**Steps followed to clone Air ALgerie web site using react js for the front end and tailwind for css:**

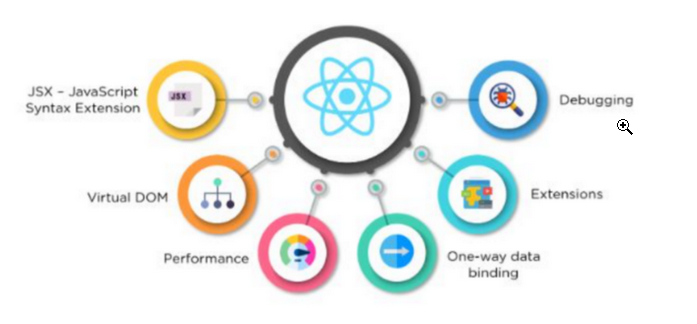
M**ERN** is one of several variations of the [MEAN stack](https://www.mongodb.com/mean-stack) (MongoDB Express Angular Node), where the traditional Angular.js frontend framework is replaced with React.js. Other variants include MEVN (MongoDB, Express, Vue, Node), and really any frontend JavaScript framework can work.

1. **Front End:**

* **React JS:** A JavaScript library for building user interfaces

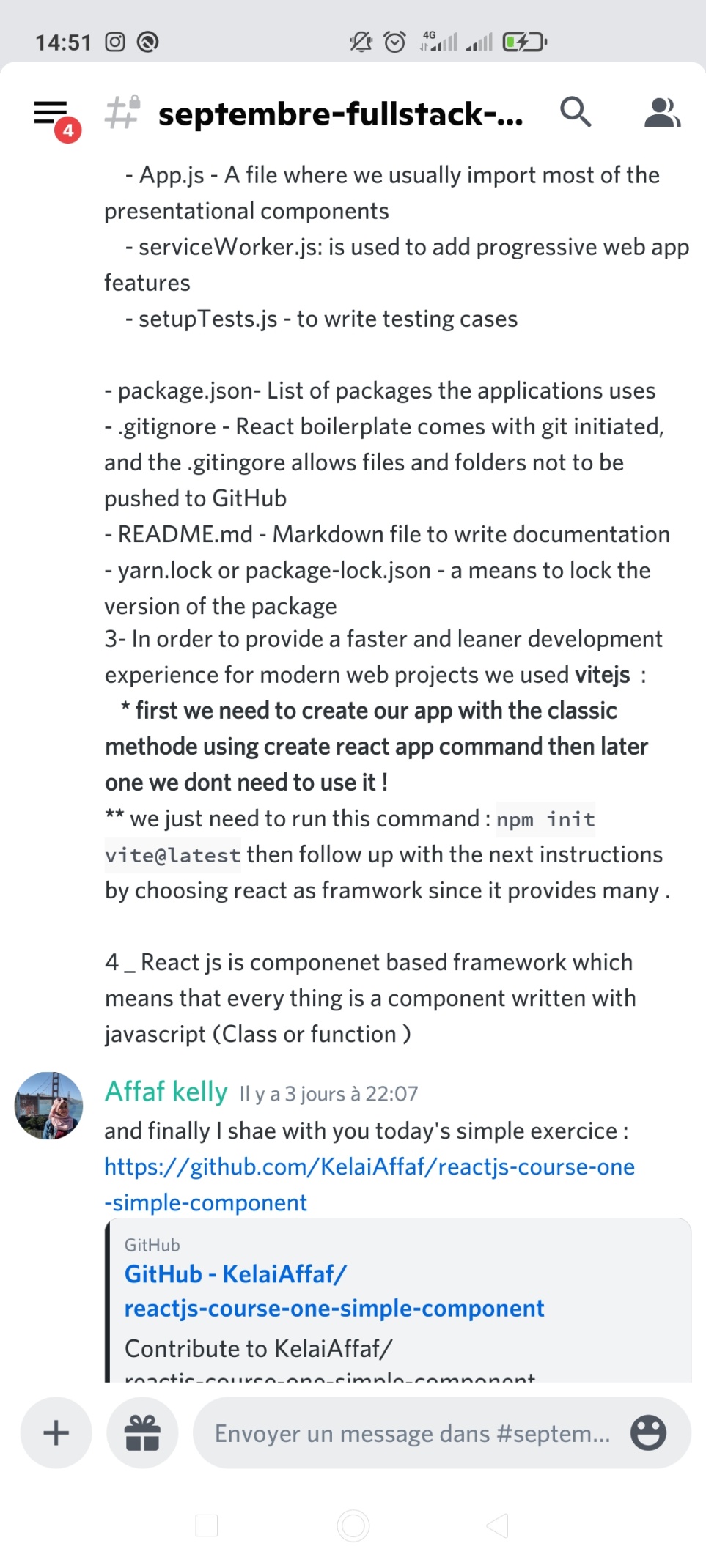
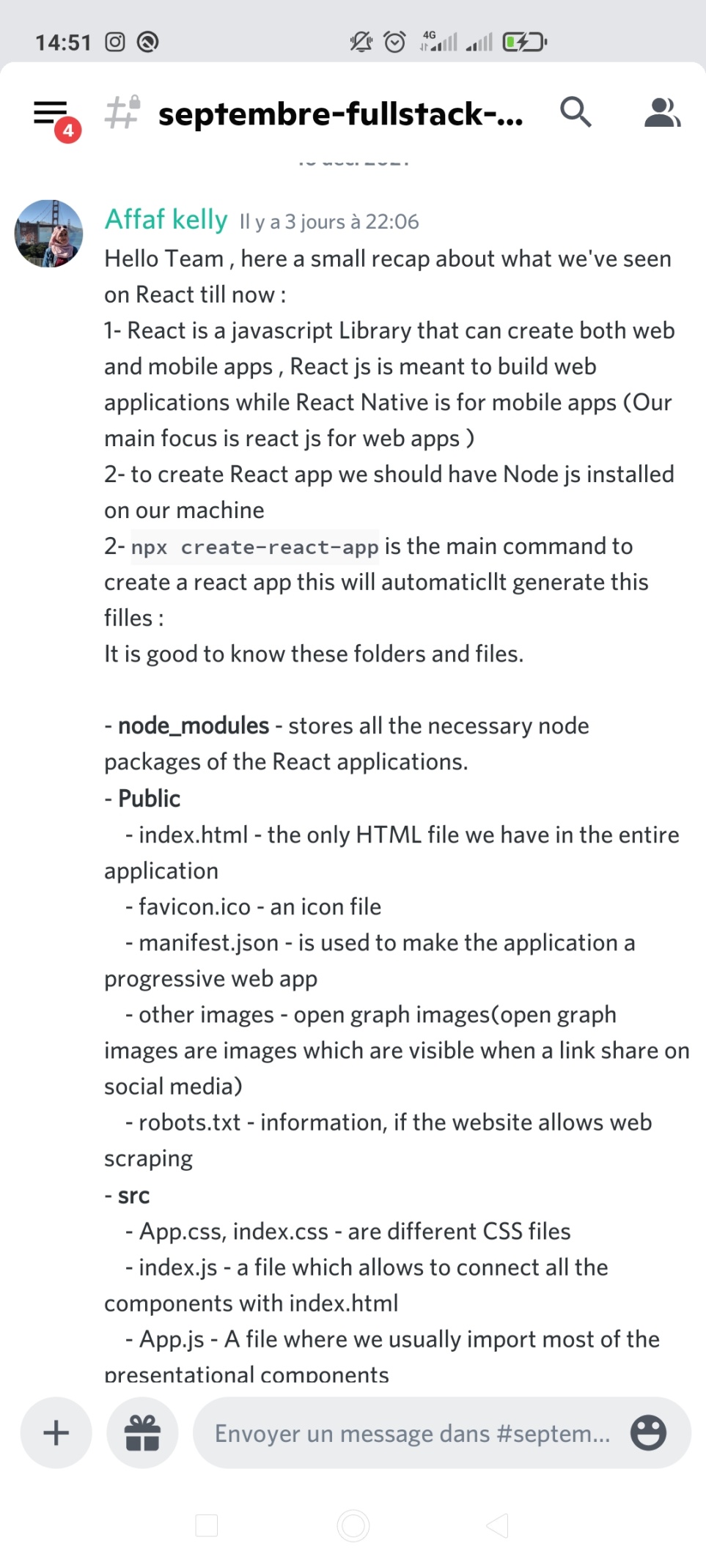
The top tier of the MERN stack is React.js, this library for Client-Side Applications in HTML is completely written in JS and lets us build complex interfaces through simple **components**! Yeah React UIs are just and only components (Keep it in mind 🙌)

React’s strong suit is handling stateful, data-driven interfaces with minimal code and minimal pain, and it has all the bells and whistles you’d expect from a modern web framework.

****

React is a javascript Library that can create both web and mobile apps ,

React js is meant to build web applications while React Native is for mobile apps

****

**How to install React?**

First, we need to have **Node js** installed on our machine since it’s the execution environment, coz we’ll be using **npm** a lot !!!

Secondo, I highly recommend one approach, and that’s using the officially recommended tool called create-react-app.

create-react-app is a command-line application, aimed at getting you up to speed with React in no time.

You start by using npx, which is an easy way to download and execute Node.js commands without installing them.

**BUT … Wait a minute**

A wise man said : *npx create-react-app* and *npm start* are a real a waste of time

**Vite.js — An Opinionated Frontend Build Tool**

**What’s Vite.js?**

Created by Evan You (also the creator of Vue.js), Vite.js is a next-generation, lightning-fast, front-end build tool that provides an amazing user experience.

