

Intro To HTTP

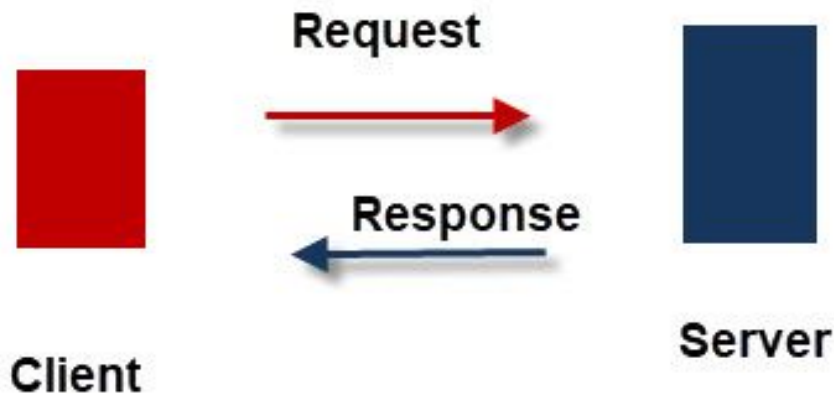
HTTP stands for hypertext transfer protocol and is used to transfer data across the Web. It's the communication protocol you use when you browse the web.

At a fundamental level, when you visit a website, your browser makes an HTTP request to a server. Then that server responds with a resource (e.g. image, video, or the HTML of a web page) - which your browser then displays for you.

There have been several versions of HTTP starting with the original **0.9 version**. The current version is 2.0.

How Does It work?

Like most Internet protocols, **HTTP** is a **command** and **response text-based** protocol using a **client-server** communications model.



HTTP Protocol Basics

The client makes a request and the server responds. The **HTTP protocol** is also a **stateless protocol** meaning that the server isn't required to store session information, and each request is independent of the other. This means:

- All requests originate at the client (your browser)
- The server responds to a request.
- The requests (commands) and responses are in readable text.
- The requests are independent of each other and the server **doesn't need to track** the requests.

Request and Response Structure

Request and response **message structures** are the same and shown below:

```
generic-message = start-line  
                  *(message-header CRLF)  
                  CRLF  
                  [ message-body ]
```

The diagram shows the structure of a generic HTTP message. The components are: 'start-line', '*(message-header CRLF)', 'CRLF', and '[message-body]'. A red arrow points from the word 'Optional' to the asterisk in the second line, and another red arrow points from the word 'Optional' to the square brackets in the fourth line.

HTTP Request or Response Message Format

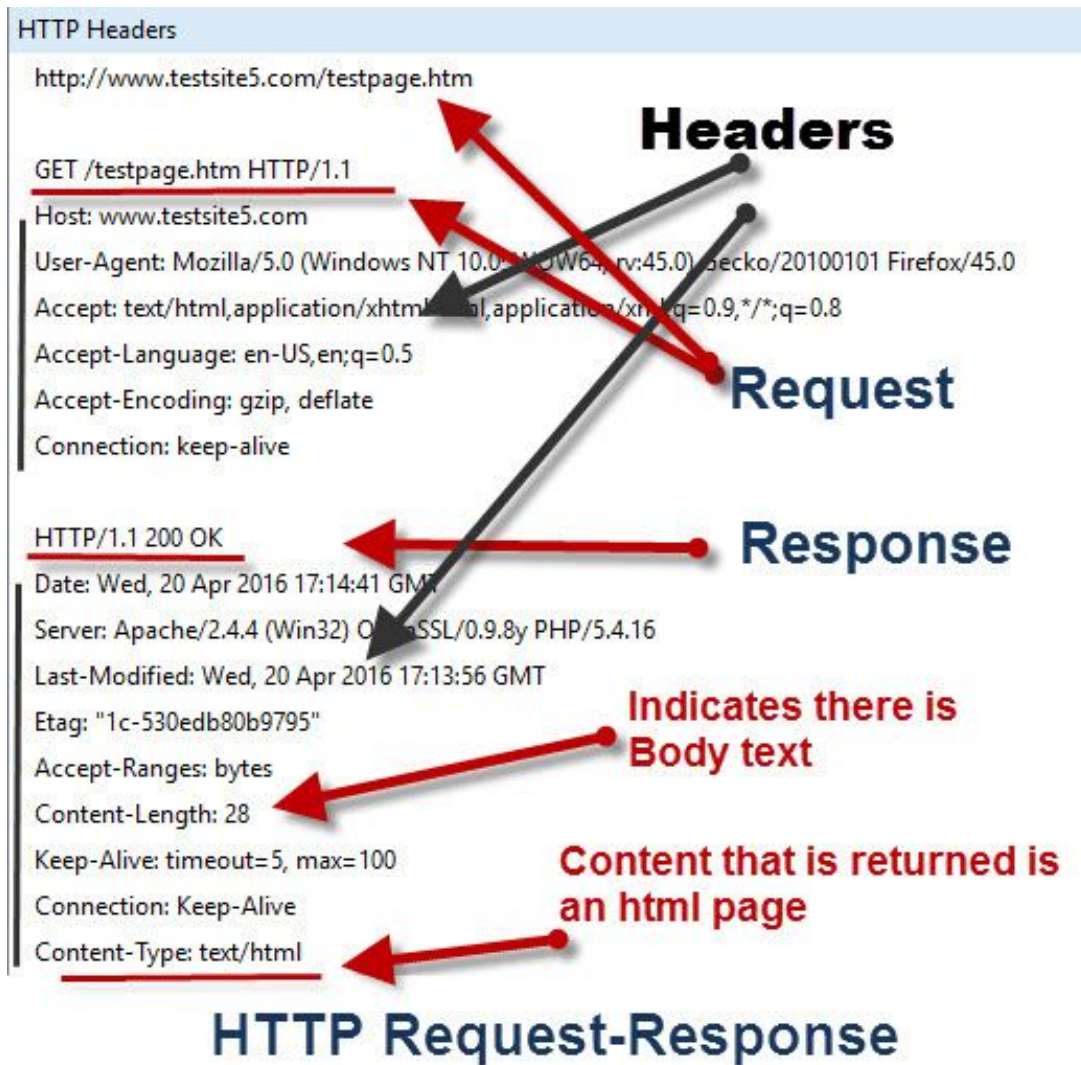
A request consists of:

A command or request + optional headers + optional body content.

A response consists of:

A status code + optional headers + optional body content.

Here is a screenshot of the **http request-response** that happens behind the scenes.



URLs

The URL (Uniform Resource Locator) is probably the most known concept of the Web. It is also one of most important and useful concepts. A URL is web address used to identify resources on the Web.

`http://www.example.com/search?item=vw+beetle`

Protocol Domain Path Parameters

Protocol: Most often they are HTTP (or HTTPS for a secure version of HTTP).

