in JavaScript does not allow us to use undefined variables, reserved keywords as variable or function name, duplicate properties of an object, duplicate parameters of the function, assign values to read-only properties, Modifying arguments object, and Deleting an undeletable property.

Example:

<script>

**try**

{

"use strict";

x **=** 1; // error

var **for** **=** 19; // error

var **break** **=** 5; // error

var person **=** { name: "John", name: "Ben" }; //error

**function** **divide**(val, val){**return** val / val }; //error

var arr **=** [1 ,2 ,3 ,4, 5];

arr.length **=** 10; //error

}

**catch**(ex)

{

document.getElementById("errorMessage").innerHTML **=** ex;

}

</script>

Strict mode can be applied to function level in order to implement strictness only in that particular function.

Example:

x **=** 1; //valid

**function** **sum**(val1, val2){

"use strict";

result **=** val1 **+** val2; //error

**return** result;

}

**JSON**

JSON

JSON (**J**ava**S**cript **O**bject **N**otation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate.

JSON Object is a set of *key and value pair* enclosed within curly braces. A key is a string enclosed in quotation marks.A value can be a string, number, boolean expression, array, or object. A key value pair follows a specific syntax, with the key followed by a colon followed by the value. Key/value pairs are separated by comma. ("name" : "Andy")

**Example:**

var Book **=** {

"id": 110, ‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬‬

"language": "Python",

"author": ["John", "Ben"]

};

We can store mutiple JSON objects in Arrays.

**Example:**

var student **=**[

{

"id":"01",

"name": "Tom",

"lastname": "Price"

},

{

"id":"02",

"name": "Nick",

"lastname": "Thameson"

}

] ;

Some of the applications of JSON are listed below:

* Used to transmit data between the server and web application using JSON.
* JSON format helps transmit and serialize all types of structured data.
* Allows us to perform asynchronous data calls without the need to do a page refresh.
* Web services and Restful APIs use the JSON format to get public data.

**Type Coercion**

## == vs ===

== is used for comparison between two variables irrespective of the data type of variable. === is used for comparison between two variables but this will check strict type, which means it will check datatype and compare two values.

**Example:**

5 **==** '5' // Returns true

5 **===** '5' // Returns False

== is a type converting equality that automatically converts the string ('5') to number (5). It does implicit type conversion of variables.

=== is strict equality also compares the data type of the variable. It returns true only if the data type and value of the two variables are the same.

## Type coercion

Type coercion is the process of converting a value from one data type to another data type.

**Explicit type coercion** - We can explicitly convert the datatype of the variable. For example: Number('3'), String (123), Boolean(2)

**Implicit type coercion** - JavaScript is a loosely-typed language, values can also be converted between different types automatically. It usually happens when you apply operators to values of different types. For example: '3' \* '2', 2/’5', 123 + ''

### **String conversion**

To explicitly convert values to a string apply the String() function. Implicit coercion is triggered by the binary + operator when any operand is a string.

**Example:**

**String**(123) // explicit

123 **+** '' // implicit

### **Boolean conversion**

To explicitly convert a value to a boolean apply the Boolean() function. Implicit conversion happens in logical context, or is triggered by logical operators ( || && !).

**Example:**

**Boolean**(21) // explicit

**if** (2) { ... } // implicit due to logical context

7 **||** 'hello' // implicit due to logical operator

### **Number conversion**

To explicitly convert a value to a boolean apply the Number() function. Implicit coercion is triggered by comparison operators (>, <, <=,>=), bitwise operators ( | & ^ ~), arithmetic operators (- + \* / % ),unary + operator and loose equality operator==.

**Example:**

Number('123') // explicit

**+**'123' // implicit

123 **!=** '456' // implicit

**toString() method** - used to convert values to String.

(100 **+** 23).toString() //returns "123"

true.toString() // returns "true"

Javascript automatically calls the variable's toString() function when you try to "output" an object or a variable.

obj **=** {name:"Ben"} // toString converts to "[object Object]"

arr **=** [1,2,3,4] // toString converts to "1,2,3,4"

date **=** **new** Date() // toString converts to "Fri Jul 18 2014 09:08:55 GMT+0200"

## Truthy and Fasly in JavaScript

### **Falsy value**

In JavaScript, any expressions or value that results in boolean false value, are considered as Falsy. The falsy values/expressions in javascript are:

1. Obviously boolean false is false.
2. Any empty string will be evaluated to false.
3. Any undefined variable will be equal to false.
4. Any null variable will be equal to false.
5. Any numerical expression with result in NaN (not a number) will be equal to false.
6. Any numerical expression with result in zero will be equal to false.

### **Truthy value**

In JavaScript, any expressions or value that results in boolean true value, are considered as Truthy. Any expression or value other than above listed falsy values – is considered truthy.