Vite is a french word that means ‘fast’ and is pronounced as ‘vit’. and it is Trop vite meme 💖 😜 . We might talk about it in a later article, but let’s say that we’ll be using it to create our react App ***after the first time with npx create-react-app*** (the secret is in here !), and it will take less and less time to create our application and also to start the server with **npm run dev** command instead of **npm start**

So to recap :

**#For the first time or maybe if you're all time patient use this :**   
npx create-react-app my-app   
cd my-app   
npm start**#Later on and if you're nervous :p#If using NPM**$ npm init vite-app <project-name>  
$ cd <project-name>  
$ npm install  
$ npm run dev  
**#If using Yarn**$ yarn create vite-app <project-name>  
$ cd <project-name>  
$ yarn  
$ yarn dev

npx create-react-app will create some folders and files , I believe it’s good to know what they are exactly :

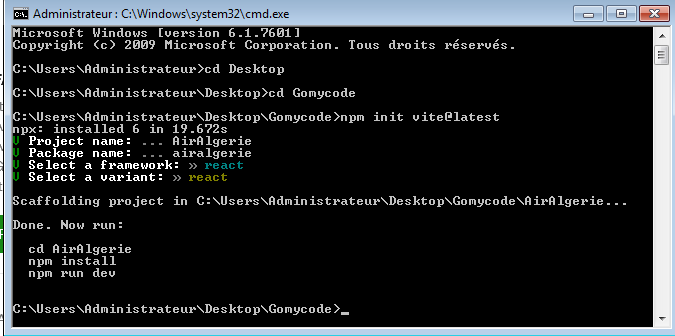
It is good to know these folders and files.

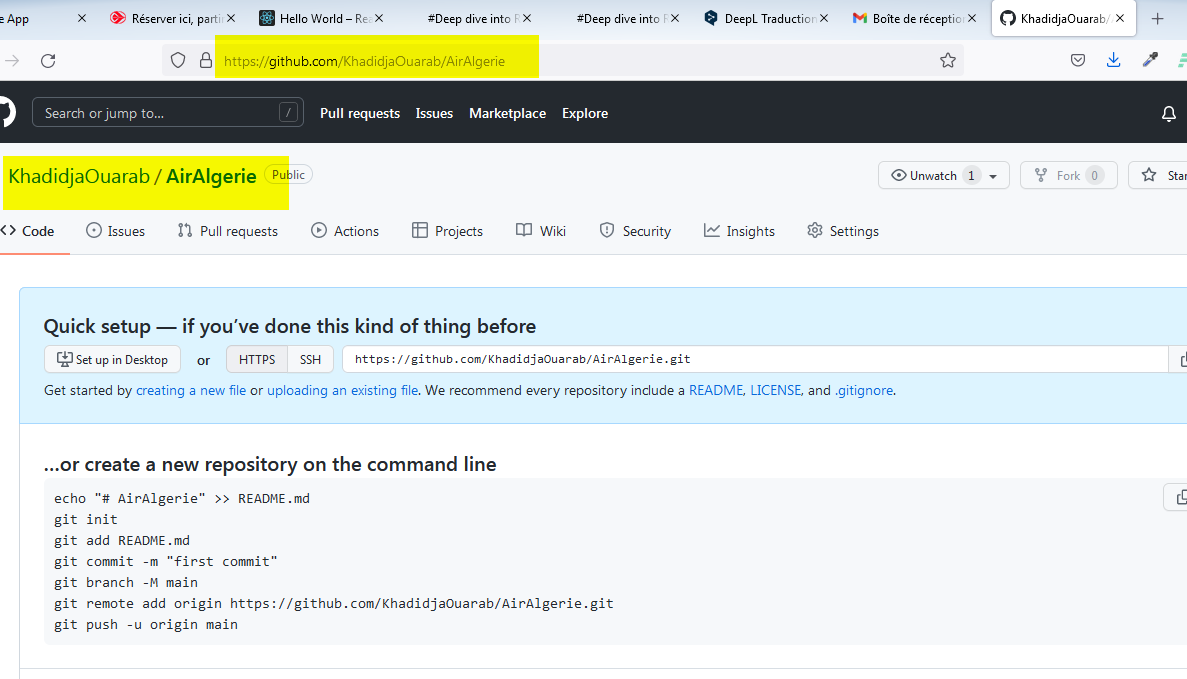
* **node\_modules** — stores all the necessary node packages of the React applications.
* **Public**
* index.html — the only HTML file we have in the entire application
* favicon.ico — an icon file
* manifest.json — is used to make the application a progressive web app
* other images — open graph images(open graph images are images which are visible when a link share on social media)
* robots.txt — information, if the website allows web scraping
* **src**
* App.css, index.css — are different CSS files
* index.js — a file which allows connecting all the components with index.html
* App.js — A file where we usually import most of the presentational components
* serviceWorker.js: is used to add progressive web app features
* setupTests.js — to write testing cases
* package.json- List of packages the applications uses
* .gitignore — React boilerplate comes with git initiated, and the .gitingore allows files and folders not to be pushed to GitHub
* [README.md](http://README.md) — Markdown file to write documentation
* yarn.lock or package-lock.json — a means to lock the version of the package

with **vite** project you’ll find a diffrent structure which is mainly the same but without the unuseful files . (so don’t panic !)