Domain: Name that is used to identify one or more IP addresses where the resource is located.

Path: Specifies the resource location on the server.

Parameters: Additional data used to identify or filter the resource on the server.

Note: When searching for articles and more information about HTTP, you may encounter the term URI (or uniform resource identifier). URI is sometimes being used instead of URL but mostly in formal specifications and by people who want to show off. ????

HTTP Requests

In HTTP, every request must have an URL address. Additionally, the request needs a method. The four main HTTP methods are:

- GET
- PUT
- POST
- DELETE

And these methods directly correspond to actions:

- read
- update
- create
- delete

We will study these methods, and more, in the HTTP Methods section. There are also some other interesting things in an HTTP request:

Referrer header: tells the URL from where the request has originated.

User-Agent header: additional information about the browser being used to generate the request.

Host header: uniquely identifies a hostname; it is necessary when multiple web pages are hosted on the same server.

Cookie header: submits additional parameters to the client.

HTTP Response Codes

Response Status codes are split into 5 groups each group has a meaning and a **three digit** code.

- **1xx** – Informational
- **2xx** – Successful
- **3xx** -Multiple Choice
- **4xx**– Client Error
- **5xx** -Server Error

For example a successful page request will return a **200** response code and an unsuccessful a **400** response code.

Here is the complete list:

Status code	Meaning
1xx Informational	
100	Continue
101	Switching protocols
102	Processing
103	Early Hints
2xx Successful	
200	OK
201	Created
202	Accepted
203	Non-Authoritative Information
204	No Content
205	Reset Content
206	Partial Content
207	Multi-Status

Status code	Meaning
208	Already Reported
226	IM Used
3xx Redirection	
300	Multiple Choices
301	Moved Permanently
302	Found (Previously "Moved Temporarily")
303	See Other
304	Not Modified
305	Use Proxy
306	Switch Proxy
307	Temporary Redirect
308	Permanent Redirect
4xx Client Error	
400	Bad Request
401	Unauthorized
402	Payment Required
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
407	Proxy Authentication Required
408	Request Timeout
409	Conflict
410	Gone
411	Length Required
412	Precondition Failed
413	Payload Too Large
414	URI Too Long
415	Unsupported Media Type
416	Range Not Satisfiable
417	Expectation Failed
418	I'm a Teapot
421	Misdirected Request

Status code	Meaning
422	Unprocessable Entity
423	Locked
424	Failed Dependency
425	Too Early
426	Upgrade Required
428	Precondition Required
429	Too Many Requests
431	Request Header Fields Too Large
451	Unavailable For Legal Reasons
5xx Server Error	
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout
505	HTTP Version Not Supported
506	Variant Also Negotiates
507	Insufficient Storage
508	Loop Detected
510	Not Extended
511	Network Authentication Required

HTTP Methods

The most common methods are GET and POST. But there are a few others, too.

GET: You use this method to request data from a specified resource where data is not modified in any way. GET requests do not change the state of the resource.

POST: You use this method to send data to a server to create a resource.

PUT: You use this method to update the existing resource on a server by using the content in the body of the request. Think of this as a way to "edit" something.

PATCH: You use this method to apply partial modifications to a resource.

DELETE: You use this method to delete the specified resource.

HTTP Headers

There are three main components that make up the request/response structure. These include:

- First line
- Headers
- Body/Content

We already talked about the first line in HTTP requests and responses. Now we'll talk about HTTP headers. The HTTP headers are added after the first line and are defined as name:value pairs separated by a colon. HTTP headers are used to send additional parameters along with the request or response.

response.

There are different types of headers that are grouped based on their usage into 4 broad categories:

- **General header**—Headers that can be used in both requests and response messages and that are independent of the data being exchanged.
- **Request header**—These headers define parameters for the data requested or parameters that give important information about the client making the request.
- **Response header**—These headers contain information about the incoming response.
- **Entity header**—The entity headers describe the content that makes up the body of the message.

```
POST / HTTP/1.1
Host: localhost:8000
User-Agent: Mozilla/5.0 (Macintosh;... )... Firefox/51.0
Accept: text/html,application/xhtml+xml,...,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Content-Type: multipart/form-data; boundary=-12656974
Content-Length: 345

-12656974
(more data)
```

Request headers

General headers

Entity headers

HTTPS (Hypertext Transfer Protocol Secure)

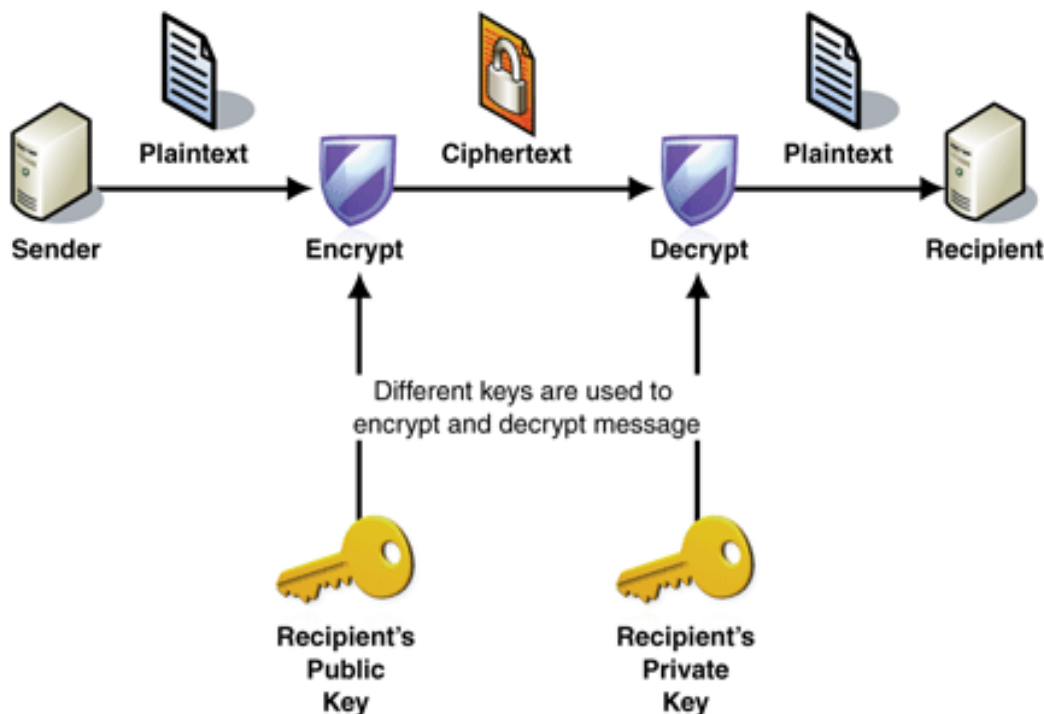
The secure version of HTTP protocol is Hypertext Transfer Protocol Secure (HTTPS). HTTPS provides encrypted communication between a browser (client) and the website (server).