**Example:**

'Hello' // truthy

**if**(1){} // truthy

**if**(**-**1){} // truthy

**new** Boolean(false); // is truthy values because 'object' is always true

**new** String('') // is truthy values because 'object' is always true

**Functions**

# JavaScript Functions

A function is a group of reusable code which can be called anywhere in the program. A JavaScript function is defined using the function keyword. The syntax for creating a function:

function name(parameter1, parameter2, parameter3) {

// code to be executed

}

A simple JavaScript Function:

**function** **showMessage**() {

alert( 'Hello everyone!' );

}

//function call

showMessage(); // outputs "Hello everyone!" in alert box.

JavaScript can return a value by executing the code after the function call. To return a value, we use the return keyword.

var product **=** multiplyFunc(4, 3); // Function called and the return value is stored in 'product' variable.

**function** **myFunction**(a, b) {

**return** a **\*** b; // Function returns the product of a and b

}

Simple Snippet : To convert centimeter to feet using JavaScript Functions

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Function for converting centimeter to feet</h2>

<p id="demo"></p>

<script>

**function** **toFeet**(cm) {

**return** (cm/30.48);

}

document.getElementById("demo").innerHTML **=** toFeet(180);

</script>

</body>

</html>

The output will be:

JavaScript Function for converting centimeter to feet

5.905511811023622

## Pass by value

In JavaScript, all function arguments are passed by value. This means that the value of any variable passed to a function is copied into the argument of the function. Any changes you make to the argument will not be reflected in the variable outside of the function.

### **Primitives**

With primitive values this behavior is straightforward. The primitive value is copied to a new variable:

function changeValue(number) {

console.log(number) // 20

number = 42

console.log(number) // 42

}

let myNumber = 20

changeValue(myNumber)

console.log(myNumber) // 20

In the above example, we defined a primitive value myNumber to be 20. When we passed this variable into the changeValue function, it copied the value 20 into the new number variable. When we changed number it did not affect myNumber because those are two different variables, each with their own value.

### **Objects**

If you pass an object into a function, the story is slightly different. The value that is stored in a variable containing an object is not the object itself. Instead, an object reference is being stored inside of that variable. When you pass a variable containing a reference to an object, that reference is copied into the arguments of the function. Since the new variable has a copy of that object reference, we can use this variable to modify the object.

let myObject = {'pet': 'Cat'}

console.log(myObject.pet) // 'Cat'

function adoptDog(obj) {

obj.pet = 'Dog'

}

adoptDog(myObject)

console.log(myObject.pet) // 'Dog'

It is tempting to conclude that objects are pass by reference, because you can modify the object that we pass into the function. However, if we attempt to change the value of the variable by assigning a new object we see that this isn't true:

let myObject = {'pet': 'Cat'}

console.log(myObject.pet) // 'Cat'

function adoptDog(obj) {

obj = {'pet': 'Dog'}

}

adoptDog(myObject)

console.log(myObject.pet) // 'Cat'

Here, we see that because we reassigned the variable obj to a new object, the value of the variable changed and so the value of the variable myObject did not change. JavaScript is pass by value.

## Function Expression / Anonymous Function

Function Expressions also are known as a named or anonymous function. An anonymous function is a function declared without any identifier refer to it. It is an expression that the variable holds a function. For example: var x = function (a, b) {return a \* b};

Example:

var **anon** **=** **function**() {

alert('I am anonymous');

};

var prd **=** **function** (a, b) {

**return** a **\*** b;

};

anon();

alert("prd = " **+** prd(2,4));

The above example results in two alert boxes on the current browser. The first alert box has "I am anonymous" inside it. The second alert box has "prd = 8" inside it.

## Self-Invoking Functions / IIFE Functions

A self-invoking function is an anonymous function that is invoked immediately after its definition. It is also known as the IIFE (Immediately Invoked Function Expression) function. It holds an anonymous function inside a set of parentheses (), which does the execution.

Syntax : (function(){ code goes here...})();

Example:

(**function**(){

// do this right now

console.log("Look at me, I'm running");

})();

## Callback Functions

A callback function is a function that gets executed after another function completes the execution. It helps us develop asynchronous JavaScript code and keeps us safe from problems and errors. JavaScript runs the code in sequential order (from top to down). If there is a case that code runs after some other execution, which is not happening in a sequence is called **asynchronous programming**.All functions in JavaScript are objects and a JavaScript function can be passed another function as an argument.

A callback function can be created by using the callback keyword as the last parameter.