**VsCode Extentions that will save your time when coding with React :**

* The ***Bracket Pair Colorizer*** extension.
* The ***change-case*** extension.
* The ***Code Spell Checker*** extension.
* The ***Duplicate Selection*** extension.
* The ***EditorConfig for VS Code*** extension.
* The ***VSCode React Refactor*** extension.
* The ***npm Intellisense*** extension.
* The ***ESLint*** extension.
* The ***ES7 React/Redux/GraphQL/React-Native snippets*** extension.
* The ***Prettier — Code formatted*** extension.

****

****

**Files structure:**

**SRC:**

* **Components**
* **Pages : contains multiple component**
* **Assets : image, json file**

**Pour styler mes pages, j’ai plusieurs options:**

* **Dans un fichier CSS à part, après j’import le fichier puis j’utilise classname**
* **Styled component : si je veux écrire le css dans javascript 🡪 for some component**

**Il faut l’installer :**

**npm install styled-componet,**

**import styled from ‘styled-component’**

**const MyButton = styled.button ‘ color: green ‘;**

**< MyButton></ MyButton>**

* **Tailwind css for react 🡪 I choose it**

[**https://nerdcave.com/tailwind-cheat-sheet**](https://nerdcave.com/tailwind-cheat-sheet)

[**https://tailwindcss.com/docs/guides/create-react-app**](https://tailwindcss.com/docs/guides/create-react-app)

[**https://larainfo.com/blogs/how-to-install-tailwind-css-in-react**](https://larainfo.com/blogs/how-to-install-tailwind-css-in-react)

## Install Tailwind CSS

Install tailwindcss and its peer dependencies via npm, and then run the init command to generate both tailwind.config.js and postcss.config.js.

Terminal

**npm install -D tailwindcss postcss autoprefixer**

**npx tailwindcss init -p**

##  Configure your template paths

Add the paths to all of your template files in your tailwind.config.js file.

tailwind.config.js

**module.exports = { content: [ "./src/\*\*/\*.{js,jsx,ts,tsx}", ], theme: { extend: {}, }, plugins: [],}**

##  Add the Tailwind directives to your CSS

Add the @tailwind directives for each of Tailwind’s layers to your ./src/index.css file.

index.css

**@tailwind base;**

**@tailwind components;**

**@tailwind utilities;**

##  Start your build process

Run your build process with npm run start.

Terminal

**npm run build**

**Npm start if not vite, 🡪 if vite we run: npm run dev**

##  Start using Tailwind in your project

Start using Tailwind’s utility classes to style your content.

**App.js**

export default function App() {

return (

<h1 **className="text-3xl font-bold underline**"> Hello world! </h1>

)}

## What Is Tailwind CSS? [#](https://www.smashingmagazine.com/2020/02/tailwindcss-react-project/#what-is-tailwind-css)

[Tailwind CSS](https://tailwindcss.com/) is a **utility-based** low-level CSS framework intended to ease building web applications with speed and less focus to writing custom CSS, without leaving the comfort zone of your HTML code, yet achieve awesome interfaces.

For example, you could style a button with just a few classes (instead of always having to declare a single big class separately from your HTML and writing a bunch of properties to make something):

<button class="bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-4 rounded ml-4 mt-4">

Button

</button>

Other CSS frameworks (such as Bootstrap, Foundation, Bulma, and so on) present you with diverse predefined components (such as modals, buttons, alerts, cards). But with Tailwind CSS, you get to make your own, or you’re forced to make your own depending on your project model. Another way to put it, you actually own the components, and you get to harness your customization power on any component of your choice. This means that there is no more need to fight against the framework, and trying to figure out which classes need to be overridden in order to get results you initially aimed for.

## Why Use Tailwind CSS? [#](https://www.smashingmagazine.com/2020/02/tailwindcss-react-project/#why-use-tailwind-css)

Maybe you’re not quite ready to betray other frameworks yet, or you haven’t been convinced to embrace the goodies that come with Tailwind CSS. Allow me to give you a few reasons why you may want to consider Tailwind CSS.

### No Naming Conventions [#](https://www.smashingmagazine.com/2020/02/tailwindcss-react-project/#no-naming-conventions)

One of the most stressful parts of writing custom CSS is having to name classes. At every point, you’re pondering which class should be generic or specific. How do you organize them and ensure they’re cascaded? Tailwind CSS solves those problems seamlessly by providing **utility-based classes that can be used all the time**.