In HTTPS, the communication protocol is encrypted using Transport Layer Security (TLS) or Secure Sockets Layer (SSL).

The protocol is therefore also often called HTTP over TLS, or HTTP over SSL.

Both the TLS and SSL protocols use an asymmetric encryption system. Asymmetric encryption systems use a public key (encryption key) and a private key (decryption keys) to encrypt a message.

Anyone can use the public key to encrypt a message. However, private keys are secret, and that means that only the intended receiver can decrypt the message.



SSL/TLS handshake

When you request a HTTPS connection to a website, the website sends its SSL certificate to your browser. That process where your browser and website initiate communication is called the "SSL/TLS handshake."

The SSL/TLS handshake involves a series of steps where browser and website validate each other and start communication through the SSL/TLS tunnel.

As you probably noticed, when a trusted secure tunnel is used during in a HTTPS connection, the green padlock icon is displayed in the browser address bar.

Benefits of HTTPS

The major benefits of a HTTPS are:

- Customer information, like credit card numbers and other sensitive information, is encrypted and cannot be intercepted.
- Visitors can verify you are a registered business and that you own the domain.
- Customers know they are not suppose to visit sites without HTTPS, and therefore, they are more likely to trust and complete purchases from sites that use HTTPS.

Intro To AJAX

AJAX stands for **Asynchronous JavaScript And XML**. It is not a programming language. It is a technology for developing better, faster and more interactive Web Applications using HTML, CSS, JavaScript and XML.

Ajax stands for **Asynchronous Javascript And Xml**. Ajax is just a means of loading data from the server and selectively updating parts of a web page without reloading the whole page.

Basically, what Ajax does is make use of the browser's built-in **XMLHttpRequest** (XHR) object to send and receive information to and from a web server asynchronously, in the background, without blocking the page or interfering with the user's experience.

Ajax has become so popular that you hardly find an application that doesn't use Ajax to some extent. The example of some large-scale Ajax-driven online applications are: Gmail, Google Maps, Google Docs, YouTube, Facebook, Flickr, and so many other applications.

XML : Extensible Markup Language (XML) is a format to store and transport data from the Web Server.

What's the meaning of Asynchronous in AJAX?

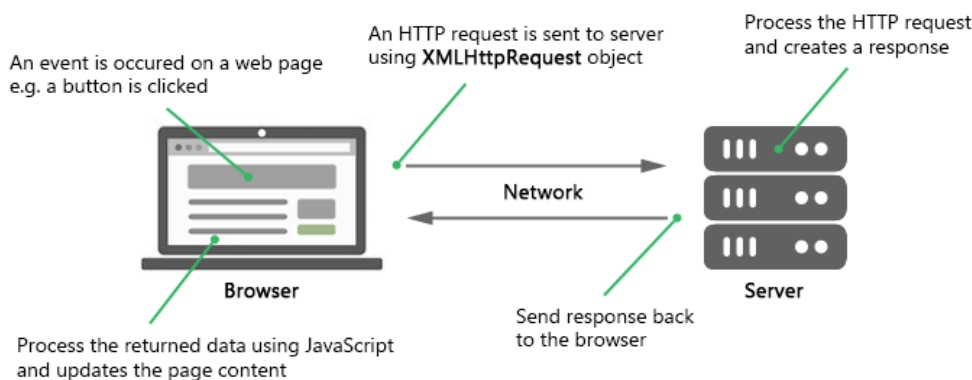
Asynchronous means that the Web Application could send and receive data from the Web Server without refreshing the page. This background process of sending and receiving data from the server along with updating different sections of a web page defines Asynchronous property/feature AJAX.

Understanding How Ajax Works?

To perform Ajax communication JavaScript uses a special object built into the browser—an **XMLHttpRequest** (XHR) object—to make HTTP requests to the server and receive data in response.

All modern browsers (Chrome, Firefox, IE7+, Safari, Opera) support the **XMLHttpRequest** object.

The following illustrations demonstrate how Ajax communication works:



Since Ajax requests are usually asynchronous, execution of the script continues as soon as the Ajax request is sent, i.e. the browser will not halt the script execution until the server response comes back.

In the following section we'll discuss each step involved in this process one by one:

Sending Request and Retrieving the Response

Before you perform Ajax communication between client and server, the first thing you must do is to instantiate an `XMLHttpRequest` object, as shown below:

```
var request = new XMLHttpRequest();
```

Now, the next step in sending the request to the server is to instantiate the newly-created request object using the `open()` method of the `XMLHttpRequest` object.

The `open()` method typically accepts two parameters—the HTTP request method to use, such as "GET", "POST", etc., and the URL to send the request to, like this:

```
request.open("GET", "info.txt"); -Or- request.open("POST", "add-user.php");
```

NOTE: The file can be of any kind, like `.txt` or `.xml`, or server-side scripting files, like `.php` or `.asp`, which can perform some actions on the server (e.g. inserting or reading data from database) before sending the response back to the client.

And finally send the request to the server using the `send()` method of the `XMLHttpRequest` object.

```
request.send(); -Or- request.send(body);
```

NOTE: The `send()` method accepts an optional `body` parameter which allows us to specify the request's body. This is primarily used for HTTP POST requests, since the HTTP GET request doesn't have a request body, just request headers.

Performing an Ajax GET Request

The GET request is typically used to get or retrieve some kind of information from the server that doesn't require any manipulation or change to the database, for example, fetching search results based on a term, fetching user details based on their id or name, and so on.

The following example will show you how to make an Ajax GET request in JavaScript:

```
<!DOCTYPE html>
<html>
<body>

<h2>The XMLHttpRequest Object</h2>
<button type="button" onclick="loadDoc()">Request data</button>

<p id="demo"></p>

<script>
function loadDoc() {
  // Creating the XMLHttpRequest object
  var request = new XMLHttpRequest();

  // Instantiating the request object
  request.open("GET", "https://jsonplaceholder.typicode.com/todos/1");

  // Defining event listener for readystatechange event
  request.onreadystatechange = function() {
    // Check if the request is complete and was successful
    if(this.readyState === 4 && this.status === 200) {
      // Inserting the response from server into an HTML element
      document.getElementById("demo").innerHTML = this.responseText;
    }
  };

  // Sending the request to the server
  request.send();
}
</script>
```



```
</body>
</html>
```

OUTPUT:

The XMLHttpRequest Object

Request data

```
{ "userId": 1, "id": 1, "title": "delectus aut autem", "completed": false }
```

When the request is asynchronous, the `send()` method returns immediately after sending the request. Therefore you must check where the response currently stands in its lifecycle before processing it using the `readyState` property of the `XMLHttpRequest` object.