Example for callback functions:

**function** **funcOne**(x) { alert("x = " **+** x); }

**function** **funcTwo**(y, callback) {

callback(y);

}

funcTwo(2, funcOne);

In the above example, funcOne is the callback function. When funcTwo(2, funcOne); is called, funcTwo takes in a variable (y) and a function (funcOne). funcTwo then passes the variable (y=2) to the function it took in, i.e. funcOne(2) is called. Then, issues an alert with x=2 on the current browser.

We can also pass an anonymous functions as a callback function.

Example:

**function** **funcTwo**(y, callback) {

callback(y);

callback(y);

}

**functionTwo**(10, **function**(x) { alert("x = " **+** x); })

The above example issues an alert two times, saying x = 10 on the current browser.

## Closures

A closure is a function that remembers and accesses the variables and arguments of its outer function even after the function return. The closure able to access the variables defined between its curly brackets, the outer function’s variables and the global variables.

Example:

**function** **greeting**() {

var message **=** 'Hi';

**function** **sayHi**() {

console.log(message);

}

**return** sayHi;

}

let hi **=** greeting();

hi(); // prints "hi" in the console.

Normally, when the greeting() function has completed executing, the message variable is no longer accessible. In this case, we execute the hi() function that references the sayHi() function, the message variable still exists. Hence, the sayHi() function is a closure.

## Hoisting of Functions and Variables

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution.

**Hoisting of Variables:**

The JavaScript compiler moves all the declarations of variables to the top so that there will not be any error.

Example:

<script>

//line 1

x **=** 1;

document.getElementById("p1").innerHTML **=** x ;

var x;

</script>

**NOTE:** var x; declaration moved to the top (at line 1) of their scope.

JavaScript Hositing only moves the variable declaration to the top, not the variables that are declared and initialized in a single line.

Example:

alert('x = ' **+** x); // displays x = undefined

var x **=** 1;

**NOTE:** Since hoisting is only possible with the declaration but not the initialization, var x = 1; not moved to the top of their scope.

**Hoisting of Functions:**

JavaScript compiler moves the function definition to the top in the same way as a variable declaration.

Example:

alert(Sum(5, 5)); // output: 10

**function** **Sum**(val1, val2)

{

**return** val1 **+** val2;

}

Javascript compiler moves only the Function declaration to the top, not the function expression.

Example:

add(5, 5); // Results an error

var add **=** **function** **sum**(a, b)

{

**return** a**+**b;

}

**Hoisting Functions Before Variables:**

JavaScript compiler moves a function's definition before the variable declaration.

Example:

alert(UseMe); // displays the UseMe Function definition

var UseMe **=** "UseMe Variable";

**function** **UseMe**()

{

alert("UseMe function called");

}

The above example will display "UseMe" function definition because JavaScript compiler moves the function before variables.Therefore, the alert box displays function UseMe() { alert("UseMe function called");}.

**NOTE:** The above code doesn't display "UseMe Variable" in the alert box due to hoisting functions before variables.

**Closures**

# JavaScript Functions

A function is a group of reusable code which can be called anywhere in the program. A JavaScript function is defined using the function keyword. The syntax for creating a function:

function name(parameter1, parameter2, parameter3) {

// code to be executed

}

A simple JavaScript Function:

**function** **showMessage**() {

alert( 'Hello everyone!' );

}

//function call

showMessage(); // outputs "Hello everyone!" in alert box.

JavaScript can return a value by executing the code after the function call. To return a value, we use the return keyword.

var product **=** multiplyFunc(4, 3); // Function called and the return value is stored in 'product' variable.

**function** **myFunction**(a, b) {

**return** a **\*** b; // Function returns the product of a and b

}

Simple Snippet : To convert centimeter to feet using JavaScript Functions

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Function for converting centimeter to feet</h2>

<p id="demo"></p>

<script>

**function** **toFeet**(cm) {

**return** (cm/30.48);

}

document.getElementById("demo").innerHTML **=** toFeet(180);

</script>

</body>

</html>

The output will be:

JavaScript Function for converting centimeter to feet

5.905511811023622

## Pass by value

In JavaScript, all function arguments are passed by value. This means that the value of any variable passed to a function is copied into the argument of the function. Any changes you make to the argument will not be reflected in the variable outside of the function.

### **Primitives**

With primitive values this behavior is straightforward. The primitive value is copied to a new variable:

function changeValue(number) {

console.log(number) // 20

number = 42

console.log(number) // 42

}

let myNumber = 20

changeValue(myNumber)

console.log(myNumber) // 20

In the above example, we defined a primitive value myNumber to be 20. When we passed this variable into the changeValue function, it copied the value 20 into the new number variable. When we changed number it did not affect myNumber because those are two different variables, each with their own value.