However, cases may arise where you need to name some classes. Sometimes this tends to happen when you need to extract certain components and use them later in your design (with the help of the @apply directives).

### Cache Benefits [#](https://www.smashingmagazine.com/2020/02/tailwindcss-react-project/#cache-benefits)

When writing custom CSS (or using any other traditional CSS framework), you always have to make changes to your CSS files when making changes in your designs. With Tailwind CSS, you need not worry a bit about that since you’re **using the same classes over and over again** within the markup. This means that you do not have to bust your CSS cache everytime in order to make small changes to your design.

## When Not To Use Tailwind CSS [#](https://www.smashingmagazine.com/2020/02/tailwindcss-react-project/#when-not-to-use-tailwind-css)

Are you saying I should always use Tailwind CSS for every project? Of course not! There are a few use cases where you may **not** want to use Tailwind CSS.

### If You’re Working On A Small Projects [#](https://www.smashingmagazine.com/2020/02/tailwindcss-react-project/#if-you-re-working-on-a-small-projects)

When you need to get started with a mini-project that has a very short deadline (especially something a few users would be using or only yourself), then Tailwind CSS is not the best option. In those cases, I’d recommend using Bootstrap, Foundation or any other framework instead. That’s because they come with predefined ready-to-use components (themes to get started with). With Tailwind CSS, you have to creatively create your own.

### If You’re A CSS Beginner [#](https://www.smashingmagazine.com/2020/02/tailwindcss-react-project/#if-you-re-a-css-beginner)

Before diving into Tailwind CSS for any project, its advisable to know CSS. Beginners that desire to use Tailwind CSS for web-based projects should first master CSS to a certain degree. It provides utility classes that are linked to the underlying CSS, therefore, only those with a solid knowledge of CSS can easily build with it.

### If You Dislike Adding A Lot Of Classes To Your Elements [#](https://www.smashingmagazine.com/2020/02/tailwindcss-react-project/#if-you-dislike-adding-a-lot-of-classes-to-your-elements)

When writing Tailwind CSS, you always have to write lots of classes, which makes your codebase (HTML) look busy and sometimes difficult to read. If you prefer keeping your code neat, you may want to consider writing custom CSS or use any other CSS framework (such as Bootstrap).

With these reasons, it’s about time to move over to the business of the day: **let’s set up Tailwind CSS in a React project together**!

## Installation (for standard modern project)

npm install react-icons --save

material UI: installation

-npm install @material-ui/core

-npm install @material-ui/icone

**Router:**

[**https://reactrouter.com/docs/en/v6/getting-started/installation**](https://reactrouter.com/docs/en/v6/getting-started/installation)

**Types of Hooks :**

**Basic Hooks : useState ,useEffect , use Context**

**Advanced Hooks : useReducer , useCallback , useMemo , useRef , useImperativeHandle , useLayoutEffect , useDebugValue**

**Custom Hooks : simple component prefixed with use : useCustomHook.jsx**

**We've seen also How to use ContextApi in React js :**

**# Context in React allows you to pass data to any component without "prop drilling".**

**props drilling : refers to the process you have to go through to get data to some part of the component tree (see the example bellow where we wanted to pass data from App to User)**

**App**

**|**

**Page**

**|**

**Layout**

**|**

**Header**

**|**

**User**

**To solve this problem we needed to use a State management tool , here we've used ContextApi in React (useContext)**

**To create a context in any React app, you need to follow 4 simple steps**

**-1- Create a context using createContext()**

**2- Create a provider**

**3- Add provider to the app**

**4- UseContext**

[**https://github.com/KelaiAffaf/Node-cheat-sheet/blob/main/cheatSheet.js**](https://github.com/KelaiAffaf/Node-cheat-sheet/blob/main/cheatSheet.js)

**What is Callback?**

Callback is an asynchronous equivalent of a function. A callback function is called at the completion of a given task. Node makes heavy use of callbacks. All the Node APIs are written in such a way that they support callbacks.

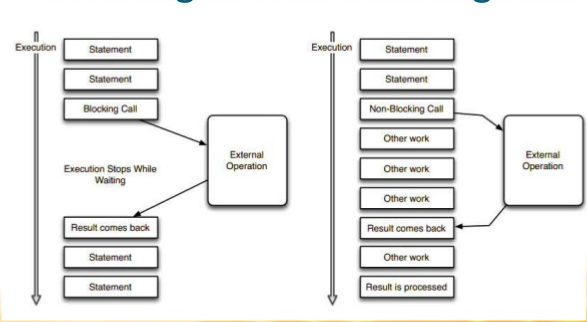
For example, a file reading function may start reading files and return control to the execution environment immediately so that the next instruction can be executed. Once file I/O is complete, it will call the callback function while passing the callback function, the content of the file as a parameter. So there is no blocking or wait for File I/O. This makes Node.js highly scalable as it can process a high number of requests without waiting for any function to return resu

**Blocking vs Non-Blocking**

These two examples explain the concept of blocking and non-blocking calls.