The `readyState` is an integer that specifies the status of an HTTP request. Also, the function assigned to the `onreadystatechange` event handler is called every time the `readyState` property changes. The possible values of the `readyState` property are summarized below.

Value	State	Description
0	UNSENT	An XMLHttpRequest object has been created, but the open() method hasn't been called yet (i.e. request not initialized).
1	OPENED	The open() method has been called (i.e. server connection established).
2	HEADERS_RECEIVED	The send() method has been called (i.e. server has received the request).
3	LOADING	The server is processing the request.
4	DONE	The request has been processed and the response is ready.

NOTE: Theoretically, the `readystatechange` event should be triggered every time the `readyState` property changes. But, most browsers do not fire this event when `readyState` changes to 0 or 1. However, all browsers fire this event when `readyState` changes to 4 .

Performing an Ajax POST Request

The POST method is mainly used to submit a form data to the web server.

The following example will show you how to submit form data to the server using Ajax.

```
<!DOCTYPE html>
<html>
<body>

<h2>The XMLHttpRequest Object</h2>
<button type="button" onclick="loadDoc()">Post data</button>

<script>
function loadDoc() {
  // Creating the XMLHttpRequest object
  var request = new XMLHttpRequest();

  // Instantiating the request object
  request.open("POST", "https://jsonplaceholder.typicode.com/posts");
  request.setRequestHeader("Content-Type", "application/json;charset=UTF-8");

  // Sending the request to the server
  request.send(JSON.stringify({
    title: 'foo',
```



```

        body: 'bar',
        userId: 1,
    }));
}
</script>

</body>
</html>

```

OUTPUT:

Name	×	Headers	Payload	Preview	Response
<input type="checkbox"/> p...		1 { 2 "title": "foo", 3 "body": "bar", 4 "userId": 1, 5 "id": 101 6 }			

JQuery AJAX Introduction

The jQuery library includes various methods to send Ajax requests. These methods internally use XMLHttpRequest object of JavaScript. The following table lists all the Ajax methods of jQuery.

jQuery Ajax Methods	Description
ajax()	Sends asynchronous http request to the server.
get()	Sends http GET request to load the data from the server.
Post()	Sends http POST request to submit or load the data to the server.
getJSON()	Sends http GET request to load JSON encoded data from the server.
getScript()	Sends http GET request to load the JavaScript file from the server and then executes it.
load()	Sends http request to load the html or text content from the server and add them to DOM element(s).

The jQuery library also includes following events which will be fired based on the state of the Ajax request.

jQuery Ajax Events	Description
ajaxComplete()	Register a handler function to be called when Ajax requests complete.
ajaxError()	Register a handler function to be called when Ajax requests complete with an error.
ajaxSend()	Register a handler function to be called before Ajax request is sent.
ajaxStart()	Register a handler function to be called when the first Ajax request begins.
ajaxStop()	Register a handler function to be called when all the Ajax requests have completed.
ajaxSuccess()	Register a handler function to be called when Ajax request completes successfully.

jQuery get() Method:

The jQuery get() method sends asynchronous http GET request to the server and retrieves the data.

```
$.get(url, [data],[callback]);
```



Parameters Description:

- url: request url from which you want to retrieve the data
- data: data to be sent to the server with the request as a query string
- callback: function to be executed when request succeeds

The following example shows how to retrieve data:

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <script type="text/javascript"
    src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.2/jquery.min.js">
  </script>
  <script type="text/javascript">
    $(document).ready(function () {

      $('#ajaxBtn').click(function(){

        $.get('https://jsonplaceholder.typicode.com/todos/1', function (data, textStatus, jqXHR) {
          console.log(data);
        });
      });
    });
  </script>
</head>
<body>
  <h1> jQuery get() method demo
  </h1>
  <input type="button" id="ajaxBtn" value="Send GET request" />
</body>
</html>
```



OUTPUT:



jQuery post() Method:

The jQuery post() method sends asynchronous http POST request to the server to submit the data to the server and get the response.

```
$.post(url,[data],[callback],[type]);
```



Parameter Description:

- url: request url from which you want to submit & retrieve the data.
- data: json data to be sent to the server with request as a form data.
- callback: function to be executed when request succeeds.
- type: data type of the response content.

Let's see how to submit data and get the response using post() method

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <script type="text/javascript"
    src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.2/jquery.min.js">
  </script>
  <script type="text/javascript">
    $(document).ready(function () {

      $('#ajaxBtn').click(function(){

        $.post('https://jsonplaceholder.typicode.com/posts',// url
```



```

        {
            title: 'foo',
            body: 'bar',
            userId: 1,
        }, // data to be submit
        function(data, status, jqXHR) { // success callback
            $('p').append('status: ' + status + ', data: ' + JSON.stringify(data));
        });
    });
</script>
</head>
<body>
    <h1> jQuery post() method demo
    </h1>
    <input type="button" id="ajaxBtn" value="Send POST request" />
    <p>
    </p>
</body>
</html>

```

OUTPUT:

jQuery post() method demo

Send POST request

status: success, data: {"title":"foo","body":"bar","userId":"1","id":101}

AJAX Calls Using Fetch API In Javascript

Fetch is an interface for making an AJAX request in JavaScript. It is implemented widely by modern browsers and is used to call an API.

```
const promise = fetch(url, [options])
```



Calling fetch returns a promise, with a Response object. The promise is rejected if there is a network error, and it's resolved if there is no problem connecting to the server and the server responded a status code. This status code could be 200s, 400s or 500s.

A sample FETCH request -

```

fetch(url)
    .then(response => response.json())
    .catch(err => console.log(err))

```



The request is sent as a GET by default. To send a POST / PATCH / DELETE / PUT you can use the method property as part of options parameter. Some other possible values options can take -

- **method** : such as GET, POST, PATCH
- **headers** : Headers object
- **mode** : such as cors , no-cors , same-origin
- **cache** : cache mode for request
- **credentials**
- **body**

Example usage:

This example demonstrates the usage of fetch to call an API and to get a list of git repositories.

```
const url = 'https://api.github.com/users/bradtraversy/repos';

fetch(url)
  .then(response => response.json())
  .then(repos => {
    const reposList = repos.map(repo => repo.name);
    console.log(reposList);
  })
  .catch(err => console.log(err))
```



To send a POST request, here's how the method parameter can be used with async / await syntax.

```
const params = {
  id: 123
}

const response = await fetch('url', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(params)
});

const data = await response.json();
```



Creating A Web Page Using API

In this section, we will give you a tutorial for creating a web application with just only basic **HTML** ,**CSS** and **JavaScript** (based on **Bootstrap 5**) perform **CRUD** operations. Well, **CRUD** operations are the four basic operations of manipulating data including **Create/Construct**, **Read**, **Upda** and **Delete**. Furthermore, our **CRUD** operations will perform by the use of an external **API** from [MeCallAPI.com](https://www.mecallapi.com).