### **Objects**

If you pass an object into a function, the story is slightly different. The value that is stored in a variable containing an object is not the object itself. Instead, an object reference is being stored inside of that variable. When you pass a variable containing a reference to an object, that reference is copied into the arguments of the function. Since the new variable has a copy of that object reference, we can use this variable to modify the object.

let myObject = {'pet': 'Cat'}

console.log(myObject.pet) // 'Cat'

function adoptDog(obj) {

obj.pet = 'Dog'

}

adoptDog(myObject)

console.log(myObject.pet) // 'Dog'

It is tempting to conclude that objects are pass by reference, because you can modify the object that we pass into the function. However, if we attempt to change the value of the variable by assigning a new object we see that this isn't true:

let myObject = {'pet': 'Cat'}

console.log(myObject.pet) // 'Cat'

function adoptDog(obj) {

obj = {'pet': 'Dog'}

}

adoptDog(myObject)

console.log(myObject.pet) // 'Cat'

Here, we see that because we reassigned the variable obj to a new object, the value of the variable changed and so the value of the variable myObject did not change. JavaScript is pass by value.

## Function Expression / Anonymous Function

Function Expressions also are known as a named or anonymous function. An anonymous function is a function declared without any identifier refer to it. It is an expression that the variable holds a function. For example: var x = function (a, b) {return a \* b};

Example:

var **anon** **=** **function**() {

alert('I am anonymous');

};

var prd **=** **function** (a, b) {

**return** a **\*** b;

};

anon();

alert("prd = " **+** prd(2,4));

The above example results in two alert boxes on the current browser. The first alert box has "I am anonymous" inside it. The second alert box has "prd = 8" inside it.

## Self-Invoking Functions / IIFE Functions

A self-invoking function is an anonymous function that is invoked immediately after its definition. It is also known as the IIFE (Immediately Invoked Function Expression) function. It holds an anonymous function inside a set of parentheses (), which does the execution.

Syntax : (function(){ code goes here...})();

Example:

(**function**(){

// do this right now

console.log("Look at me, I'm running");

})();

## Callback Functions

A callback function is a function that gets executed after another function completes the execution. It helps us develop asynchronous JavaScript code and keeps us safe from problems and errors. JavaScript runs the code in sequential order (from top to down). If there is a case that code runs after some other execution, which is not happening in a sequence is called **asynchronous programming**.All functions in JavaScript are objects and a JavaScript function can be passed another function as an argument.

A callback function can be created by using the callback keyword as the last parameter.

Example for callback functions:

**function** **funcOne**(x) { alert("x = " **+** x); }

**function** **funcTwo**(y, callback) {

callback(y);

}

funcTwo(2, funcOne);

In the above example, funcOne is the callback function. When funcTwo(2, funcOne); is called, funcTwo takes in a variable (y) and a function (funcOne). funcTwo then passes the variable (y=2) to the function it took in, i.e. funcOne(2) is called. Then, issues an alert with x=2 on the current browser.

We can also pass an anonymous functions as a callback function.

Example:

**function** **funcTwo**(y, callback) {

callback(y);

callback(y);

}

**functionTwo**(10, **function**(x) { alert("x = " **+** x); })

The above example issues an alert two times, saying x = 10 on the current browser.

## Closures

A closure is a function that remembers and accesses the variables and arguments of its outer function even after the function return. The closure able to access the variables defined between its curly brackets, the outer function’s variables and the global variables.

Example:

**function** **greeting**() {

var message **=** 'Hi';

**function** **sayHi**() {

console.log(message);

}

**return** sayHi;

}

let hi **=** greeting();

hi(); // prints "hi" in the console.

Normally, when the greeting() function has completed executing, the message variable is no longer accessible. In this case, we execute the hi() function that references the sayHi() function, the message variable still exists. Hence, the sayHi() function is a closure.

## Hoisting of Functions and Variables

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution.

**Hoisting of Variables:**

The JavaScript compiler moves all the declarations of variables to the top so that there will not be any error.

Example:

<script>

//line 1

x **=** 1;

document.getElementById("p1").innerHTML **=** x ;

var x;

</script>

**NOTE:** var x; declaration moved to the top (at line 1) of their scope.

JavaScript Hositing only moves the variable declaration to the top, not the variables that are declared and initialized in a single line.

Example:

alert('x = ' **+** x); // displays x = undefined

var x **=** 1;

**NOTE:** Since hoisting is only possible with the declaration but not the initialization, var x = 1; not moved to the top of their scope.

**Hoisting of Functions:**

JavaScript compiler moves the function definition to the top in the same way as a variable declaration.

Example:

alert(Sum(5, 5)); // output: 10

**function** **Sum**(val1, val2)

{

**return** val1 **+** val2;

}

Javascript compiler moves only the Function declaration to the top, not the function expression.