* The first example shows that the program blocks the call until it reads the file and only then it proceeds to end the program.
* The second example shows that the program does not wait for file reading and proceeds to print "Program Ended" and at the same time, the program, without blocking, continues reading the file.

Thus, a blocking program is executed in a sequence. From a programming point of view, it is easier to implement the logic but non-blocking programs do not execute in a sequence. In case a program needs to use any data to be processed, it should be kept within the same block to make it a sequential execution.



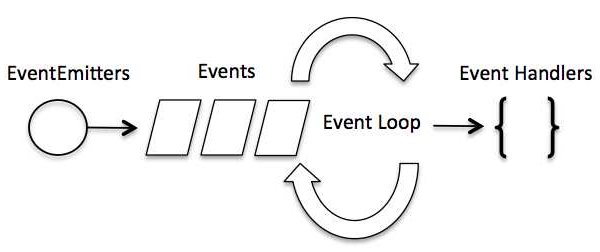
# Node.js - Event Loop

Node.js is a **single-threaded** application, but it can support concurrency via the concept of **event** and **callbacks**. Every Node.js API is asynchronous and being single-threaded, they use **async function calls** to maintain concurrency. Node uses observer patterns. Node thread keeps an event loop and whenever a task gets completed, it fires the corresponding event which signals the event-listener function to execute.

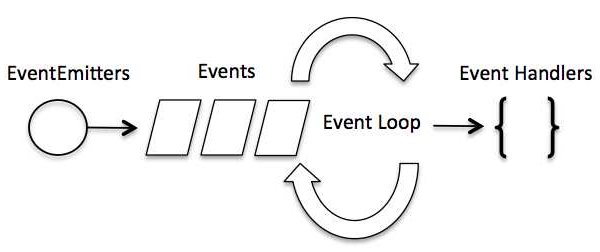
## Event-Driven Programming

Node.js uses events heavily and it is also one of the reasons why Node.js is pretty fast compared to other similar technologies. As soon as Node starts its server, it simply initiates its variables, declares functions and then simply waits for the event to occur.

In an event-driven application, there is generally a main loop that listens for events, and then triggers a callback function when one of those events is detected.



## Event Loop



Although events look quite similar to callbacks, the difference lies in the fact that callback functions are called when an asynchronous function returns its result, whereas event handling works on the observer pattern. The functions that listen to events act as Observers. Whenever an event gets fired, its listener function starts executing. Node.js has multiple built-in events available through events module and EventEmitter classes which are used to bind events and event-listeners as follows:

// Import events module

var events = require('events');

// Create an eventEmitter object

var eventEmitter = new events.EventEmitter();

The following is the syntax to bind an event handler with another event:

// Bind event and event handler as follows

eventEmitter.on('eventName', eventHandler);

We can fire an event programmatically as follows:

// Fire an event

eventEmitter.emit('eventName');

**How Do Node Applications Work?**

In Node Application, any async function accepts a callback as the last parameter and a callback function accepts an error as the first parameter. Let's revisit the previous example again. Create a text file named **input.txt** with the following content:

Dummy Text Just for Testing !!!!

Create a .js file named **main.js** having the following code:

var fs = require("fs");

fs.readFile('input.txt', function (err, data) {

if (err) {

console.log(err.stack);

return;

}

console.log(data.toString());

});

console.log("Program Ended");

Here fs.readFile() is an async function whose purpose is to read a file. If an error occurs during the read operation, then the **err object** will contain the corresponding error, or else data will contain the contents of the file. **readFile** passes (err) and data to the callback function after the read operation is complete, which finally prints this content:

Program Ended

Dummy Text Just for Testing !!!!

**EventEmitter Class**

Many objects in a Node emit events, for example, a net.Server emits an event each time a peer connects to it. An fs.readStream emits an event when the file is opened. All objects which emit events are the instances of events.EventEmitter.

As we have seen in the previous section, EventEmitter class lies in the events module. It is accessible via the following code:

// Import events module

var events = require('events');

// Create an eventEmitter object

var eventEmitter = new events.EventEmitter();

When an EventEmitter instance faces any error, it emits an 'error' event. When a new listener is added, 'newListener' event is fired and when a listener is removed, 'removeListener' event is fired.

EventEmitter provides multiple properties like on and emit. on property is used to bind a function with the event and emit is used to fire an event.