Let's Code! (HTML and CSS)

Create **index.html** with the following links:

- [Sweetalert](#) for easily creating nice popups using JavaScript
- **index.css** for defining extra CSS (Cascading Style Sheets) to style your **index.html**
- **index.js** for JavaScript using in **index.html** to call **API** for **CRUD** operations.

```

<html lang="en">
  <head>
    <!-- Required meta tags -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>CRUD FROM API</title>
    <link rel="preconnect" href="https://fonts.googleapis.com">
    <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
    <link href="https://fonts.googleapis.com/css2?family=Roboto:wght@300;400&display=swap"
      rel="stylesheet">
    <link href="index.css" rel="stylesheet">
  </head>
  <body>

    <nav class="navbar navbar-dark bg-mynav">
      <div class="container-fluid">
        <a class="navbar-brand" href="#">My App</a>
      </div>
    </nav>

    <div class="container">
      <div class="d-flex bd-highlight mb-3">

```

```

      <div class="me-auto p-2 bd-highlight"><h2>Users</div>
      <div class="p-2 bd-highlight">
        <button type="button" class="btn btn-secondary"
          onclick="showUserCreateBox()">Create</button>
      </div>
    </div>

```

```

    <div class="table-responsive">
      <table class="table">
        <thead>
          <tr>
            <th scope="col">#</th>
            <th scope="col">Avatar</th>
            <th scope="col">First</th>
            <th scope="col">Last</th>
            <th scope="col">Username</th>
            <th scope="col">Action</th>
          </tr>
        </thead>
        <tbody id="mytable">
          <tr>
            <th scope="row" colspan="5">Loading...</th>
          </tr>
        </tbody>
      </table>
    </div>
  </div>

```

```

<script src="index.js"></script>
<script src="https://cdn.jsdelivr.net/npm/sweetalert2@11.0.16/dist/

```

```

  sweetalert2.all.min.js"></script>
  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.0.1/dist/js/bootstrap.bundle.min.js"
    integrity="sha384-gtEjrD/SeCtmISkJKNUaakMoLD0//ElJ19smozuhV6z3Iehds+3U1b9Bn9Plx0x4"
    crossorigin="anonymous"></script>
  </body>
</html>

```

Create **index.css** to define extra CSS *****(Cascading Style Sheets) for your **index.html**

```
*{
    margin: 0;
    padding: 0;
    box-sizing: border-box;
}

body {
    font-size: 1.25rem;
    background-color: #f6f8fa;
    font-family: 'Roboto', sans-serif;
}

h2 {
    display: block;
    font-size: 1.5em;
    margin-block-start: 0.83em;
    margin-block-end: 0.83em;
    margin-inline-start: 0px;
    margin-inline-end: 0px;
    font-weight: bold;
}

button {
    appearance: auto;
    writing-mode: horizontal-tb !important;
    text-rendering: auto;
    color: buttontext;
    letter-spacing: normal;
    word-spacing: normal;
    line-height: normal;
    text-transform: none;
    text-indent: 0px;
    text-shadow: none;
    display: inline-block;
    text-align: center;
    align-items: flex-start;
    cursor: default;
    box-sizing: border-box;
    background-color: buttonface;
    margin: 0em;
    padding: 1px 6px;
    border-width: 2px;
    border-style: outset;
    border-color: buttonborder;
    border-image: initial;
}

button, select {
    text-transform: none;
}

button, input, optgroup, select, textarea {
    margin: 0;
    font-family: inherit;
    font-size: inherit;
    line-height: inherit;
}

[type=button]:not(:disabled), [type=reset]:not(:disabled), [type=submit]:not(:disabled), button:not(:disabled) {
    cursor: pointer;
}

table {
```




```

        display: table;
        caption-side: bottom;
        border-collapse: collapse;
    }

tbody, td, tfoot, th, thead, tr {
    border-color: inherit;
    border-style: solid;
    border-width: 0;
}

thead {
    display: table-header-group;
    vertical-align: middle;
    border-color: inherit;
}

tbody {
    display: table-row-group;
    vertical-align: middle;
    border-color: inherit;
}

th {
    display: table-cell;
    vertical-align: inherit;
    font-weight: bold;
    text-align: -internal-center;
    text-align: start;
    border-bottom: 1px solid #000;
}

tr {
    display: table-row;
    vertical-align: inherit;
    border-color: inherit;
}

td {
    line-height: 3rem;
    display: table-cell;
    vertical-align: inherit;
}

.container {
    padding-left: 30px;
    padding-right: 30px;
    margin-right: auto;
    margin-left: auto;
    max-width: 1320px;
}

.btn {
    display: inline-block;
    font-weight: 400;
    line-height: 1.5;
    color: #212529;
    text-align: center;
    text-decoration: none;
    vertical-align: middle;
    cursor: pointer;
    -webkit-user-select: none;
    -moz-user-select: none;
    user-select: none;
    background-color: transparent;

```

```

border: 1px solid transparent;
padding: 0.375rem 0.75rem;
font-size: 1rem;
border-radius: 0.25rem;
transition: color .15s ease-in-out,background-color .15s ease-in-out,border-color .15s ease-in-out,box-shadow .15s e
}

.btn-secondary {
color: #fff;
background-color: #6c757d;
border-color: #6c757d;
}

.btn-outline-secondary {
color: #6c757d;
border-color: #6c757d;
}

.btn-outline-danger {
color: #dc3545;
border-color: #dc3545;
}

.d-flex {
display: flex!important;
}

.mb-3 {
margin-bottom: 1rem!important;
}

.p-2 {
padding: 0.5rem!important;
}

.me-auto {
margin-right: auto!important;
}

.bg-mynav {
background-color: #2c3e50;
position: relative;
display: flex;
flex-wrap: wrap;
align-items: center;
justify-content: space-between;
padding-top: 0.5rem;
padding-bottom: 0.5rem;
}

.container-fluid{
display: flex;
flex-wrap: inherit;
align-items: center;
justify-content: space-between;
padding: 6px;
}

.navbar-brand{
padding-top: 0.3125rem;
padding-bottom: 0.3125rem;
margin-right: 1rem;
font-size: 1.3rem;
text-decoration: none;
white-space: nowrap;

```

```

        color:#fff;
    }

.table-responsive {
    overflow-x: auto;
}

.table {
    --bs-table-bg: transparent;
    --bs-table-accent-bg: transparent;
    --bs-table-striped-color: #212529;
    --bs-table-striped-bg: rgba(0, 0, 0, 0.05);
    --bs-table-active-color: #212529;
    --bs-table-active-bg: rgba(0, 0, 0, 0.1);
    --bs-table-hover-color: #212529;
    --bs-table-hover-bg: rgba(0, 0, 0, 0.075);
    width: 100%;
    margin-bottom: 1rem;
    color: #212529;
    vertical-align: top;
    border-color: #dee2e6;
}

.table>thead {
    vertical-align: bottom;
}

.table>tbody {
    vertical-align: inherit;
}

.table>:not(caption)>* {
    padding: 0.5rem 0.5rem;
    background-color: var(--bs-table-bg);
    border-bottom-width: 1px;
    box-shadow: inset 0 0 9999px var(--bs-table-accent-bg);
}

```

Read operation (JavaScript)

Create file **index.js** to call an **API** for **CRUD** operations starting from **Read**.API URL: <https://www.mecallapi.com/api/users>

Method: GET

To read data with JavaScript, we will use AJAX (**A**synchronous **J**avaScript **A**nd **X**ML) technique to have our web site update asynchronously. Concretely we will have data exchange using API in the background. Thus, the data is retrieved, we will have our web page updated without doing a refresh.

