

# Summer Coding Bootcamp for Beginners

Christopher Ryu

M.S in Software Engineering (MSE)

<http://mse.fullerton.edu>

## The purpose of this Bootcamp

This Bootcamp is primarily designed to help **current** or **prospective MSE students** learn modern application development techniques, technologies, and tools to experience a **bird's-eye and vertical view** of an application so that they can better understand the subjects that will be discussed in the MSE program and ultimately be successful in their careers.

## Expected audience

This tutorial is written primarily for **beginners**, especially those who wish to learn the development techniques, technologies, and tools required for web or mobile applications. This tutorial may also be helpful to those who want to learn application development using APIs, web services, serverless computing, microservice architecture, CI/CD practices, and related toolchains.

## Acknowledgment

This tutorial could not be written without relying on the experience and knowledge voluntarily shared by numerous people. I tried to cite all the sources to express my gratitude to them for helping each other learn better. However, given the short period to prepare this tutorial, I most likely missed many of those valuable resources I accessed. They all deserve enormous credit for helping our learning and improving our society.

## Table of Contents

### 1. Web and Mobile Applications

- 1.1. HTML, CSS, and JavaScript
- 1.2. Continuous Integration and Continuous Deployment (CI/CD)
- 1.3. Introduction to Node.js
- 1.4. Introduction to React.js framework for frontend development
- 1.5. Introduction to Express.js for backend development
- 1.6. Introduction to Nex.js framework for backend development

### 2. API, Web Service, and Serverless Computing

- 2.1. API, REST API, and web service
- 2.2. GraphQL
- 2.3. Serverless computing with AWS Lambda

### 3. Microservices and Microservice Architecture

### 4. Deploying Microservices

# 1. Web and Mobile Applications

## 1.1. HTML, CSS, and JavaScript


**HTML** (HyperText Markup Language) is the standard markup language for documents designed to be displayed in a web browser with the support of Cascading Style Sheets (CSS) and JavaScript. **CSS** is a style sheet language used for describing the presentation of a document in a markup language such as HTML.

**JavaScript (JS)** is the most popular programming language, originally developed for a web application, making it interactive, and evolved to be a language that can be used for web, mobile, and other application development.

### Study guide for HTML, CSS, and basic JavaScript

- Review various **HTML tags**, create HTML files with those tags, and open it with a browser.
- Review **CSS** and apply some CSS to an HTML file and see how CSS changes the presentation of an HTML document.
- **Review the basic JavaScript** syntax, variables (data types, operators, scope, assignment, de-structuring assignment), string, conditional statement, function, arrow function  $\Rightarrow$ , function as an object, built-in functions (math, dates, etc.), built-in data structures such as array ([a list of indexed values]), array list (push, pop, etc.), objects ({key: value} like a dictionary, built-in objects such as Number, Math, Dates, etc.), set, loops statements (while, for, break/continue, for in, for of, forEach, map, filter, reduce), template literals with back-ticks, try-catch block for error handling, event and event handler, DOM, modules, various ways of exporting and importing modules (default, named, and combination), modules, exporting and importing modules (default, named), JSON (JavaScript Object Notation), class and object.
- Review **TypeScript**. Understand the difference between JavaScript and TypeScript.

### Preparation for web or mobile application development

- **Download and install** an Interactive Development Environment (IDE) such as [Visual Studio Code](#). Note: Visual Studio Code (**VS Code**) is not the same as Microsoft Visual Studio. Although it is developed by Microsoft using a **JavaScript**-based desktop framework, [Electron](#), it is a free **cross-platform** IDE that supports a variety of programming languages and extensions for additional functionalities. In this lesson, we will use VS Code as our IDE for examples and demos.
- In VS Code, **add an extension** called ‘[live server](#)’ by clicking the extension button on the left menu bar and typing “live server” keyword on the search box, and installing it. To use ‘live server’, right-click the file and open with the “live server.” Other **recommended extensions** are Auto Rename Tag, TabOut, JavaScript (ES6) code snippets, gitignore, ES7 React/Redux/GraphQL/React-Native snippets.
- **Click Help** to learn VS Code commands, editor features, and keyboard shortcuts to improve your code typing productivity. For example, when you want to create an HTML file, **typing “!”** in an HTML editing panel will automatically create an HTML template for you.
- **Download [Node.js](#)** (Node JS or Node) from [nodejs.org](#) and **install** it. Node is based on JavaScript. It comes with the node package manager (**npm**). The node can be used as a JavaScript runtime (to run JavaScript code) without a browser. We will learn Node.js later in this tutorial.
- **To verify the installation of VS Code and Node**, create a folder named “Bootcamp” (either “mkdir” command in a terminal or clicking  in VS Code, open and choose the folder “Bootcamp” (File ->

Open folder -> Bootcamp), add a new file named “hello.js”, and type the following lines of JavaScript code:

```
//hello.js
```

```
console.log("Hello, Node!");
```

- To execute ‘hello.js’ using **Node**, type the following code at a **terminal** from Visual Studio Code:

```
$ node hello.js
```

If everything is fine with your code and Node installation, you should see the message “Hello, Node!”.

### Some tutorial sites for HTML, CSS, and JavaScript:

- [https://developer.mozilla.org/en-US/docs/Learn/Getting\\_started\\_with\\_the\\_web](https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web)
- <https://www.w3schools.com>
- <https://javascript.info/>

**JavaScript** is a lightweight script, interpretive, and event-driven language. It enables you to dynamically update both HTML and CSS and allows client-side validation, manipulation of HTML tags, and interactive web pages.

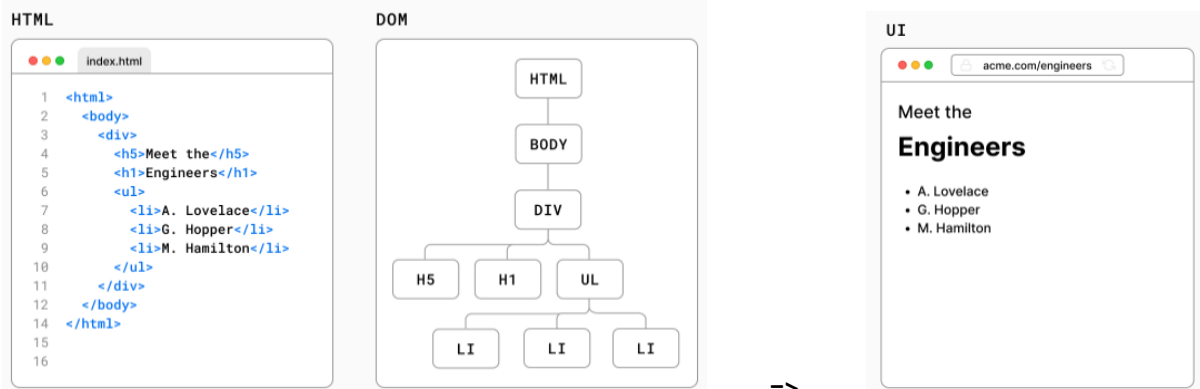
To understand how a HTML page with JavaScript is displayed in a browser, create a HTML file with JavaScript codes using `<script></script>` tag (type “!” key in the VS code editing panel), embed a JavaScript code within `<script>` tag. There are various ways of including JavaScript code in a HTML document, e.g., inline JavaScript code using `<script>` tag and external JavaScript files, e.g., `<script src="test_script.js"></script>` or `<script src="http://www.w3schools.com/js/myScript.js"></script>`.

### How can I test and debug HTML documents and JavaScript codes?

- To test an HTML file, open the file with a browser and see if it shows what you expected.
- To test JavaScript code, you can use a developer tool available in the **browser** (since it includes a **JavaScript runtime**) or **Node** (with JavaScript runtime) command ‘**node hello.js**’ where hello.js is a JavaScript file name. JavaScript built-in function “**console.log**” is used to display the value of a variable like print functions in many other languages or you can use [Node.js debugging in VS Code](#) using built-in support for JavaScript.

### HTML DOM (Document Object Model)

When a user visits a web page, the server returns an HTML file to the browser. The browser then reads the HTML file and creates a DOM of the page. The HTML DOM is constructed as a tree of objects that represents the HTML elements, is used to render user interfaces on the browser, as shown [below](#):



**HTML DOM** defines the HTML elements as **objects**, the **properties** of all HTML elements, the **methods** to access all HTML elements, and the **events** attached to all HTML elements. The HTML DOM is a standard object model and programming interface for HTML to get, change, add, or delete HTML elements. You can use DOM methods with a programming language like JavaScript to listen to the user events and manipulate the DOM by selecting, adding, updating, and deleting specific elements in the user interface.

### Browser Object Model (BOM)

BOM allows JavaScript to communicate with browsers. There are no official standards for BOM.

Some BOM objects include Windows, Screen, Location, History, Navigator, Popup, Timing, Cookies, etc.

### The event, event listener, and event handling

Events are signals fired inside an application (e.g., browser, mobile, or desktop apps) that notify of changes (by user action, updated data, time, etc.) in the application or environment. An event handler is a function or program to execute when the event occurs. The event handling program is common in applications (Web, mobile, or desktop) that interact with the users.