**Methods**

* **addListener(event, listener)** : Adds a listener at the end of the listeners array for the specified event. No checks are made to see if the listener has already been added. Multiple calls passing the same combination of event and listener will result in the listener being added multiple times. Returns emitter, so calls can be chained.
* **on(event, listener)** : Adds a listener at the end of the listeners array for the specified event. No checks are made to see if the listener has already been added. Multiple calls passing the same combination of event and listener will result in the listener being added multiple times. Returns emitter, so calls can be chained.
* **once(event, listener)** : Adds a one time listener to the event. This listener is invoked only the next time an event is fired, after which it is removed. Returns emitter, so calls can be chained.
* **removeListener(event, listener)** : Removes a listener from the listener array for the specified event. Caution − It changes the array indices in the listener array behind the listener. removeListener will remove, in most occasions, one instance of a listener from the listener array. If any single listener has been added multiple times to the listener array for the specified event, then removeListener must be called multiple times to remove each instance. Returns emitter, so calls can be chained.
* **removeAllListeners([event])** : Removes all listeners, or those of the specified event. It's not a good idea to remove listeners that were added elsewhere in the code, especially when it's on an emitter that you didn't create (e.g. sockets or file streams). Returns emitter, so calls can be chained.
* **setMaxListeners(n)** : By default, EventEmitters will print a warning if more than 10 listeners are added for a particular event. This is a useful default state which helps finding memory leaks. Obviously not all Emitters should be limited to 10. This function allows that to be increased. Set to zero for unlimited.
* **listeners(event)** : Returns an array of listeners for the specified event.
* **emit(event, [arg1], [arg2], [...])** : Executes each of the listeners in order with the supplied arguments. Returns true if the event had listeners, false otherwise.

**Class Methods**

**listenerCount(emitter, event)** : Returns the number of listeners for a given event.

* **Example:** create a file named listenerCount.js saved on your machine,

//import event module

var events = require('events');

// Create an eventEmitter object

var eventEmitter = new events.EventEmitter();

eventEmitter.emit('connection');

// count the number of listener to this event

eventListeners = require('events').EventEmitter.listenerCount(eventEmitter,'connection');

console.log(eventListeners + " Listner(s) listening to connection event");

Execute this piece of code by running the following command:

$ node listenerCount.js

* **Output:**

1 Listner(s) listening to connection even

**Events**

* **newListener**
  + **event** − String: the event's name
  + **listener** − Function: the event handler's function.  
    This event is emitted any time a listener is added. When this event is triggered, the listener may have not yet been added to the array of listeners for the event.
* **removeListener**
  + **event** − String: The event's name.
  + **listener** − Function: The event handler's function.  
    This event is emitted any time someone removes a listener. When this event is triggered, the listener may not yet have been removed from the array of listeners for the event.
* **Example**
* Create a .js file named main.js with the following Node.js code
* var events = require('events');
* var eventEmitter = new events.EventEmitter();
* // listener #1
* var listner1 = function listner1() {
* console.log('listner1 executed.');
* }
* // listener #2
* var listner2 = function listner2() {
* console.log('listner2 executed.');
* }
* // Bind the connection event with the listner1 function
* eventEmitter.addListener('connection', listner1);
* // Bind the connection event with the listner2 function
* eventEmitter.on('connection', listner2);
* var eventListeners = require('events').EventEmitter.listenerCount
* (eventEmitter,'connection');
* console.log(eventListeners + " Listner(s) listening to connection event");
* // Fire the connection event
* eventEmitter.emit('connection');
* // Remove the binding of listner1 function
* eventEmitter.removeListener('connection', listner1);
* console.log("Listner1 will not listen now.");
* // Fire the connection event
* eventEmitter.emit('connection');
* eventListeners = require('events').EventEmitter.listenerCount(eventEmitter,'connection');
* console.log(eventListeners + " Listner(s) listening to connection event");
* console.log("Program Ended.");
* Now run the main.js to see the result.
* $ node main.js
* Verify the Output
* 2 Listner(s) listening to connection event
* listner1 executed.
* listner2 executed.
* Listner1 will not listen now.
* listner2 executed.
* 1 Listner(s) listening to connection event
* Program Ended.

# Synchronous vs Asynchronous

## File System

Node implements File I/O using simple wrappers around standard POSIX functions. The Node File System (fs) module can be imported using the following syntax:

var fs = require("fs")

Every method in the FS module has synchronous as well as asynchronous forms. Asynchronous methods take the last parameter as the completion function callback and the first parameter of the callback function as an error. It is better to use an asynchronous method instead of a synchronous method, as the former never blocks a program during its execution, whereas the latter does.

## Example

Create a text file named **input.txt** with the following content:

Dummy Text For Testing !!!!

Let us create a .js file named **main.js** with the following code:

var fs = require("fs");

// Asynchronous read

fs.readFile('input.txt', function (err, data) {

if (err) {

return console.error(err);

}

console.log("Asynchronous read: " + data.toString());

});

// Synchronous read

var data = fs.readFileSync('input.txt');

console.log("Synchronous read: " + data.toString());

console.log("Program Ended");

Now run the main.js to see the result

$ node main.js

Verify the Output:

Synchronous read: Dummy Text For Testing !!!!