Example:

add(5, 5); // Results an error

var add **=** **function** **sum**(a, b)

{

**return** a**+**b;

}

**Hoisting Functions Before Variables:**

JavaScript compiler moves a function's definition before the variable declaration.

Example:

alert(UseMe); // displays the UseMe Function definition

var UseMe **=** "UseMe Variable";

**function** **UseMe**()

{

alert("UseMe function called");

}

The above example will display "UseMe" function definition because JavaScript compiler moves the function before variables.Therefore, the alert box displays function UseMe() { alert("UseMe function called");}.

**NOTE:** The above code doesn't display "UseMe Variable" in the alert box due to hoisting functions before variables.

**async/await keywords**

async and await keyword

Async and Await are extensions of promises.

**async** -An async function is a function that operates asynchronously via the event loop, returns a Promise object implicitly. async ensures that the function returns a promise.

**await** - The await keyword is only valid inside async functions. await makes JavaScript wait until that promise settles and returns its result.

*Example:* Returns a promise object

async **function** **asyncFunc**() {

let response **=** await fetch(protectedUrl);

let text **=** await response.text(); // response body consumed

document.write(text);

}

asyncFunc();

**OOP in JS**

### **Encapsulation via Closures**

Encapsulation means hiding information or data.The simplest a way to create encapsulation in JavaScript is using closures. A closure can be created as a function with private state. When creating many closures sharing the same private state, we create an object.

**Example:**

const **Book** **=** **function**(t, a) {

let title **=** t;

let author **=** a;

**return** {

summary : **function**() {

console.log(`${title} written by ${author}.`);

}

}

}

const book1 **=** **new** Book('Hippie', 'Paulo Coelho');

book1.summary(); // Returns Hippie written by Paulo Coelho.

### **Prototypical inheritance**

Object Prototypes (\_\_proto\_\_ )- All JavaScript objects have a prototype. Browsers implement prototypes through the \_\_proto\_\_ property.

Function prototypes (prototype ) - In JavaScript, all functions are also objects, which means that they can have properties.Any time you create a function, it will automatically have a property called prototype.Thus, Functions also have a prototype property.

When we call a function with new, it sets the returned object’s \_\_proto\_\_ property equal to the function’s prototype property. This is the key to inheritance.

Inheritance in JavaScript is implemented through the **prototype chain**. Every normally created object, array, and function has a prototype chain of \_\_proto\_\_ properties ending with Object.prototype at the top.

**Example:** Here, we can say that "animal is the prototype of rabbit" or "rabbit prototypically inherits from animal". The animal properties and methods are become automatically available in rabbit. Such properties are called “inherited”.

let animal **=** {

eats: true

**walk**() {

alert("Animal walk");

}

};

let rabbit **=** {

jumps: true

\_\_proto\_\_: animal // sets animal to be a prototype of rabbit.

};

// we can find both properties in rabbit now:

alert( rabbit.eats ); // true

alert( rabbit.jumps ); // true

// walk is taken from the prototype

rabbit.walk(); // Animal walk

In the above example, when alert tries to read property rabbit.eats, it’s not in rabbit but JavaScript follows the prototype reference and finds it in animal.Also the walk() method is automatically taken from the prototype.

Day - 5

**AJAX Workflow with XmlHttpRequest Object**

# AJAX

AJAX stands for **A**synchronous **J**avaScript And **X**ML. Ajax describes the process of exchanging data from a web server asynchronously with the help of XML, HTML, CSS, and JavaScript. It just loads the data from the server and selectively updates some webpage parts without refreshing the page. Ajax uses the browser's built-in XMLHttpRequest (XHR) object to send and receive data to and from a web server asynchronously, in the background, without blocking the page or interfering with the user's experience.

Despite the name, modern usage of AJAX usually works with JSON data rather than XML as we will see.

## Ajax Work Flow

Ajax works flow starts from the client-side, on the browser. The steps involved in Ajax communication as follows:

1. A client event occurs on a webpage. (for example, the user clicks a button)
2. JavaScript creates an XMLHttpRequest object.
3. The XMLHttpRequest object makes an asynchronous request to the server.
4. The server process the received HttpRequest.
5. The server creates a response and sends data back to the browser.
6. Browser process the returned data using JavaScript.
7. The page content updated by javascript.

## The XMLHttpRequest Object

The keystone of AJAX is the XMLHttpRequest object. An XMLHttpRequest (XHR) object used to make HTTP requests to the server and receive data in response.

### **Create an XMLHttpRequest Object:**

Before you perform Ajax communication between client and server, we should create an XMLHttpRequest object.

var xhttp **=** **new** XMLHttpRequest();

### **Send a Request To a Server:**

To send a request to a server, we use the open() and send() methods of the XMLHttpRequest object.