We will use **XMLHttpRequest** to call an API for retrieving data in JSON and display/update such data on the web page. To update the data on the web page, we will manipulate HTML **DOM** (Document Object Model). In our code, we will update the data in table by referring to the id of the HTML data table element, namely (**mytable**)

```

function loadTable() {
    const xhttp = new XMLHttpRequest();
    xhttp.open("GET", "https://www.mecallapi.com/api/users");
    xhttp.send();
    xhttp.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200) {
            console.log(this.responseText);
            var trHTML = '';
            const objects = JSON.parse(this.responseText);
            for (let object of objects) {
                trHTML += '<tr>';
                trHTML += '<td>'+object['id']+'</td>';
                trHTML += '<td></td>';
                trHTML += '<td>'+object['fname']+'</td>';
                trHTML += '<td>'+object['lname']+'</td>';
                trHTML += '<td>'+object['username']+'</td>';
            }
        }
    }
}

```



```

        trHTML += '<td><button type="button" class="btn btn-outline-secondary" onclick="showUserEditBox('+object['id']+
        trHTML += '<button type="button" class="btn btn-outline-danger" onclick="userDelete('+object['id']+')>Del</butt
        trHTML += "</tr>";
    }
    document.getElementById("mytable").innerHTML = trHTML;
}
};
}

loadTable();









```

Open **index.html** on a web Browser, you will see the result:

My App

Users

Create

#	Avatar	First	Last	Username	Action
1		Karn	Yong	karn.yong@mecallapi.com	<div>EditDel</div>
2		Ivy	Cal	ivy.cal@mecallapi.com	<div>EditDel</div>
3		Walter	Beau	walter.beau@mecallapi.com	<div>EditDel</div>
4		Gayla	Bertrand	gayla.bertrand@mecallapi.com	<div>EditDel</div>
5		Benjamin	Chaz	benjamin.chaz@mecallapi.com	<div>EditDel</div>
6		Delia	Robin	delia.robin@mecallapi.com	<div>EditDel</div>
7		Hector	Graves	hector.graves@mecallapi.com	<div>EditDel</div>
8		Diego	Greene	diego.greene@mecallapi.com	<div>EditDel</div>

Create operation (JavaScript)

API URL:

<https://www.mecallapi.com/api/users/create>

Method: POST

Sample body (JSON):

```

{
  "fname": "Cat",
  "lname": "Chat",
  "username": "cat.chat@mecallapi.com",
  "email": "cat.chat@mecallapi.com",
  "avatar": "https://www.mecallapi.com/users/cat.png"
}

```

Add the following code in **index.js**

```

function showUserCreateBox() {
  Swal.fire({
    title: 'Create user',
    html:
      '<input id="id" type="hidden">' +
      '<input id="fname" class="swal2-input" placeholder="First">' +
      '<input id="lname" class="swal2-input" placeholder="Last">' +
      '<input id="username" class="swal2-input" placeholder="Username">' +
      '<input id="email" class="swal2-input" placeholder="Email">',
    focusConfirm: false,
    preConfirm: () => {
      userCreate();
    }
  })
}

```

```
function userCreate() {
  const fname = document.getElementById("fname").value;
  const lname = document.getElementById("lname").value;
  const username = document.getElementById("username").value;
  const email = document.getElementById("email").value;

  const xhttp = new XMLHttpRequest();
  xhttp.open("POST", "https://www.mecallapi.com/api/users/create");
  xhttp.setRequestHeader("Content-Type", "application/json;charset=UTF-8");
  xhttp.send(JSON.stringify({
    "fname": fname, "lname": lname, "username": username, "email": email,
    "avatar": "https://www.mecallapi.com/users/cat.png"
  }));
  xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      const objects = JSON.parse(this.responseText);
      Swal.fire(objects['message']);
      loadTable();
    }
  };
}
```

We access form values using the `value` property of DOM Elements

The `Swal.fire` is used for the popup which we are using with help of [Sweetalert](#)

preConfirm :

Function to execute before confirming, may be async (Promise-returning) or sync.

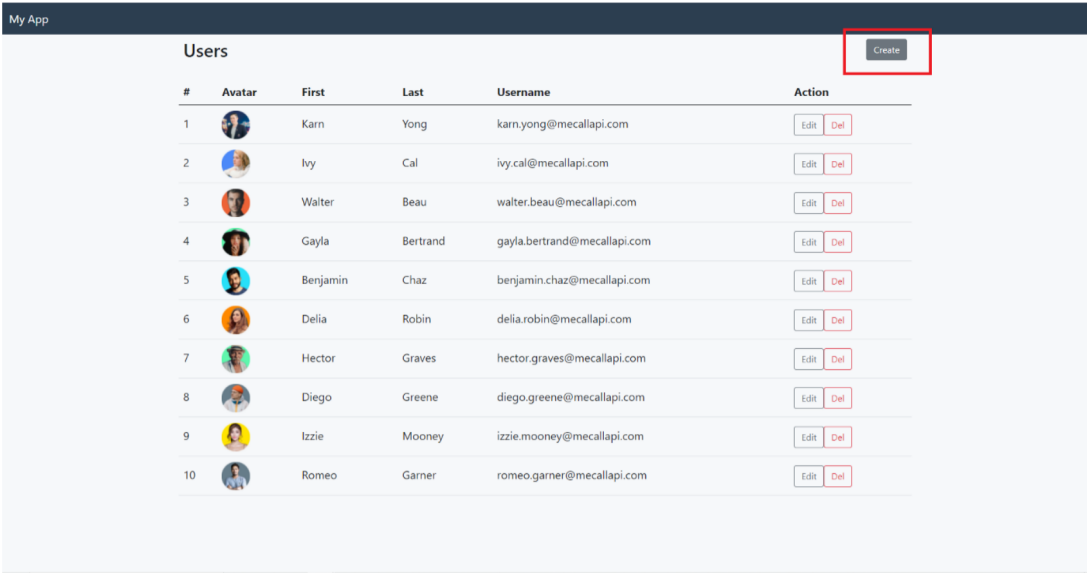
Returned (or resolved) value can be:

- **false** to prevent a popup from closing
- anything else to pass that value as the **result.value** of **Swal.fire()**
- **undefined** to keep the default **result.value**

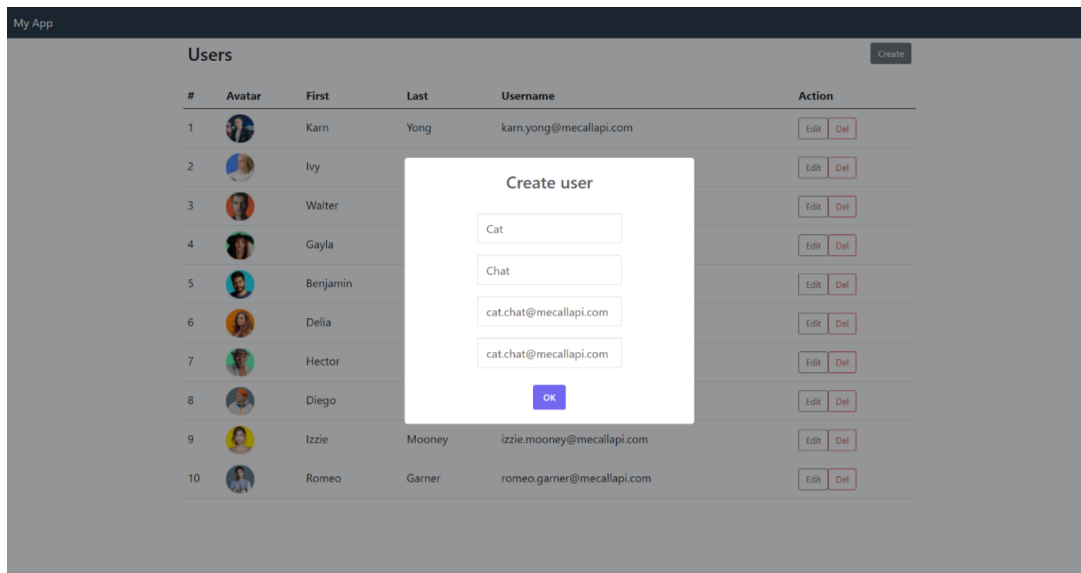
focusConfirm :

Set to **false** if you want to focus the first element in tab order instead of "Confirm"-button by default.

Refresh `index.html` and try to perform the create operation:



Click Create



Input Data

Users						Create
#	Avatar	First	Last	Username	Action	
1		Karn	Yong	karn.yong@mecallapi.com	Edit Del	
2		Ivy	Cal	ivy.cal@mecallapi.com	Edit Del	
3		Walter	Beau	walter.beau@mecallapi.com	Edit Del	
4		Gayla	Bertrand	gayla.bertrand@mecallapi.com	Edit Del	
5		Benjamin	Chaz	benjamin.chaz@mecallapi.com	Edit Del	
6		Delia	Robin	delia.robin@mecallapi.com	Edit Del	
7		Hector	Graves	hector.graves@mecallapi.com	Edit Del	
8		Diego	Greene	diego.greene@mecallapi.com	Edit Del	
9		Izzie	Mooney	izzie.mooney@mecallapi.com	Edit Del	
10		Romeo	Garner	romeo.garner@mecallapi.com	Edit Del	
11		Cat	Chat	cat.chat@mecallapi.com	Edit Del	

New Data Added

UPDATE operation (JavaScript)

API URL:

<https://www.mecallapi.com/api/users/update>

Method: PUT

Sample body (JSON):

```
{
  "id": 11,
  "fname": "Cat",
  "lname": "Gato",
  "username": "cat.gato@mecallapi.com",
  "email": "cat.gato@mecallapi.com",
  "avatar": "https://www.mecallapi.com/users/cat.png"
}
```

Add the following code in **index.js**

```
function showUserEditBox(id) {
  console.log(id);
  const xhttp = new XMLHttpRequest();
  xhttp.open("GET", "https://www.mecallapi.com/api/users/"+id);
```

```

xhttp.send();
xhttp.onreadystatechange = function() {
  if (this.readyState == 4 && this.status == 200) {
    const objects = JSON.parse(this.responseText);
    const user = objects['user'];
    console.log(user);
    Swal.fire({
      title: 'Edit User',
      html:
        '<input id="id" type="hidden" value='+user['id']+'>' +
        '<input id="fname" class="swal2-input" placeholder="First" value="'+user['fname']+'">' +
        '<input id="lname" class="swal2-input" placeholder="Last" value="'+user['lname']+'">' +
        '<input id="username" class="swal2-input" placeholder="Username" value="'+user['username']+'">' +
        '<input id="email" class="swal2-input" placeholder="Email" value="'+user['email']+'">',
      focusConfirm: false,
      preConfirm: () => {
        userEdit();
      }
    })
  }
};
}

function userEdit() {
  const id = document.getElementById("id").value;
  const fname = document.getElementById("fname").value;
  const lname = document.getElementById("lname").value;
  const username = document.getElementById("username").value;
  const email = document.getElementById("email").value;

  const xhttp = new XMLHttpRequest();
  xhttp.open("PUT", "https://www.mecallapi.com/api/users/update");
  xhttp.setRequestHeader("Content-Type", "application/json;charset=UTF-8");
  xhttp.send(JSON.stringify({
    "id": id, "fname": fname, "lname": lname, "username": username, "email": email,
    "avatar": "https://www.mecallapi.com/users/cat.png"
  }));
  xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      const objects = JSON.parse(this.responseText);
      Swal.fire(objects['message']);
      loadTable();
    }
  };
}

```












To update a particular user we will first retrieve his/her info from the same API and populate the update form with that info. Then we will send a PUT request using AJAX to update the info. And on successful request

Refresh **index.html** and try to perform the update operation:

My App

Users

Create












#	Avatar	First	Last	Username	Action
1		Karn	Yong	karn.yong@mecallapi.com	<div>EditDel</div>
2		Ivy	Cal	ivy.cal@mecallapi.com	<div>EditDel</div>
3		Walter	Beau	walter.beau@mecallapi.com	<div>EditDel</div>
4		Gayla	Bertrand	gayla.bertrand@mecallapi.com	<div>EditDel</div>
5		Benjamin	Chaz	benjamin.chaz@mecallapi.com	<div>EditDel</div>
6		Delia	Robin	delia.robin@mecallapi.com	<div>EditDel</div>
7		Hector	Graves	hector.graves@mecallapi.com	<div>EditDel</div>
8		Diego	Greene	diego.greene@mecallapi.com	<div>EditDel</div>
9		Izzie	Mooney	izzie.mooney@mecallapi.com	<div>EditDel</div>
10		Romeo	Garner	romeo.garner@mecallapi.com	<div>EditDel</div>
11		Cat	Chat	cat.chat@mecallapi.com	<div>EditDel</div>