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

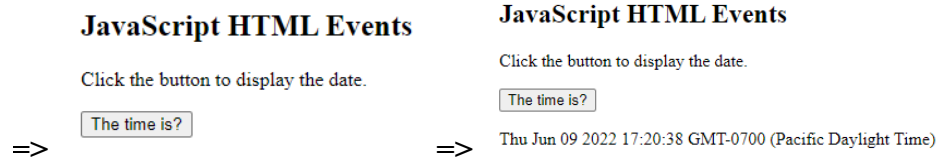
<h2>JavaScript HTML Events</h2>
<p>Click the button to display the date.</p>

<button onclick="displayDate()">The time is?</button>

<script>
function displayDate() {
  document.getElementById("demo").innerHTML = Date();
}
</script>

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

</body>
</html>
```



### Best practices for JavaScript coding

- Avoid global variables and declare local variables if possible
- Declare variables and initialize them at the top of every scope to avoid undefined values and to provide a single place to initialize variables. Accessing undeclared variables will be “undefined”.
- Understand JavaScript hoisting (default behavior of JavaScript) refers to the process whereby the interpreter appears to move the declarations of functions, variables, or classes to the top of their scope regardless of their scope prior to execution of the code. This may create an issue with undeclared variables when values are assigned to the variables, as all those undeclared variables will be implicitly created as global variables.
- Beware of automatic type conversions. Note that a variable can contain all data types. A type of variable can be changed through an assignment of different types of values. This can cause a run-time error as JavaScript is an interpretive language (or script), unlike a compiler language such as Java, C#, or C++.
- Declare objects and arrays with `const` to avoid any accidental change of type.
- Use default parameter values for a function if possible.
- For equality comparison operator, use `===` instead of `==` as `==` operator always convert to matching types before comparison. `===` operator forces comparison of both value and type.
- Be careful using `eval()` function for security reason.

## Exercise

- Review the above tutorial sites, and follow the examples to learn HTML, CSS, and basic JavaScript programming.
- Create an HTML document with JavaScript codes, manipulate it, and understand how it works.

## What are the required techniques to develop a Web or mobile application?

- HTM, DOM (Document Object Model), CSS (Cascading Style Sheet)
- JavaScript
- JSON (JavaScript Object Notation)
- **Frontend development** using HTML, CSS, JavaScript, and front-end framework (typically based on JavaScript such as React.js, Angular.js, Bootstrap, etc.
- **Backend development** using back-end frameworks such as Node.js, Express, Next (based on JavaScript and Node.js), Django (Python), Flask (Python), ASP.NET (C#), Spring (Java), PHP, etc.
- DBMS (Database Management System) such as SQLite, MongoDB, MySQL, PostgreSQL, Oracle, SQL Server, etc.
- Web server
- Internet, TCP/IP, HTTP (computer networking)
- Mobile devices such as Android, iOS (for mobile applications)
- Application Programming Interface (API), web services
- Version control
- Continuous integration and continuous deployment (CI/CD)
- Cloud services such as AWS, Azure, GCP, etc.
- Other techniques and supporting tools to improve the coding productivity

## Study guide for web or mobile application concepts

- Understand how web applications work, including the protocol, communication mechanism between browser (client) and servers (web server, database server, and other backend servers), request, and response.
- Understand the concepts of frontend and backend, differences between web and mobile applications, and technology environment.
- Understand the required infrastructure, technologies, development techniques, and toolchains.

## Mobile application

Mobile applications can run directly on their mobile device or remote servers such as Web servers through the Internet using web services.

## Frontend, backend, and Full Stack

- **Frontend** means client-side in an application, typically developed using HTML, CSS, JavaScript, React.js, Angular.js, etc. **Backend** means server-side, typically created by the developers using JavaScript, Python, C#, etc.
- **Full Stack** means both frontend and backend.

## The required techniques and technologies for frontend application development

- **HTML** and **DOM**
- **CSS** (Cascading Style Sheet)

- **JavaScript**
- **Frontend application frameworks** include React.js, Bootstrap, Angular.js, and Electron (for desktop applications). These are all JavaScript-based frameworks. A **software framework** is a structure (with built-in software patterns) and platform that provides a foundation for developing software applications. The use of a software framework improves efficiency (coding productivity).
- Other related techniques and tools

### The required techniques and technologies for backend application development

- HTML, CSS, DOM, JavaScript, understanding frontend framework
- Backend application framework such as Node, Express and programming languages such as JavaScript, Python, C#, etc.
- JSON, files
- DBMS, database design, query language such as SQL
- Web servers, HTTP, API, web services
- Other related techniques and tools

### Some problems with JavaScript

JavaScript is one of the most popular and powerful languages. However, error detection and debugging are complicated since it's an interpretive (script) language; **no type checking** is done until run time. Even simple syntax errors are not detected until the related statements are executed during the runtime if you don't use an intelligent IDE.

### TypeScript is a strongly typed JavaScript

Anders Hejlsberg (C# designer) at Microsoft developed **TypeScript** as a **compiler language** in 2012. The stable version was released in March 2022. It is a typed **superset of JavaScript** with additional features. TypeScript is an object-oriented language supporting classes, interfaces, and strongly types like C# or Java. File extension for TypeScript can be .ts or .tsx. Since it is compiled to JavaScript, **it is supported by all browsers**.

To install the TypeScript compiler:

```
npm install -g typescript or npm install typescript --save-dev
```

“**-g**” option is to specify the global configuration which sets the package install location to the folder where you installed Node. “**--save-dev**” option is to save the package under “devDependencies” in “package.json” file.

This option is useful when you are installing only development packages (not for production).

To verify if it's installed properly:

```
tsc --v
```

To write a TypeScript code, write a program and save it into .ts extension, 'test.ts'.

To compile a TypeScript file 'test.ts':

```
tsc test.ts
```

With Visual Studio Code, right-click the file 'test.ts', open in command prompt option, then compile it.

The compilation will result in a JavaScript file, e.g., 'test.js' in this case.

To run the 'test.js', either use Node (e.g., node test.js) or browser by including <script> tag in an HTML file.

## Advanced JavaScript programming

JavaScript is the most popularly used programming language. JavaScript was originally started as a client-side web programming language, but it has evolved to a general-purpose programming language that can be used for both frontend and backend, plus mobile, desktop, embedded, and other applications. Unlike many other interpretive languages, JavaScript is fast and efficient. Many application frameworks, such as **React**, Angular, Vue (front-end), **Node**, **Express**, and **Next** (backend), **React Native** (for mobile), and **Electron** (desktop), are all developed using JavaScript.

### Study guide

- Understand how **callback functions** work.
- Understand how **synchronous** and **asynchronous function calls** (Promise, Async/Await) work.
- Understand **HTTP protocol**.
- Understand the **request-response** mechanism.
- Understand **CORS** (cross-origin resource sharing) policy.
- Understand how the **fetch API** works.
- Understand the role of a **proxy server**.

### Callback function

A **callback** is a function passed as an argument to another function, which is then invoked inside the outer function to complete some kind of routine or action. This is possible since functions are objects with IDs or pointers in JavaScript. The callback can be synonymous (blocking) or asynchronous (non-blocking). A **synchronous callback** is executed immediately, whereas the execution of an **asynchronous callback** is deferred to a later point in time.

The following code is an **example of a synchronous callback**. Run this code by 'node sync\_callback.js'.

```
//sync_callback.js
function greeting(name) {
  //alert('Hello ' + name);
  console.log('Hello ' + name);
}
function processUserInput(my_callback) {
  //var name = prompt('Please enter your name.');
```

```
var name = 'Bob';
my_callback(name);
}
processUserInput(greeting);
```

The following code is synchronous function call:

```
//sync_calls.js
function fun1() {
  console.log('Function 1');
```

```
fun2();
}
```



```
function fun2() {
  console.log('Function 2');
  fun3();
}
function fun3() {
  console.log('Function 3');
}
fun1();
fun2();
fun3();
```



**JavaScript code is synchronous and single-threaded**, but allowing various callback mechanisms. The **JavaScript engine runs the code synchronously** in the execution context. Each function call is added to a stack called ‘**call stack**’ and popup when it completes.

### Asynchronous function call

A JavaScript function call is **only asynchronous** (non-blocking) **when it relies on something outside the JavaScript engine**, such as browser functions (timer including `setTimeout`, `setInterval`), events (user interactions like a mouse click), API calls (such as Ajax calls, browser API, HTTP Request), therefore it will take a long time, we don’t know when the function will complete, or actions that might happen at any time, basically for handling more than one call at a time). So, in this situation, the asynchronous function call makes sense. If you use other programming languages, you have to write your own code to deal with these asynchronous situations (most likely using multi-threaded programming). **But JavaScript code is synchronous, how can it be asynchronous? We use callback function for asynchronous tasks.** With the idea of callback function, you can make a regular function asynchronous. But keep it mind that callback itself is not automatically asynchronous. In addition to using callback, JavaScript also supports “**worker thread**” (spawn from the main thread). Other than these, the main body of the JavaScript code is synchronous, but it is still very efficient without using any multi-threading because of the way the JavaScript runtime engine is designed.

The following code is an **example of asynchronous callback and synchronous calls**:

```
//sync_async.js
function async_fun() {
  console.log('I am an async function.');
```