* **open(method, URL, async)**

where, method — Specifies the HTTP request method to use, such as "GET", "POST", etc., URL - Specifies the location of the server async - Specifies whether the request should be handled asynchronously or not. If "true" then the script processing carries without waiting for a response. If "false" then the script waits for a response.

* **send()** - Sends the request to the server. It accepts an optional parameter that allows us to specify the request's body (used for POST).

In the GET method, the data is sent as URL parameters. In the POST method, the data is sent to the server as a part of the HTTP request body, which is not visible in the URL.

xhttp.open("GET", "ajax\_info.txt", true);

xhttp.send();

### **Server Response**

The Server Response returned in the form of responseText or responseXML or status or statusText.

* **responseText** - Returns the response as a string.
* **responseXML** - Returns the response as XML.
* **status** - Returns the status as a number (For example, 200: "OK", 403: "Forbidden",404: "Page not found")
* **statusText** -Returns the status as a string (e.g., "Not Found" or "OK").

We have to wait for the data to be available to process it, and in this purpose, the state of availability of data is given by the **readyState** attribute of XMLHttpRequest. The **onreadystatechange** function is called every time the readyState changes.

The readyState property defines the status of the XMLHttpRequest:

* readyState = 0 : Not intialized
* readyState = 1 : Connection establised
* readyState = 2 : request received
* readyState = 3 : processing request
* readyState = 4 : request finished and response is ready

**Example:**

<script>

**function** **loadDoc**() {

var xhttp **=** **new** XMLHttpRequest();

**xhttp.onreadystatechange** **=** **function**() {

**if** (this.readyState **==** 4 **&&** this.status **==** 200) {

document.getElementById("demo").innerHTML **=**

this.responseText;

}

};

xhttp.open("GET", "ajax\_info.txt", true);

xhttp.send();

}

</script>

## Working with JSON in Ajax

Sending JSON request payload and receiving the JSON response object are common tasks while dealing with AJAX. JavaScript retrieves the JSON data sent by the server, parse them, and displaying them on the webpage.

### **JSON in Request Payload**

In JavaScript, to send a request using JSON data, we need to serialize our JSON object into a string. The **JSON.stringify()** method is used to converting an object to a string. Then, the server receives the string and process the request.

**Example:**

var data **=** {"name" : "Matt"};

var xmlhttp **=** **new** XMLHttpRequest();

xmlhttp.open("POST", "/demo", true);

//Use stringify() method to get string

xmlhttp.send( JSON.stringify( data ) );

### **JSON in Response Body**

If the response from the server is string/text, we need to parse them into a JSON object. The **JSON.parse()** method converts a JSON string representation to a JSON object.

**Example:**

var xmlhttp **=** **new** XMLHttpRequest();

**xmlhttp.onreadystatechange** **=** **function**() {

**if** (this.readyState **==** 4 **&&** this.status **==** 200) {

//Use parse() method to convert JSON string to JSON object

var responseJsonObj **=** JSON.parse(this.responseText);

console.log( responseJsonObj.name );

console.log( responseJsonObj.age );

}

};

xmlhttp.open("GET", "/demo", true);

xmlhttp.send();

### **XML in Response Body**

While we normally wish to parse our response from JSON, sometimes we might be dealing with an older server and we need to parse XML data. The **responseXML** field contains the data as a read-only Document Object Model.

**Example:**

var xmlhttp **=** **new** XMLHttpRequest();

**xmlhttp.onreadystatechange** **=** **function**() {

**if** (this.readyState **==** 4 **&&** this.status **==** 200) {

var responseXml **=** this.responseXML;

console.log( responseXml.getElementsByTagName('myTag') )

}

};

xmlhttp.open("GET", "/demo", true);

xmlhttp.send();

**Working with JSON in AJAX**

# AJAX

AJAX stands for **A**synchronous **J**avaScript And **X**ML. Ajax describes the process of exchanging data from a web server asynchronously with the help of XML, HTML, CSS, and JavaScript. It just loads the data from the server and selectively updates some webpage parts without refreshing the page. Ajax uses the browser's built-in XMLHttpRequest (XHR) object to send and receive data to and from a web server asynchronously, in the background, without blocking the page or interfering with the user's experience.

Despite the name, modern usage of AJAX usually works with JSON data rather than XML as we will see.

## Ajax Work Flow

Ajax works flow starts from the client-side, on the browser. The steps involved in Ajax communication as follows:

1. A client event occurs on a webpage. (for example, the user clicks a button)
2. JavaScript creates an XMLHttpRequest object.
3. The XMLHttpRequest object makes an asynchronous request to the server.
4. The server process the received HttpRequest.
5. The server creates a response and sends data back to the browser.
6. Browser process the returned data using JavaScript.
7. The page content updated by javascript.