Edit newly added user

My App

Users

Create

#	Avatar	First	Last	Username	Action
1		Karn	Yong	karn.yong@mecallapi.com	<div>EditDel</div>
2		Ivy			<div>EditDel</div>
3		Walter			<div>EditDel</div>
4		Gayla			<div>EditDel</div>
5		Benjamin			<div>EditDel</div>
6		Delia			<div>EditDel</div>
7		Hector			<div>EditDel</div>
8		Diego			<div>EditDel</div>
9		Izzie	Mooney	izzie.mooney@mecallapi.com	<div>EditDel</div>
10		Romeo	Garner	romeo.garner@mecallapi.com	<div>EditDel</div>
11		Cat	Chat	cat.chat@mecallapi.com	<div>EditDel</div>

Edit User












OK

Input Data

My App

Users

Create

#	Avatar	First	Last	Username	Action
1		Karn	Yong	karn.yong@mecallapi.com	<div>EditDel</div>
2		Ivy	Cal	ivy.cal@mecallapi.com	<div>EditDel</div>
3		Walter	Beau	walter.beau@mecallapi.com	<div>EditDel</div>
4		Gayla	Bertrand	gayla.bertrand@mecallapi.com	<div>EditDel</div>
5		Benjamin	Chaz	benjamin.chaz@mecallapi.com	<div>EditDel</div>
6		Delia	Robin	delia.robin@mecallapi.com	<div>EditDel</div>
7		Hector	Graves	hector.graves@mecallapi.com	<div>EditDel</div>
8		Diego	Greene	diego.greene@mecallapi.com	<div>EditDel</div>
9		Izzie	Mooney	izzie.mooney@mecallapi.com	<div>EditDel</div>
10		Romeo	Garner	romeo.garner@mecallapi.com	<div>EditDel</div>
11		Cat	Gato	cat.gato@mecallapi.com	<div>EditDel</div>

DELETE operation (JavaScript)

API URL:

<https://www.mecallapi.com/api/users/delete>

Method: DELETE

Sample Body (JSON):

```
{
  "id": 11
}
```



Add the following code in **index.js**

```
function userDelete(id) {
  const xhttp = new XMLHttpRequest();
  xhttp.open("DELETE", "https://www.mecallapi.com/api/users/delete");
  xhttp.setRequestHeader("Content-Type", "application/json;charset=UTF-8");
  xhttp.send(JSON.stringify({
    "id": id
  }));
  xhttp.onreadystatechange = function() {
    if (this.readyState == 4) {
      const objects = JSON.parse(this.responseText);
      Swal.fire(objects['message']);
      loadTable();
    }
  };
}
```

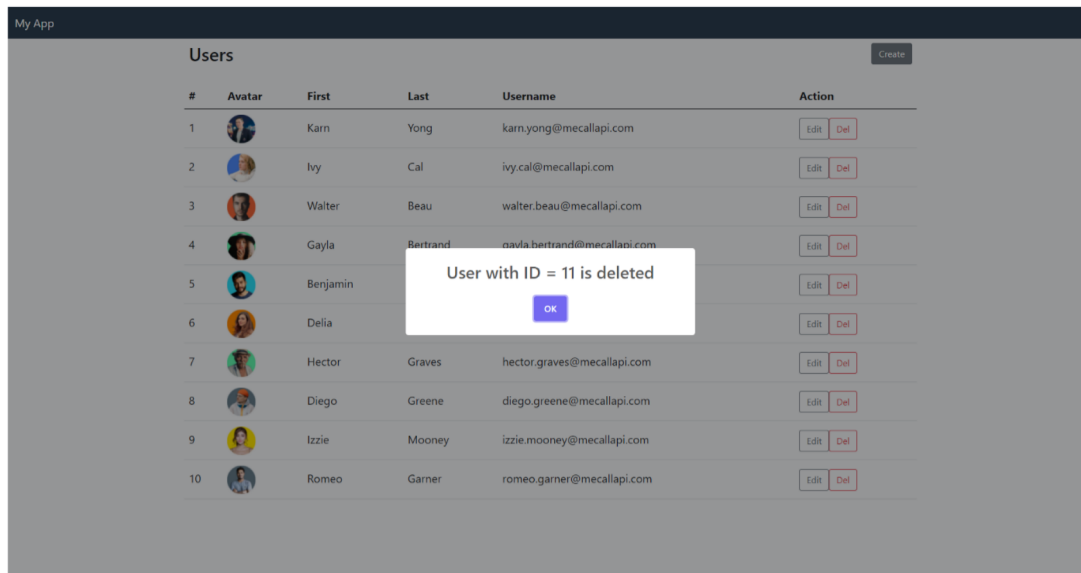


we make a delete AJAX Request and along send the id of the user to delete.

Refresh **index.html** and try to perform the delete operation:

My App

Users					Create
#	Avatar	First	Last	Username	Action
1		Karn	Yong	karn.yong@mecallapi.com	<input type="button" value="Edit"/> <input type="button" value="Del"/>
2		Ivy	Cal	ivy.cal@mecallapi.com	<input type="button" value="Edit"/> <input type="button" value="Del"/>
3		Walter	Beau	walter.beau@mecallapi.com	<input type="button" value="Edit"/> <input type="button" value="Del"/>
4		Gayla	Bertrand	gayla.bertrand@mecallapi.com	<input type="button" value="Edit"/> <input type="button" value="Del"/>
5		Benjamin	Chaz	benjamin.chaz@mecallapi.com	<input type="button" value="Edit"/> <input type="button" value="Del"/>
6		Delia	Robin	delia.robin@mecallapi.com	<input type="button" value="Edit"/> <input type="button" value="Del"/>
7		Hector	Graves	hector.graves@mecallapi.com	<input type="button" value="Edit"/> <input type="button" value="Del"/>
8		Diego	Greene	diego.greene@mecallapi.com	<input type="button" value="Edit"/> <input type="button" value="Del"/>
9		Izzie	Mooney	izzie.mooney@mecallapi.com	<input type="button" value="Edit"/> <input type="button" value="Del"/>
10		Romeo	Garner	romeo.garner@mecallapi.com	<input type="button" value="Edit"/> <input type="button" value="Del"/>
11		Cat	Gato	cat.gato@mecallapi.com	<input type="button" value="Edit"/> <input type="button" value="Del"/>



That's a wrap! Hopefully, this section will help you understanding the basic of developing web application by using just HTML, CSS, and JavaScript.

Conclusion

So till now we learnt basics of HTTP and AJAX and at last we combined that knowledge to build a webpage using an API

In this module we have learned about:

- What is HTTP
- What is AJAX
- Different AJAX APIs and Libraries
- Building a web page using API

Thank You !