```
}
function fun1() {
  console.log('Function 1');
  fun2();
}
function fun2() {
  console.log('Function 2');
  fun3();
}
function fun3() {
  console.log('Function 3');
```



```

}
setTimeout(async_fun, 1000); //Browser API setTimeout executes async_fun after 1 sec. elapsed
//setTimeout(()=>{console.log('I am an async function.'), 1000}); //using => function
fun1();
fun2();
fun3();

```

The expected output from this program is:

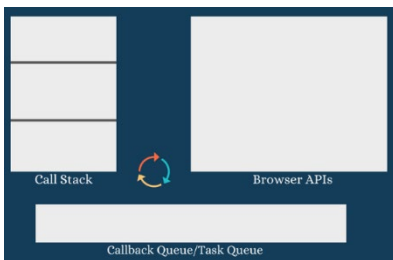
```

Function 1
Function 2
Function 3
Function 2
Function 3
Function 3
I am an async function.

```

Here is how the above program produced the result:

[A diagram for Call Stack and Callback Queue](#)



When functions `fun1()`, `fun2()`, and `fun3()` are called, they are put in **Call Stack** (LIFO). `Async_fun()` is passed to `setTimeout()` as a callback function that is put in **Callback Queue** (or **Event Queue**) and will be notified in the order of the events that occur. Whenever the Call Stack is empty, the JavaScript engine periodically looks at the Callback Queue (FIFO) to see if there is an event to be notified about. If so, it pulls the callback function and moves to the Call Stack when the Call Stack is empty. This loop continues. This loop is called “**Event Loop**”. For example, a callback function passed to `setTimeout()` is put in the Callback Queue, and the Event Loop checks if the Call Stack is empty. If empty, it pushes the callback function from Callback Queue to the Call Stack, and the callback function gets removed from the Callback Queue and executed.

Here is another example with an event and event handler:

```

//event_test.js
function waitfunction() {
    var a = 5000 + new Date().getTime();
    while (new Date() < a){}
    console.log('waitfunction() context will be popped after this line');
}

function my_clickHandler() {
    console.log('This is a click event handler.');
```

//my\_clickHandler is a callback function that is added to click event.

```

document.addEventListener('click', my_clickHandler);
waitfunction(); //This function is placed on the Call Stack
console.log('Global context from Call Stack will be popped after this line.');
```

Test this HTML with a browser, enable the debugging tool, and open it with ‘Live Server’ (right-clicking the HTML file in VS code), and click the browser to generate a click event.

```
<html>
  <head></head>
  <body>
    <h1>Hello!</h1>
    <script src="event_test.js"></script>
  </body>
</html>
```

Now, we can expect a situation in which we have complex nested synchronous and asynchronous callbacks where each callback takes arguments that have been obtained as a result of previous callbacks. This kind of callback structure leads to the problem of readability and maintainability, called ‘**Callback hell**.’

We can **avoid the complexity of nested callbacks** with the idea of **Promise**, which avoids writing the nested callbacks, **making the callbacks easier to read**. Promise in JavaScript is a way to handle asynchronous operations **by making them synchronous**. With Promise, an asynchronous method returns a **Promise object** that can return a final value when it is available in the future. The mechanism is similar to a promise in real life (promising something in the future). With this mechanism, we can defer the execution of a code block until an async request is completed.

The **syntax to create a promise** is as follows:

```
// "Producing Code" that may take some time
let myPromise = new Promise(function(myResolve, myReject) {
  myResolve(); // conditional logic to deal with resolve when successful
  myReject(); // conditional logic to deal with reject with when error
});

// "Consuming Code" that must wait for a resolved (fulfilled) Promise or rejected (failed) Promise
myPromise.then(
  function(value) { /* code to execute when resolved or successful */ },
  function(error) { /* code to execute when rejected or failed or error */ }
);
```

**An example with a simple Promise:**

```
<!DOCTYPE html>
<html lang="en">
<body>
  <h1 id="demo"></h1>
  <script>
    let promise = new Promise((resolve, reject) => {
      resolve("A message from promise!");
    });
    promise.then((result) => document.getElementById("demo").innerHTML = result);
  </script>
</body>
</html>
```

The following code is a simple example of callback:

```

<!DOCTYPE html>
<html>
<body>
  <h2>JavaScript Callback using SetTimeout()</h2>
  <p>Wait 3 seconds (3000 milliseconds) for this page to change.</p>
  <h1 id="demo"></h1>
<script>
  setTimeout(function() { myFunction("A value passed to my function as a callback."); }, 3000);
  function myFunction(value) {
    document.getElementById("demo").innerHTML = value;
  }
</script>
</body>
</html>

```

The following code is an example using Promise that is equivalent to the above code:

```

<!DOCTYPE html>
<html>
<body>
  <h2>JavaScript Promise</h2>
  <p>Wait 3 seconds (3000 milliseconds) for this page to change.</p>
  <h1 id="demo"></h1>
<script>
  const myPromise = new Promise(function(myResolve, myReject) {
    setTimeout(function(){ myResolve("A value passed to my function as resolve."); }, 3000);
  });
  myPromise.then(function(value) {
    document.getElementById("demo").innerHTML = value;
  });
</script>
</body>
</html>

```

**A Promised object can be in one of the three states:**

- Pending: initial state before the Promise
- Resolved: Completed Promise
- Rejected: Failed Promise, throw an error

A **Promise** is made of pending (resolve, reject): **resolve.then.then.finally** and **reject.catch.finally**. The promise is used to keep track of whether an asynchronous event has been executed or not and determines what happens after the event has occurred.

**A complex situation with Promise**

Sometimes, we need to call multiple asynchronous requests. After the first Promise is resolved (or rejected), a new process will start to attach another Promise, chaining them.

**An example of Promise chaining:**

```

<!DOCTYPE html>
<html lang="en">
<body>
  <h1 id="demo1">Demo</h1>
  <h1 id="demo2"></h1>
  <script>
    let promise1 = new Promise((resolve, reject) => {
      resolve("A message from promise 1!");
    });
    promise1.then(function(msg1) {
      document.getElementById("demo1").innerHTML = msg1;
      return new Promise(function(resolve, reject) {
        setTimeout(function() {
          resolve('A message from inner promise!');
        }, 1000);
      });
    }).then(function(msg2) {
      document.getElementById("demo2").innerHTML = msg2;
    })
  </script>
</body>
</html>

```

As you can see in the above example, a **chain of Promise objects** can still be complicated to understand. JavaScript provides a simpler syntactic mechanism called “**async/await**” to relieve this complexity. JavaScript **async** makes a function return a **Promise**. Other values (non-promise) are wrapped in a resolved promise automatically. The following code is based on Promise:

```

function myFunction() {
  return Promise.resolve("A message from async function");
}
myFunction().then(function (msg) {
  console.log(msg);
});

```

The following code using **async** is equivalent to the above code using **Promise**:

```

//myFunction returns a Promise object
async function myFunction() {
  return "A message from async function";
}
p = myFunction();
p.then(function(msg) { //equivalent to p.then((msg) => console.log(msg));
  console.log(msg)
});

```

The return value of **async** is a **Promise** which will be resolved with the value returned by the **async** function or rejected with an exception thrown from or uncaught within. As you can see, **async()** makes the coding simpler.

To make an asynchronous call to a synchronous call (because asynchronous calls in certain cases are default in JavaScript), you can add “**await**” to **async** since **await** will make a function wait for a Promise. The “**await**” makes JavaScript wait until that promise settles and returns its result. The “**await**” works only inside “**async**”. Now, **async with wait** makes an asynchronous function call look more like a synchronous function but much easier to read. The following code is an example of using **async** and **await**:

```
async function myFunction() {
  let myPromise = new Promise(function(resolve, reject) {
    resolve("A message from myFunction");
  });
  msg = await myPromise;
  console.log(msg);
}
myFunction();
```

By the way, in addition to Callback Queue, the **JavaScript engine** has another queue called “**Microtask Queue**”. It is like the **Callback Queue** (another Queue for callback functions), but it has a **higher priority**. All the callback functions coming through Promise (also, **async/await**) and Mutation Observer will go inside the Microtask Queue. Promise handling always has a higher priority. So the JavaScript engine executes all the tasks from Microtask Queue first, then moves to the Callback Queue.

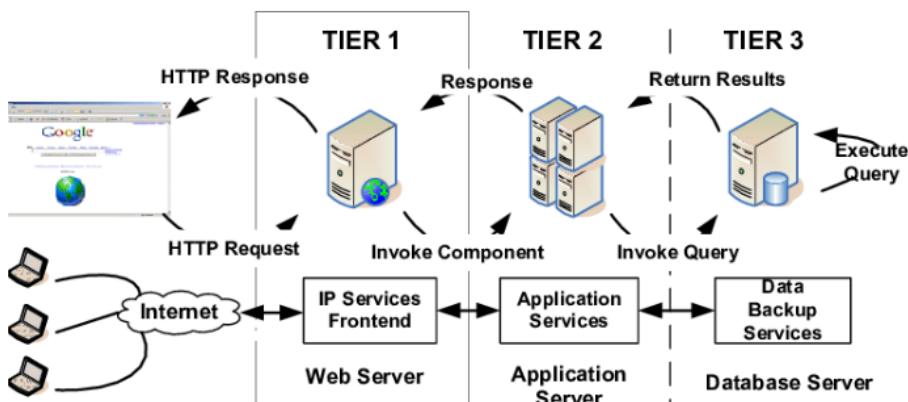
## How does a web or mobile application work?

A HTML page consists of multiple elements such as HTML tags, CSS, JavaScript, etc. A web browser is a complex software application used to locate, retrieve, and display content, including HTML pages, images, videos, and other files. It can also interpret JavaScript, applications, and other content. Browser extensions or offer plug-ins extend the capabilities of the browser. **Major components of the browser** are:

- The user interface
- Rendering engine
- Networking
- JavaScript interpreter

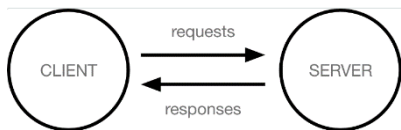
When the user enters a URL, the browser (client) contacts the DNS server to find the server (a special computer providing services to the client computers) to connect. This process is called DNS resolution. Once the browser has identified the server, it will initiate a TCP connection and begin the communication using **HTTP** (Hypertext Transfer Protocol) by sending a **request** and the server replying back with a **response**.

The following diagram illustrates the flow of HTTP requests and responses:



The **HTTP** is an application layer protocol in the Internet protocol suite model. It functions as a request-response in the **client-server model**. For example, a web browser (or other application) may be the client, whereas a web server running on a computer or hosting websites may be the server. The client submits an HTTP request message to the server. The server provides resources such as HTML files and other contents or performs other functions on behalf of the client and returns a response message to the client.

**Apps** or **browsers** on desktop or mobile computers are examples of **clients**. A **web server** is a machine that runs a software package, e.g., Apache, IIS, NGINX. A **database server** is a machine that runs a **Database Management System (DBMS)**. Oracle, MySQL, SQL Server, PostgreSQL, and MongoDB are examples of DBMS. Databases are created with a DBMS. In some cases, applications running on a separate server is called an **application server**.



An **HTTP client** initiates a **request** by establishing a **TCP connection** to a particular port on a server (e.g., 80). An HTTP server listening on that port waits for a client's request message. Upon receiving the request, the server finds the requested resource. If a request includes data from a database, it will spawn a process to connect to a DBMS and retrieve the data from the database. Once all the resources are ready for the client, it sends a **response** including a status line like "OK" and a message including the requested data. The **connection** between client and server is generally **closed** after a single request/response pair (or could be reused for other requests to improve the server performance). This request and response cycle continues.

### HTTP message header and body

The request and response message consists of a header and body. The **header** includes the message's context and metadata about the request or response. The **body** contains the data, called "**payload**," transmitted in an HTTP transaction message immediately following the headers.

The message body part is optional for an HTTP message, but it's the one that carries actual request data (including form data, uploaded files, etc.) If the body is associated, the header includes information about the data such as Content-type, Content-length, etc.

### Request header and body

The request header contains information about the resource to be fetched or the client itself. A basic request with one header may look like this:

```
<request method> URL <http version>
Host: <host address>
<optionally body>
```

**Request methods** include GET, POST, PUT, DELETE, PATCH, HEAD, CONNECT, etc. URL specifies the location of the resources being requested.

For example, when the user enters an URL (Unified Resource Locator) for an image on "example.com", the header may look like this:

```
GET /images/logo.jpg HTTP/1.1
Host: example.com
```

**URL** has a format of protocol\_schme://server\_address:port/target\_resource, e.g.,  
http://mse.fullerton.edu/about.

The `target_resource` can be specified by a query string. A **query string** is a part of a URL that assigns values to specified parameters as a series of field-value pairs, visible on the browser. For example, a URL containing a query string “`http://example.com/about?name=bob&age=20`” pass string values “`name=bob` and `age=20`” on the page “`about`” to the server.

The **GET** and **POST** are the most commonly used methods to fetch a resource from the server. **HTML form** can be set to use either GET or POST method, e.g., `<form method="GET" action="foo.js"> ...</form>` or e.g., `<form method="POST" action="foo.js"> ...</form>`.

**When the GET method is used** in a form, the HTTP request sends the form fields and values as a **query string** that is a part of the URL to the server (visible to the user on the browser). In this case, the body of a GET request is empty. Sending data through a query string is fast but may not be secure and has a size limitation.

**When the POST method is used**, the browser sends the request for data to the server in the **message body** (not visible to the user). The data sent via the POST method is relatively secure and does not have a size limitation.

**To retrieve the data sent by the browser**, you have to write a **backend program** to parse the data depending on the request method and send the requested data back to the browser.

### Response header and body

A server sends a response message to the client. The response message consists of a header including the protocol version and status code) and body. The response header contains information about the response, like its location or about the server itself. For example, a response header and body may look like this:

```
HTTP/1.1 200 OK
```

```
Content-Type: text/html
```

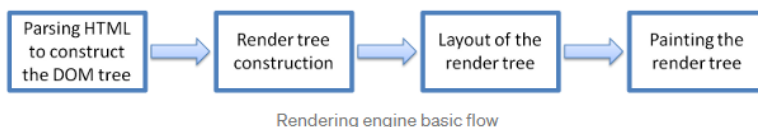
```
Content-Length: 155
```

```
<html>
  <head>Hello</head>
  <body><p>Hello, World</p></body>
</html>
```

You can find more information about HTTP [here](#) or [here](#).

### Rendering

Rendering content is one of the most important functions of a browser. When the browser receives the response from the server, it parses the content (and interprets code, e.g., if there is JavaScript code), constructs the DOM tree and layout, and renders the content on the browser window. The following diagram illustrates the flow:



Each browser uses a **rendering engine** (e.g., Blink for Chrome, EdgeHTML for Microsoft Edge) to parse and interpret the HTML, CSS (styling the content), and JavaScript by constructing the DOM tree and displaying it on a computer screen.

### Stateless connection



**HTTP is a stateless protocol.** So, the connection **between the client and web server is stateless**. A stateless connection does not require the server to retain information or status about each user for the duration of multiple requests. That means that when you load a page in your browser and then you navigate to another page on the same website, neither the server nor the browser has any way of knowing that it's the same browser visiting the same site. The main advantage of the stateless protocol is **scalability**. However, you can imagine many problems such as login and processing data on the page, etc. **To maintain a state** to associate a request to any other request, we need a way to store user data between requests. **Cookies, Query string, hidden fields** (HTML tags not visible), and **sessions** can be used to store a unique identifier. The server then uses that identifier to retrieve the appropriate session information. This is not done automatically. Web applications (backend programs) need to maintain and manage the user sessions by implementing states using HTTP cookies, server-side sessions, and hidden variables.

### Stateful connection

The connection **between the web server and the database server is stateful**. Stateful connection is not scalable as it maintains the connection until it completes a task for each request. **To deal with the scalability issue**, a connection pool (either client-side or server-side) can be used to serve many data requests. With the connection pool, any available connection from the pool can be assigned to a database request without going through the opening and closing connections.

### Mobile application

A mobile application (or mobile app) is an application software designed to run on a mobile device (e.g., smartphone or tablet) that is a small computer.

## Can I fetch data from a website or file using JavaScript?

Yes, you can use **JavaScript fetch API** to fetch data. The following JavaScript code fetches data from a website, a JSON data placeholder available for the developers to test their APIs:

```
//get_users.js
const users = [
  {name: 'Bob', age: 20},
  {name: 'Joe', age: 30}
];

function getUsers() {
  setTimeout(() => {
    let output = '';
    users.forEach((user, index) => {
      output += `<li>${user.name}</li>`; //this is a backquote
      //console.log("idx: " + index)
    });
    document.getElementById("demo").innerHTML = output;
  }, 1000);
}

function createUsers(user) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      users.push(user);
      const error = false;
    });
  });
}
```

```

        if (!error) {
            resolve();
        } else {
            reject('Error occurred');
        }
    }, 2000);
})
}

async function init() {
    await createUsers({name: 'Jane', age: 25});
    getUsers();
}

async function fetchUsers() {
    const res = await fetch('https://jsonplaceholder.typicode.com/users');
    const data = await res.json();
    console.log(data);
}

init();
fetchUsers();

```

```

<html>
  <head></head>
  <body>
    <h1>A user list</h1>
    <p id="demo"></p>
    <script src="get_users.js"></script>
  </body>
</html>

```

In the above program, **async/await** is used to work with promises in asynchronous functions. Therefore, **async/await** is just a **wrapper** to restyle code and make Promise **easier to read and use**. **Await** only blocks the code execution within the **async** function for a Promise. It only makes sure that the next line is executed when the “Promise” resolves. So, if an asynchronous activity has already started, the “await” will not have any effect on it.

In the above example code, you used **JavaScript fetch API** to fetch JSON data provided by the JSON placeholder “https://jsonplaceholder.typicode.com/users”. Since **2015**, the JavaScript “fetch API” allows a web browser to make HTTP requests to web servers. You can easily create your own APIs and expose the interface for other applications. Now, you can imagine there will be many different APIs.

## What is API?

API stands for **Application Programming Interface**. API is a type of **software interface** offering a **service** to other applications. An API is a way for two programs to communicate with each other (similar to the user interface). Since API is just an interface, a document or standard that describes how to build and use such a connection or interface, such as input and output, is important. That document or standard is called API

specification. A computer system that meets this standard is said to implement or expose an API. An API call is typically made to request a service from the server or any application that provides the service.

### Reading data from a local file using JavaScript

For another example of fetch API, create a text file called “mydata.txt” with any text and save it in your local directory, try the following JavaScript code to read the text data from ‘mydata.txt’ using “**getText**” method:

```
<html>
  <head></head>
  <body>
    <h1>My Data</h1>
    <p id="demo"></p>
    <script>
      getText("mydata.txt");
      async function getText(file) {
        let x = await fetch(file);
        let y = await x.text();
        document.getElementById("demo").innerHTML = y;
      }
    </script>
  </body>
</html>
```

When you run it, you will notice that this code **will not work**. It is because of the **Cross-Origin Source Sharing** policy for security.

### What is Cross-Origin Source Sharing policy?

**Cross-Origin Source Sharing (CORS)** is a mechanism that allows restricted resources on a Web page to be requested from another domain outside the domain from which the first resource was served. This is called **cross-origin**. Here **Origin** is a portion of the URL consisting of protocol, host, and port (<http://www.xyz.com:8080/>). A web page may freely embed cross-origin images, stylesheets, scripts, iframes, images, and fonts. For security reasons, **browsers restrict cross-origin HTTP requests initiated from scripts**. Certain **cross-domain requests**, notably Ajax requests, **are forbidden by default** by the same-origin security policy. A web application can only request resources from the **same origin** the application was loaded from unless the response from other origins includes the right CORS headers. In order to keep a website and its users secure from the security risks involved with sharing resources across multiple domains, the use of CORS is recommended. CORS policy defines a way in which a browser and server can interact to determine whether it is safe to allow the cross-origin request. With the use of CORS, the browser and server can communicate to determine whether it is safe to allow a cross-origin request. For more information about CORS, refer to [this page](#).

### How to resolve the CORS-related issues

- (a) When you have control of the backend, use a **whitelist** for the “Access-Control-Allow-Origin” header in the backend program. For a backend program with Node.js, install the “cors” package within a Node application, add “const cors = require(‘cors’), and use it by “app.use(cors({origin: ‘https://www.other\_site.com’})); or origin: ‘\*’)”. CORS goes hand in hand with APIs. A good use case scenario of CORS is when developing RESTful APIs. We will learn these topics later.

- (b) When you do not have control of the backend at all, write **your own proxy** that will sit between the browser application and the API. The proxy does not have to be running on the same domain as your application, as long as the proxy itself properly supports CORS when communicating with the client. The proxy server is a mediator or a replacement server that allows your computer to connect to the Internet using a different IP address, concealing the user's real address from web servers. It adds an extra layer of privacy and can also save a lot of bandwidth in certain situations.

- (c) When you need a **temporary solution only for your development and testing**, you can make your browser ignore the CORS mechanism.

For example, **to test this approach in Windows**, (a) close all Chrome browsers, (b) Windows key + R (c) Type "chrome.exe --user-data-dir='C://Chrome dev session' --disable-web-security" and enter (d) open the above html file with cross-origin request", and see if the data from 'mydata.txt' is displayed. It works, but **we will not use this approach again** as it is not recommended practice. Instead, we will discuss approaches (a) and (b) later.

### Here is another example related CORS:

```
<!DOCTYPE html>
<html lang="en">
<head><title>Progress Bar </title>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.1/css/bootstrap.min.css"
crossorigin="anonymous">
  <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.6.0/jquery.min.js"
crossorigin="anonymous"></script>
  <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.1/js/bootstrap.min.js"
crossorigin="anonymous"></script>
</head>
<body>
<div class="container">
  <h2>Basic Progress Bar</h2>
  <div class="progress">
    <div class="progress-bar" role="progressbar" aria-valuenow="70" aria-valuemin="0" aria-valuemax="100"
style="width:70%">70% Complete</div>
  </div>
</div>
</body>
</html>
```

The **HTML <script> tag attribute 'crossorigin'** sets the mode of the request to an HTTP CORS request for a resource, such as stylesheets, iframes, images, fonts, or scripts from another domain. is used for loading an external script into their domain from a third-party server or another domain with the support of HTTP CORS Request. The attribute value can be 'anonymous' or 'use-credentials', **<script crossorigin="anonymous"></script>**. 'anonymous' means a cross-origin request is performed without credentials being sent. 'use-credentials' means credentials are sent (e.g., a cookie, a certificate, or HTTP basic authentication), e.g., **<script crossorigin="anonymous | use-credentials"></script>**. crossorigin attribute specifies that CORS is supported when loading an external script file from a third-party server or domain. If you omit "crossorigin", CORS behavior is disabled. Also, look at the **integrity** attribute for the hash value of a

resource like a checksum that has to be matched to make the browser execute it to ensure that the file was not modified and contains expected data to avoid loading different or malicious resources.

## What is bootstrap.min.js?

It is a **minified version of Bootstrap's** JavaScript. **Bootstrap** is a free **front-end CSS framework** originally developed at Twitter and designed for creating a responsive and mobile-first website (automatically adjusts itself to look good on all devices, from smartphones to desktops). Review Bootstrap at: <https://www.w3schools.com/bootstrap/default.asp>.

### Learn bootstrap from examples

- [Some examples here](#)
- [More examples](#)

### Exercise

Create menu bars for your website using Bootstrap.

## What is jQuery?

jQuery is a JavaScript library that simplifies JavaScript programming. It was designed to handle browser incompatibilities and to simplify HTML DOM manipulation, event handling, and Ajax.

There are several ways to start using jQuery on your website:

- Download the jQuery from jQuery.com.
- Include jQuery from a Content Delivery Network (CDN), e.g., from Google.

### Downloading jQuery

```
<head>
  <script src='jquery-3.6.0.min.js'></script>
</head>
```

### jQuery CDN

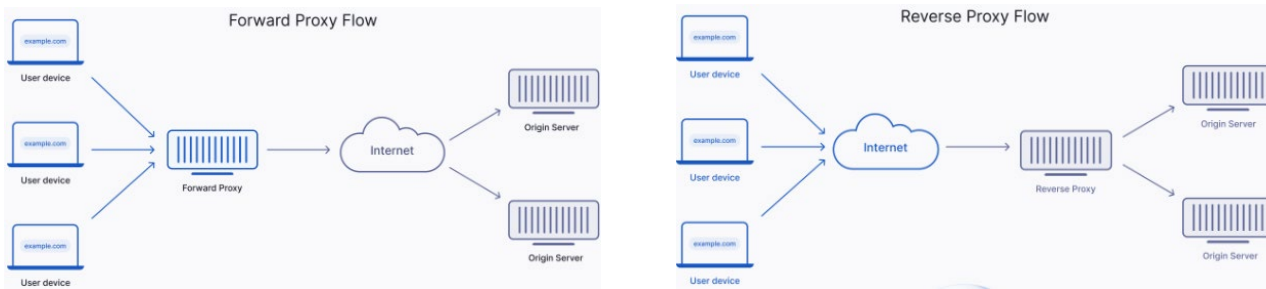
```
<head>
  <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.6.0/jquery.min.js"></script>
</head>
```

Basic syntax is **\$(selector).action()** where (selector) to query or find HTML elements and action() to perform on the elements. Review some examples at: <https://www.w3schools.com/jquery/default.asp>.

## How does a proxy server work?

There are many different types of proxy servers, categorized by traffic flow, anonymity level, application, service, IPs, and accessibility. A **forward proxy server** is commonly known as a '**proxy server**'. It is a type of proxy server that typically passes requests from users in an internal network to the Internet via a firewall (*proxies on behalf of clients*). Forward proxies are configured to either 'allow' or 'deny' the user's request to pass through the firewall to access content on the Internet. If the proxy allows the user's request, it forwards it to the web server through the firewall. The web server sends its response to the proxy. The proxy then sends this response back to the user. A forward proxy will first check if the user's requested information is **cached** before retrieving it from the server. If the requested information is cached, the proxy will send it directly to the user.

The proxy stores cached information itself, eliminating the need to request it from the server. If the proxy denies the user's request, it sends the user an error or redirects a message.



A **reverse proxy server** is a proxy server that typically passes requests from the Internet through to users in an internal network via a firewall; essentially, a forward proxy in **'reverse'** (*proxies on behalf of servers*). Reverse proxies are configured to restrict and monitor users' access to web servers containing sensitive data. User requests are passed through the Internet via a firewall to the reverse proxy. A reverse proxy will first check if the user's requested information is cached before retrieving it from the server.

## What is AJAX, and how does it work?

Ajax (Asynchronous JavaScript and XML) is a JavaScript library or framework used in writing client-side programming to allow for data to be sent and received to and from a database or server. AJAX uses a combination of a browser built-in XMLHttpRequest object (to request data from a web server) and JavaScript and HTML DOM (to display or use the data). Ajax is not a language. All modern browsers support the XMLHttpRequest object. Try the following example:

```
<!DOCTYPE html>
<html>
<body>
  <h2>The XMLHttpRequest Object</h2>
  <div id="demo">
    <p>Let AJAX change this text.</p>
    <button type="button" onclick="loadDoc()">Click this to change Content</button>
  </div>
  <script>
    function loadDoc() {
      const xhttp = new XMLHttpRequest(); //creating an XMLHttpRequest object
      xhttp.onload = function() {
        document.getElementById("demo").innerHTML = this.responseText; //server Response properties
      }
      xhttp.open("GET", "mydata.txt"); //get data from the file
      xhttp.send(); //send a Request to a server
    }
  </script>
</body>
</html>
```

## Send a Request to a server

- open(method, URL, async) specifies the type of request either GET or POST, async (asynchronous is true) or false, e.g., xhttp.open('GET', 'test.txt', true)
- send() sends the request to the server (used for GET) and send(string) used for POST, e.g., xhttp.send()

### Server response properties

- responseText gets the response data as a string.
- responseXML gets the response data as XML data.

### Server response methods

- getResponseHeader() returns specific header information from the server resource.
- getAllResponseHeader() returns all the header information from the server resource.

**Note:** *After **JavaScript version 5 (2009)**, most programming problems using the jQuery utilities and AJAX can be solved with the standard JavaScript. AJAX and jQuery are being replaced by JavaScript or JavaScript frameworks such as React or Node/Express.js.*

### Learn complex use cases of HTML and JavaScript from example projects

- [Some HTML examples here](#)
- [More HTML examples](#)
- [JavaScript projects](#)



## 1.2. Continuous Integration and Continuous Deployment (CI/CD)



**Continuous Integration (CI)** is the practice of merging all the developers' "**working code**" into a single shared repository to build and test for automating the integration of code changes from multiple developers. **Continuous Deployment (CD)** is the practice of automatically passing the integrated software features that passed the automated testing phase into the production environment, making changes that are visible to the users. CI/CD is one of the important practices for the **DevOps process**.

### Study guide

- Create a local repository and a central repository, and connect them together.
- Practice the process of committing and publishing the committed code to the central repository.
- Practice code refactoring and continuous integration.
- Practice continuous deployment through automatic deployment to real production environments, e.g., AWS, Heroku, etc.
- Understand the entire pipeline of the CI and CD process.

### How can I save my project files to continue working on my project anywhere or/and collaborating with my team members?

#### Creating a project repository using Git and GitHub and connecting them with Visual Studio Code

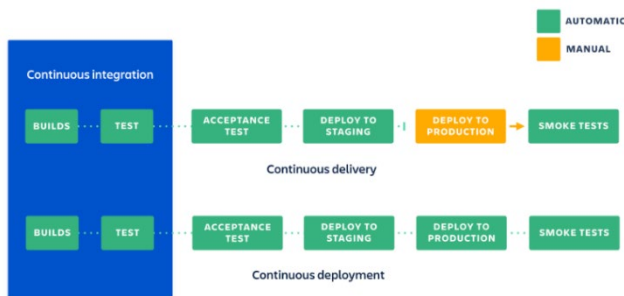
- Create a **project folder** for your workspace, e.g., 'myapp', and copy the project files, e.g., 'hello.js' to this directory.
- Download and install Git. **Git is a version control system** that allows you to manage and keep track of your source codes.
- Enable Git in VS Code: File->Preferences->Settings, type 'Git: Enabled' in the search box, and make sure that the box for 'Git: Enabled' is checked.
- **Create a Git repository** (Git repo) for your project from  'Source Control' on the left menu bar of VS Code (or View->Source Control) by clicking 'Initialize Repository'. You can also initialize git by **Git commands** 'git init' at the terminal in your project directory.
- **Verify** if the Git repository (.git folder) was created in your project folder, e.g., 'myapp'. This repository tracks all changes made to files in your project. So if you delete the .git/ folder, you delete your project's history.
- **Create an account at GitHub.com and create a repository for your project**. GitHub is a cloud-based hosting service that lets you manage Git repositories in a centralized location.
- **Connect** your local Git repository on your computer with GitHub in VS Code: View->Command Palette->Git: Add remote..., enter the remote repository URL (from GitHub, '<>Code' -> Code (green color) -> copy the URL). Or create a repository on GitHub first and **clone** it in your local directory.
- **To verify the connection, create a simple HTML file**, e.g., 'hello.html', click  for Source Control, click + to stage changes, enter a commit message, click ✓ to commit, click '**Sync Changes**' (or **Publish Branch** if you have not published this branch on GitHub before). If connected properly, now you can push the source codes of your project in a local repository to the GitHub repository, allowing you to access it and collaborate with your team members anywhere.
- **Open the repository page on your GitHub account to verify** if the file was pushed correctly and click <> Code. If you find the file ('hello.html') in the GitHub repository, everything works fine. For a detailed demo, [watch this tutorial](#).

- **Make sure you add .gitignore file** to let **Git ignore** tracking the changes of certain files or folders. You can use a .gitignore extension or manually create it. For an **auto-generated .gitignore** file (View->Command Palette (Ctrl+Shift+P), type 'Add gitignore', choose JavaScript, add VS code extension, and edit it to add or remove ignore items as needed (e.g., add a line '.vscode/'). You can also **right-click a file** (in Source Control) to add it to .gitignore list.

Creating a project folder, a local project repository, and a repository on GitHub and connecting all these repositories and various servers related to the project (e.g., **development server**, **staging server**, and **production server**) is **one of the important steps for CI/CD practice**.

## What is Continuous Integration, Continuous Delivery, Continuous Deployment, or Delivery? Why are they important?

**Continuous Integration** (CI) is the practice of integrating any new module developed and testing each change done to your codebase automatically and as early as possible. **Continuous Delivery** (CD) is an extension of continuous integration since it automatically deploys all code changes to a testing and/or product environment after the build stage. With the automated release process, you can deploy your application at any time by clicking a button. **Continuous Deployment** (CD) goes one step further than continuous delivery. With this practice, every change that passes all stages of your production pipeline is released to your customers without human intervention.



CI/CD practice is to accelerate the feedback loop with your customers and take pressure off the team. CI/CD is a frequently used industry best practice and one of the fundamental **DevOps best practices**. Many **toolchains** are available to support this practice since process automation is the key.

### Advantages of CI/CD are:

- Increased productivity and quality as developers can focus on building software (increased productivity) and receive fast feedback from customers through a tighter feedback loop and collaboration that allows faster bug fixes (increases the quality).
- Faster time to market
- Reduced risk

## How can I deploy my application based on continuous deployment (CD) practice?

In order to deploy an application, we need a server. In this example, we will use **Heroku** and **AWS** cloud for our servers.

### Deploying an application to the Heroku cloud from the GitHub repository

1. Make sure you have a **repository created on GitHub** and all the program files pushed to this repository and ready to deploy to the production server.
2. In this example, we will deploy a simple HTML page "hello.html" and **PHP** file 'index.php' with only one line of code `<?php include_once("hello.html");?>` that includes 'hello.html' file. The server we will use is

**Heroku cloud.** We use PHP for **deployment testing purpose** as Heroku does not host a static website with only HTML files. **We will not use PHP in this lesson.**

3. **Push** these two files to the GitHub repository.
4. **Open an account at Heroku for a free plan.** Heroku is a container-based cloud Platform as a Service (PaaS). The free plan offers many Heroku services for free.
5. **Create a new application at Heroku**, e.g., ‘**myapp-heroku**’ (Heroku requires an available name). You can add a **pipeline** (that lets you connect multiple apps), but we will skip it now. You can add a **buildpack** at Heroku settings if necessary. Skip this step. We will try this later.
6. **Choose ‘GitHub’ as a deployment method** among Heroku Git, GitHub, and Container Registry. If you choose Heroku Git, you use Heroku CLI. For this, install the Heroku CLI, log in to your Heroku account by ‘heroku login’, and connect your local repository to the Heroku repository by ‘heroku git:remote -a myapp-heroku’, push the code to the Heroku repository to deploy it to Heroku.
7. Choose either Automatic deploys or Manual deploy. We will choose “**Manual deploy**” until we set up a Continuous Integration service configured.
8. **Connect Heroku to Github** by entering the GitHub repository name, ‘myreact-heroku’.
9. **Open the application** ‘Open app’ or ‘View’ or open a page ‘<https://myapp-heroku.herokuapp.com/>’ using your browser. If everything went well, you should be able to see the ‘hello.html’ page loaded by ‘index.php’.

You can also **deploy from the Heroku repository** by choosing ‘**Heroku Git**’ instead of GitHub as the deployment method, following the steps below:

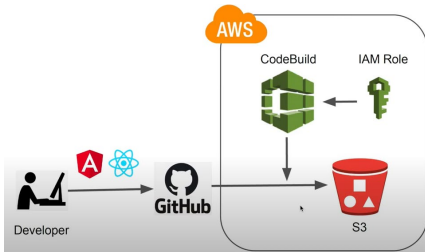
- Install Heroku CLI and verify it by ‘heroku -v’
- Log in to Heroku using Heroku CLI by ‘heroku login’
- Connect your local repository to Heroku Repository by ‘heroku git:remote -a myapp-heroku’.
- Commit and push your code to the Heroku repository.
- Open your application from <https://myapp-heroku.herokuapp.com/>.

### Deploying a simple web application to AWS using Amazon S3

1. **Create an account at AWS.** You may want to try the [AWS free tier](#) that allows you to explore and try out AWS services free of charge up to specified limits for each service.
2. **Create a bucket in S3:** enter a bucket name, e.g., ‘**myapp-bucket-aws**’, -> uncheck ‘Block all public access, click ‘Create bucket’ button. If everything goes well, you should be able to see a list of buckets created on the Buckets page. A **bucket** is a container for objects stored in [Amazon S3](#) (Simple Storage Service) offered by **Amazon Web Services (AWS)** that provides object storage through a web service interface.
3. **Find the bucket** you just created and click it to upload files.
4. **Upload an HTML file**, e.g., ‘hello.html’ (In AWS, we don’t need to use ‘index.php’) and close the upload page to go back to the **bucket object page** ‘myapp-bucket-aws’.
5. **Allow the site for public access:** Click ‘Properties’ -> Edit ‘**Static website hosting**’ at the bottom of the page (for ‘hello.html’) -> Enable ‘Static website hosting’ -> Type ‘hello.html’ to specify the default page of the website (index.html or home.html are typical default page) -> Click ‘Save changes’ (or click ‘Permissions’ -> Edit and uncheck ‘Block Public Access’ -> check the box to accept the acknowledgment).
6. **Create a Bucket policy for the site:** Click ‘Permission’ on the bucket object page -> Edit ‘Bucket policy’. If you have not generated any policy, it will ask you to generate a policy, or it will show you a previously generated policy. **Bucket Policy** is a resource-based policy that you can use to grant access permissions to your bucket and the objects in it. Read [more information about the bucket policy](#). **Copy**

**Bucket ARN** (Amazon Resource Name) for your application to generate a bucket policy. ARN uniquely identifies AWS resources.

7. **Generate a bucket policy statement:** Click 'Generate policy', select 'S3 Bucket Policy' for the type of policy, 'Allow' for Effect, '\*' for Principal, 'GetObject' for Actions, copy and paste ARN (add /\* to allow all files), click 'Add Statement', click 'Generate Policy' (that will show a policy statement in JSON format), **copy the bucket policy statement**, click 'Close, and paste it to Bucket policy editor, and choose 'Save changes.' It should show a success message.
8. **Test your website endpoint:** Buckets -> choose your bucket -> Properties -> At the bottom of the page, under 'Static website hosting', choose your Bucket website endpoint or click the URL to open your site.
9. For more information, refer to [this tutorial page](#). You will see the default page of your site.



### Deploying a simple web application to AWS using Amazon S3 from GitHub for Continuous Deployment

1. Create a new bucket at AWS, e.g., 'myapp-bucket-aws-github'.
2. Upload an HTML file to the bucket.
3. Allow public access to the bucket.
4. Create a bucket policy for the bucket. Now you should be able to manually create a policy statement by directly typing it in the policy editor.
5. Open the page of the IAM dashboard (by searching it by keyword) to set up AWS IAM (Identity and Access Management). [AWS IAM](#) is a web service that helps you securely control access to AWS resources. You use IAM to control who is authenticated (signed in) and authorized (has permission) to use resources.
6. **Add an IAM user** by following the [steps here](#) (assuming you didn't create a user): User name (e.g., myapp-aws) -> check 'Programmatic access for AWS credential type and click Next: Permissions -> click Attach existing policies directly -> type s3full for Filter policies -> select AmazonS3FullAccess -> click Next:Tags -> click Next: Review (ignoring Tags in this example) -> click Create user (You should see a success message if this process went well) -> (optionally download .csv that contains a user name, access key ID, secret access key, console login link that will be used later) -> In the bottom of the page click 'Close'. You should be able to see a user created.
7. **Verify if ACLs are enabled:** Open your AWS bucket page for this application -> Permissions -> Edit on Object ownership -> select 'ACLs enabled' for the bucket and check the acknowledge statement -> Save changes.
8. **Open the repository on GitHub to deploy.**
9. **Add credentials for remote server** (AWS Access ID and key for a bucket): Click **Settings** -> **Secrets** (on the left menu bar) -> Actions -> New repository secret -> type YOUR\_SECRET\_NAME and value pair by 'AWS\_ACCESS\_KEY\_ID' and ID value, 'AWS\_SECRET\_ACCESS\_KEY' and key value (from the IAM user).

10. **Create a workflow at GitHub for automatic deployment:** In VS code, create a folder `‘.github/workflows’` and add a YAML file, e.g., `‘main.yml’`, and type the following text, save it, commit it, and push it to GitHub:

```
name: Upload Website
on:
  push:
    branches:
      - master
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@master
      - uses: jakejarvis/s3-sync-action@master
        with:
          args: --acl public-read --follow-symlinks --delete
    env:
      AWS_S3_BUCKET: ${ secrets.AWS_S3_BUCKET }
      AWS_ACCESS_KEY_ID: ${ secrets.AWS_ACCESS_KEY_ID }
      AWS_SECRET_ACCESS_KEY: ${ secrets.AWS_SECRET_ACCESS_KEY }
```

[YAML](#) is a data serialization language that is often used for writing configuration files. This YML file is based on the template from [‘S3 Sync’](#) available on the [GitHub marketplace](#). You can search this template by typing ‘sync s3’ on the GitHub Marketplace search box.

11. **Verify the deployment:** Click ‘Actions’ and see automatic deployment done correctly (green color). If failed, click it to see the error messages and correct it.
12. **Test it:** Modify the HTML file and open it from the link provided by AWS. If you see the modified page, you have now successfully set up the Continuous Deployment pipeline with GitHub for your application. If it doesn’t open the page you expected, check the Website endpoints, especially the region (the default region is ‘us-east-1’) or default page name (‘hello.html’ in this example).

### Additional methods to deploy to AWS from GitHub

- AWS also provides a tool [AWS CodePipeline](#) for a continuous delivery service that enables you to model, visualize, and automate the steps required to release your software.
- [AWS Amplify](#) is a tool provided by AWS that helps mobile and web developers build and deploy full-stack applications on AWS. The tool includes authentication, data store, API, analytics, and push notification.
- Infrastructure as a code (IAC) is a process of **managing** and **provisioning** computing resources and infrastructure through machine-readable definition files rather than physical hardware configuration or interactive configuration. **AWS CloudFormation** is an IAC tool developed for AWS infrastructure.

## 1.3. Introduction to Node.js

**Node.js** is an open-source and cross-platform **runtime environment** for server-side and network applications, originally developed by Ryan Dahl in 2009 for fast and scalable applications. It uses **Chrome's JavaScript engine**, V8. Node runs on top of the V8 engine that **compiles the JavaScript code** in the **native machine code** and executes it. It uses various components.

Many backend web application frameworks such as Express, Next, etc. are developed based on Node by adding commonly used features for web developers to improve their coding productivity.

### Study guide

- Understand event, event emitter, event loop, and callback concepts and mechanisms for event-driven programming.
- Understand the concepts of asynchronous event, buffering, non-buffering, and event loop workflow.
- Importing and exporting modules

### Tutorial sites

- [Nodejs.org](https://nodejs.org) (official website for Node)
- [Nodejs.dev](https://nodejs.dev)
- <https://www.tutorialsteacher.com/nodejs>

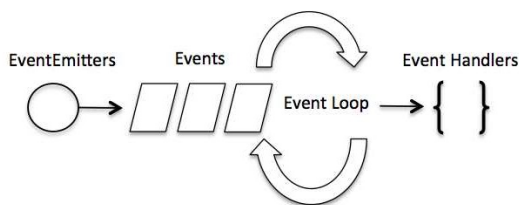
In previous lessons, you've already installed Node. However, **to verify Node installation**, create the following JavaScript, and execute it by typing 'node hello.js':

```
//hello.js
console.log('Hello, Node');
```

### Primary features of Node

- Asynchronous and event-driven
- No buffering
- Single-threaded but with event looping, asynchronous, and non-blocking
- Very fast, highly scalable, and cross-platform that runs across distributed services

### Event-driven (dealing with event emitters, events, and event handlers)



### Blocking vs. non-blocking

Blocking refers to an operation that blocks further execution until that operation finishes, while non-blocking refers to an operation that does not block execution. In Node, all I/O operations (interactions with file systems, databases, networks) supported by **libuv** and **libeio** libraries are asynchronous and non-blocking, accept callback functions, wait for I/O operations to complete, and are delegated after completion. These I/O



operations typically require stateful connections and take time to complete. With the non-blocking, Node can free up the time needed to process per request to handle many other requests.

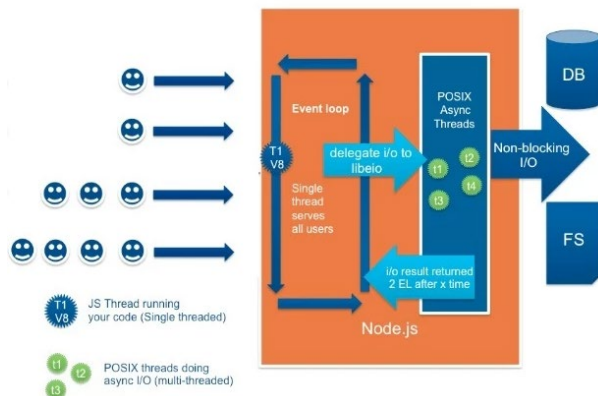
**Non-blocking code example:** create a text file 'input.txt' before you run this program.