## The XMLHttpRequest Object

The keystone of AJAX is the XMLHttpRequest object. An XMLHttpRequest (XHR) object used to make HTTP requests to the server and receive data in response.

### **Create an XMLHttpRequest Object:**

Before you perform Ajax communication between client and server, we should create an XMLHttpRequest object.

var xhttp **=** **new** XMLHttpRequest();

### **Send a Request To a Server:**

To send a request to a server, we use the open() and send() methods of the XMLHttpRequest object.

* **open(method, URL, async)**

where, method — Specifies the HTTP request method to use, such as "GET", "POST", etc., URL - Specifies the location of the server async - Specifies whether the request should be handled asynchronously or not. If "true" then the script processing carries without waiting for a response. If "false" then the script waits for a response.

* **send()** - Sends the request to the server. It accepts an optional parameter that allows us to specify the request's body (used for POST).

In the GET method, the data is sent as URL parameters. In the POST method, the data is sent to the server as a part of the HTTP request body, which is not visible in the URL.

xhttp.open("GET", "ajax\_info.txt", true);

xhttp.send();

### **Server Response**

The Server Response returned in the form of responseText or responseXML or status or statusText.

* **responseText** - Returns the response as a string.
* **responseXML** - Returns the response as XML.
* **status** - Returns the status as a number (For example, 200: "OK", 403: "Forbidden",404: "Page not found")
* **statusText** -Returns the status as a string (e.g., "Not Found" or "OK").

We have to wait for the data to be available to process it, and in this purpose, the state of availability of data is given by the **readyState** attribute of XMLHttpRequest. The **onreadystatechange** function is called every time the readyState changes.

The readyState property defines the status of the XMLHttpRequest:

* readyState = 0 : Not intialized
* readyState = 1 : Connection establised
* readyState = 2 : request received
* readyState = 3 : processing request
* readyState = 4 : request finished and response is ready

**Example:**

<script>

**function** **loadDoc**() {

var xhttp **=** **new** XMLHttpRequest();

**xhttp.onreadystatechange** **=** **function**() {

**if** (this.readyState **==** 4 **&&** this.status **==** 200) {

document.getElementById("demo").innerHTML **=**

this.responseText;

}

};

xhttp.open("GET", "ajax\_info.txt", true);

xhttp.send();

}

</script>

## Working with JSON in Ajax

Sending JSON request payload and receiving the JSON response object are common tasks while dealing with AJAX. JavaScript retrieves the JSON data sent by the server, parse them, and displaying them on the webpage.

### **JSON in Request Payload**

In JavaScript, to send a request using JSON data, we need to serialize our JSON object into a string. The **JSON.stringify()** method is used to converting an object to a string. Then, the server receives the string and process the request.

**Example:**

var data **=** {"name" : "Matt"};

var xmlhttp **=** **new** XMLHttpRequest();

xmlhttp.open("POST", "/demo", true);

//Use stringify() method to get string

xmlhttp.send( JSON.stringify( data ) );

### **JSON in Response Body**

If the response from the server is string/text, we need to parse them into a JSON object. The **JSON.parse()** method converts a JSON string representation to a JSON object.

**Example:**

var xmlhttp **=** **new** XMLHttpRequest();

**xmlhttp.onreadystatechange** **=** **function**() {

**if** (this.readyState **==** 4 **&&** this.status **==** 200) {

//Use parse() method to convert JSON string to JSON object

var responseJsonObj **=** JSON.parse(this.responseText);

console.log( responseJsonObj.name );

console.log( responseJsonObj.age );

}

};

xmlhttp.open("GET", "/demo", true);

xmlhttp.send();

### **XML in Response Body**

While we normally wish to parse our response from JSON, sometimes we might be dealing with an older server and we need to parse XML data. The **responseXML** field contains the data as a read-only Document Object Model.

**Example:**

var xmlhttp **=** **new** XMLHttpRequest();

**xmlhttp.onreadystatechange** **=** **function**() {

**if** (this.readyState **==** 4 **&&** this.status **==** 200) {

var responseXml **=** this.responseXML;

console.log( responseXml.getElementsByTagName('myTag') )

}

};

xmlhttp.open("GET", "/demo", true);

xmlhttp.send();

**Promises**

Promises

Promises are used to handle asynchronous operations in JavaScript.

The constructor syntax for a promise object is:

let promise = new Promise(function(resolve, reject) {

// executor (the producing code, "singer")

});

The function passed to new Promise is called the **executor**. When a promise object is created, the executor runs automatically. It contains the code which produces the result. The arguments resolve and reject are callbacks.

When the executor obtains the result, it should call one of these callbacks:

* resolve(value) — if the job finished successfully, with the result value.
* reject(error) — if an error occurred, the error is the error object.

The Promise.status property, gives information about the state of the Promise object. The promise object can have three states: **pending**, **fulfilled**, and **rejected**.

A Promise object connects the executor and the consuming functions which will receive the result or error. Consuming functions can be registered using methods .then, .catch and .finally.

**Example:**

var promise **=** **new** **Promise**(**function**(resolve, reject) {

const x **=** 5;

const y **=** 3;

**if**(x **>=** y) {

resolve();

} **else** {

reject();

}

});

promise.

**then**(**function** () {

console.log('Sucess! x have grater value');

}).

**catch**(**function** () {

console.log('Error');

});

**Fetch API**

Fetch API

Instead of using XMLHttpRequest enabled Ajax, we can use Fetch API, which is modern and versatile. The Fetch API provides a fetch() method defined on the window object. This method used to send requests and returns a Promise that retrieved from the response. A **Promise** object represents a value that may not be available now but, will be resolved in the future. It allows us to write asynchronous code.

The syntax for fetch() method: let promise = fetch(url, [options])

The browser requests the server and returns a promise as a response. When the request unable to make HTTP-request due to network problems or response has failure HTTP-status code is 404 or 500, then the Fetch API rejects the Promise object. When we get a response successfully form the server, the promise object returned in the **Response Body**.

The methods to access the response body in various formats:

* response.text() – read the response and return as text.
* response.json() – parse the response as JSON.
* response.formData() – return the response as FormData object .
* response.blob() – return the response as Blob (binary data with type).
* response.arrayBuffer() – return the response as ArrayBuffer(low-level representation of binary data).

We also use the async and await keyword with the fetch() method. The async keyword is added to functions to tell them to return a promise rather than directly returning the value. The await keyword only works inside async functions, used to pause the code on that line until promise gets complete.

*Example:*

async **function** **asyncFunc**() {

let response **=** await fetch(protectedUrl);

let text **=** await response.text(); // response body consumed

document.write(text);

}

asyncFunc();

**Response headers** - The response headers are available in a Map-like headers object in response.headers. To get individual headers by name or iterate over them.

*Example:*

async **function** **asyncFunc**() {

let response **=** await fetch(githubUrl);

// get one header

alert(response.headers.get('Content-Type')); // application/json; charset=utf-8

// iterate over all headers

**for** (let [key, value] of response.headers) {

alert(`${key} **=** ${value}`);

}

}

asyncFunc();

**Request headers** - To set a request header inside the fetch method, we can use the headers attribute.

*Example:*

let response **=** fetch(protectedUrl, {

headers: {

Authentication: 'secret'

}

});

**POST Request:**

To make a POST request, we need to mention the HTTP method (method)and request the body (body) inside the fetch method. The request body can be string, FormData object, or blob. If the request body is a string, then Content-Type header is set to text/plain;charset=UTF-8. If the request body is a JSON, then Content-Type header is set to application/json;charset=UTF-8. We don't set Content-Type manually for blob object.

*Example:*

async **function** **asyncFunc**() {

let user **=** {

name: 'John',

surname: 'Smith'

};

// url is a server location

let response **=** await fetch(url, {

method: 'POST',

headers: {

'Content-Type': 'application/json;charset=utf-8'

},

body: JSON.stringify(user)

});

let result **=** await response.json();

alert(result.message);

}

asyncFunc();

Handling Errors

The Fetch API generates a promise, meaning that if the request fails, it will cause the promise to enter the reject state. To handle this, we need to either surround our await instruction with a try...catch block or to append a catch() callback to our promise.

*Example:*

async **function** **asyncFunc**() {

let user **=** {

name: 'John',

surname: 'Smith'

};

// url is a server location

let response **=** await fetch(url, {

method: 'POST',

headers: {

'Content-Type': 'application/json;charset=utf-8'

},

body: JSON.stringify(user)

});

**try** {

let result **=** await response.json();

alert(result.message);

} **catch** (error) {

console.error(error);

}

}

asyncFunc();

*Example:*

let user **=** {

name: 'John',

surname: 'Smith'

};

// url is a server location

let response **=** fetch(url, {

method: 'POST',

headers: {

'Content-Type': 'application/json;charset=utf-8'

},

body: JSON.stringify(user)

}).then((response)**=>**{

let result **=** response.json();

alert(result.message);

}).**catch**((error)**=>**{

console.error(error);

});

**Coding Challenge 2**

Please find below the link to the coding challenge

[www.hackerrank.com/220502-coding-challenge-collection](https://www.hackerrank.com/220502-coding-challenge-collection)

Make sure to change the compiler to Java 8+

All the Best!