```
//nonblocking.js
var fs = require("fs");
fs.readFile('input.txt', function (err, data) {
  if (err) return console.error(err);
  console.log(data.toString());
});
console.log("This last statement executed first before the file reading done.");
```

This program will print 'This last statement executed first before the file reading done.' from the last statement before the message from file I/O.

### Node.js single thread event loop workflow

Node uses **single-thread** event loop architecture (as shown in the diagram below) to process request-response requests from clients, utilizing lesser computing resources compared to most other web application frameworks that rely on **multi-threaded** request-response architecture to handle multiple concurrent clients.



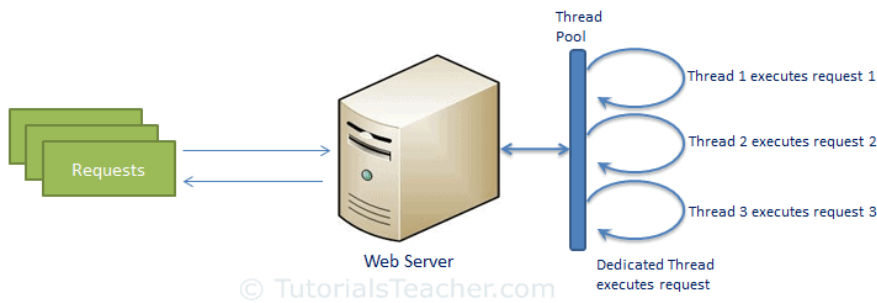
### **An example Node application that creates a simple web server**

```
//myserver.js
var http = require("http");
http.createServer(function (request, response) {
  // Send the HTTP header
  // HTTP Status: 200 : OK
  response.writeHead(200, {'Content-Type': 'text/plain'});
  // Send the response body as "Hello Node!"
  response.end('Hello world!\n');
}).listen(8081);
// Console will print the message
console.log('My Web server running at http://127.0.0.1:8081/');
```

You can **create a small web server** within your Node project to handle all browser requests and responses.



In this case, **do we still need a web server** like Nginx or Apache with Node? The answer is yes because of other necessary services provided by a Web server such as HTTP load balancing, reverse proxy, etc.

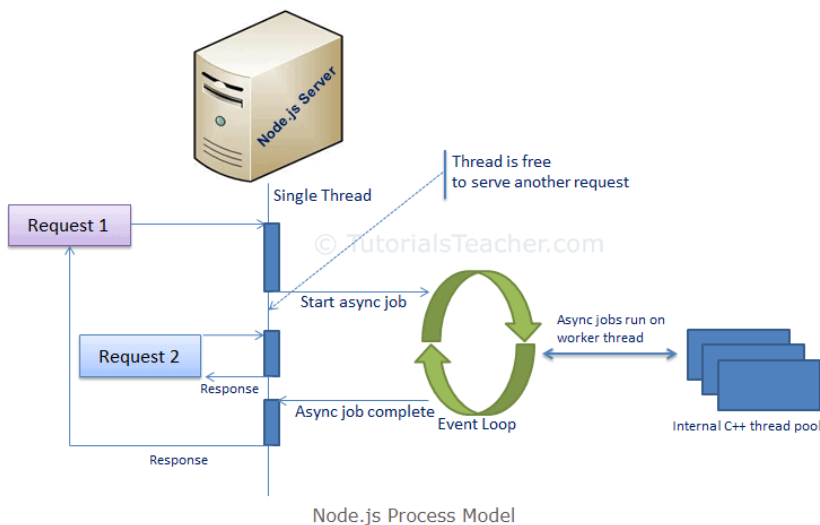


Traditional Web Server Model

### Comparing traditional web server model vs. Node web server

**In the traditional web server model**, each request is handled by a dedicated thread from the thread pool. The dedicated thread executes a request and does not return to the thread pool until it completes the execution. If no thread is available in the pool, the request waits until the next available thread.

**In the Node web server model**, Node runs in a single process, and the application code runs in a single thread (fewer resources needed). All the user requests will be handled by a single thread,



Node.js Process Model

but all the I/O work or long-running job for each request is performed asynchronously. So, this single thread does not have to wait for the request to complete and is free to handle the next request. When asynchronous work completes, it processes the request further and sends the response, completing the request.

An **event loop** is constantly watching for the events to be raised for an asynchronous job and executing a callback function when the job completes. Internally, Node uses “libev”

for the event loop, which in turn uses an internal C++ thread pool to provide asynchronous jobs.

### Application areas of Node

- I/O bound and network applications
- Data-intensive real-time applications
- JSON APIs-based applications
- Single Page Applications

### The main problem of Node

- Not good for CPU-intensive operations or heavy computation (e.g., image processing) because it takes time to process a request and thereby blocks the single thread. In this case, a big tasks can be broken down into smaller tasks or “worker” thread can be used.

### Node syntax

Node is based on JavaScript. So, it uses the JavaScript syntax plus a few Node-specific types such as the “buffer” type to store binary data and “process” objects.

### Quick testing Node program using Node CLI

Type “**node**” at the command prompt and write Node programs. To exit from CLI, ‘**.exit**’.

### Node modules

Each module has its own context (scoping) that can be placed in a separate “.js” file. The node includes three types of modules: core (HTTP, URL, query-string, path, fs, util), local (modules created in your application), and third-party modules (developed by third-party).

### Creating a local module, e.g., “persons.js”:

```
//Person.js
var Person = {
  name: function () {console.log("Bob");},
  age: function () {console.log(30);}
}
module.exports = Person;
```

### Exporting modules using module.exports or exports

module.exports = {value, function}

exports.value = value

exports.function = function

For example you can export a function as follow:

```
exports.handler = async(event) => {
  var val = 100;
  return val;
}
```

The “**module**” is a variable that represents the current module, and “**exports**” is an object that will be exported as a module. Since “exports” is an object, you can attach properties and methods to it.

### An example of exporting modules

```
module.exports = Person;
module.exports = "Hello, world";
exports.Hello = "Hello world";
```

### Importing modules: var module = require(‘module\_name’);

```
var person = require("./Person");
var hello_msg = require("./Hello.js");
person.name();
hello_msg.Hello;
```

To import part of an object, you use {}, e.g., {dog} from “animals.js”.

## Available commands and options for Node package manager

npm help

### Installing packages or modules

- npm install <package\_name> to install package locally
- -g option to install package globally
- --save option to add dependency into package.json file.
- npm update <package\_name>
- npm uninstall <package\_name>

### Creating Node Web server

```
var http = require('http');
var server = http.createServer(function (req, res) {
    //code to handle incoming requests here
});
server.listen(<port>);
```

### An example Node code to handle HTTP requests and responses

```
//server.js
var http = require('http'); // Import Node.js core module
var server = http.createServer(function (req, res) { //create web server
    if (req.url == '/') { //check the URL of the current request
        // set response header
        res.writeHead(200, { 'Content-Type': 'text/html' });

        // set response content
        res.write('<html><body><p>This is home Page.</p></body></html>');
        res.end();
    }
    else if (req.url == "/student") {
        res.writeHead(200, { 'Content-Type': 'text/html' });
        res.write('<html><body><p>This is student Page.</p></body></html>');
        res.end();
    }
    else if (req.url == "/admin") {
        res.writeHead(200, { 'Content-Type': 'text/html' });
        res.write('<html><body><p>This is admin Page.</p></body></html>');
        res.end();
    }
    else
        res.end('Invalid Request!');
});
server.listen(5000); //6 - listen for any incoming requests
console.log('Node.js web server at port 5000 is running..')
```

To run it, type “node server.js” or “nodemon server.js”.

### Node file system

To read a file, fs.readFile(filename, [,options], callback)

```
var fs = require('fs');
fs.readFile('test.txt', function(err, data) {
  if (err) throw err;
  console.log(data);
});
```

To read a file synchronously, you use fs.readFileSync() method. To write to a file, you call fs.writeFile(filename, data[, options], callback).

See more information about the Node file system at the [official Node website](#).

### Event emitter

Node allows us to create and handle custom events using the events module. The event module includes EventEmitter class which can be used to raise and handle custom events. The following example demonstrates EventEmitter:

```
// get the reference of EventEmitter class of events module
var events = require('events');

//create an object of EventEmitter class by using above reference
var em = new events.EventEmitter();

//Subscribe FirstEvent
em.on('FirstEvent', function (data) {
  console.log('First subscriber: ' + data);
});

//Subscribe SecondEvent
em.addListener('SecondEvent', function (data) {
  console.log('Second subscriber: ' + data);
});

// Raising FirstEvent
em.emit('FirstEvent', 'This is my first Node.js event emitter example.');
```

The **on()** method or **addListener()** method is to subscribe to an event, and the **emit()** function raises the specified event.

Refer to the official [Node website](#) for additional information about **EventEmitter methods**.