Program Ended

Asynchronous read: Dummy Text For Testing !!!!

Let's take a look at a set of good examples on major File I/O methods:

# Open a File

## Syntax

The following is the method's syntax for opening a file in asynchronous mode:

fs.open(path, flags[, mode], callback)

## Parameters

Here is the description of the parameters used :

* **path** − This is the string that has the file name and the path.
* **flags** − Flags indicate the behavior of the file to be opened. All possible values have been mentioned below.
* **mode** − It sets the file mode (permission and sticky bits), but only if the file was created. It defaults to 0666, readable and writeable.
* **callback** − This is the callback function which gets two arguments (err, fd).

## Flags

* **r** : Opens file for reading. An exception occurs if the file does not exist.
* **r+** : Opens file for reading and writing. An exception occurs if the file does not exist.
* **rs** : Opens file for reading in synchronous mode.
* **rs+** : Opens file for reading and writing, asking the OS to open it synchronously. See notes for 'rs' about using this with caution.
* **w** : Opens file for writing. The file is created (if it does not exist) or truncated (if it exists).
* **wx** : Like 'w' but fails if the path exists.
* **w+** : Opens file for reading and writing. The file is created (if it does not exist) or truncated (if it exists).
* **wx+** : Like 'w+' but fails if path exists.
* **a** : Opens file for appending. The file is created if it does not exist.
* **ax** : Like 'a' but fails if the path exists.
* **a+** : Opens file for reading and appending. The file is created if it does not exist.
* **ax+** : Like 'a+' but fails if the the path exists.

# \_\_filename

Node.js global objects are global in nature and they are available in all modules. We do not need to include these objects in our application, but instead, we can use them directly. These objects are modules, functions, strings and objects themselves. We will explain more in the upcoming slides.

The \_\_filename represents the filename of the code being executed. This is the resolved absolute path of the code file. For a main program, this is not necessarily the same filename used in the command line. The value inside a module is the path to that module's file.

## Example

Create a .js file named **main.js** with the following code:

// Let's try to print the value of \_\_filename

console.log( \_\_filename );

Now run the main.js to see the result:

$ node main.js

Based on the location of your program, it will print the main file name as follows:

/web/com/1427091028\_21099/main.js

# \_\_dirname

The \_\_dirname represents the name of the directory that the currently executing script resides in.

## Example

Create a .js file named **main.js** with the following code:

// Let's try to print the value of \_\_dirname

console.log( \_\_dirname );

Now run the main.js to see the result:

$ node main.js

Based on the location of your program, it will print current directory names as follows:

/web/com/1427091028\_21099

# setTimeout(cb, ms)

The **setTimeout(cb, ms)** global function is used to run **callback** (cb) after at least **ms milliseconds**. The actual delay depends on external factors like OS timer granularity and system load. A timer cannot span more than 24.8 days.

## Example

Create a .js file named main.js with the following code:

function printHello() {

console.log( "Hello, World!");

}

// Now call above function after 2 seconds

setTimeout(printHello, 2000);

Now run the main.js to see the result:

$ node main.js

Verify that the output is printed as **after a little delay**:

Hello, World!

**What is a Web Server?**

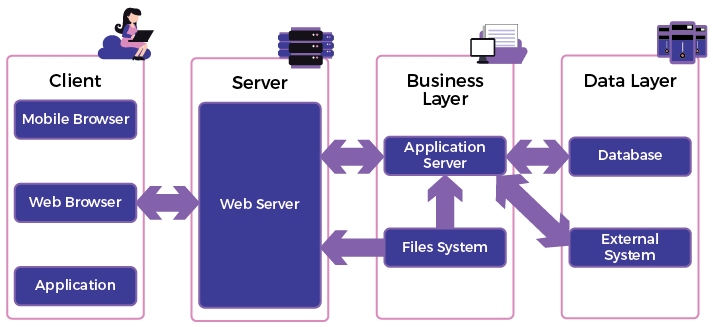
A Web Server is a software application which handles HTTP requests sent by the HTTP client, like web browsers, and returns web pages in response to the clients. Web servers usually deliver HTML documents along with images, style sheets, and scripts.

Most of the web servers support server-side scripts. They use scripting languages and redirect the task to an application server which retrieves data from a database and performs complex logic. Finally a result is sent to the HTTP client through the web server.

**Web Application Architecture**

A Web application is usually divided into four layers:

**Client ------------------------------NodeJS----------------------MONGOSSE-------------------MongoDB**



* **Client** : This layer consists of web browsers, mobile browsers or applications which can make HTTP requests to the web server.
* **Server** : This layer has the web server which can intercept the requests made by the clients and pass them the response.
* **Business** : This layer contains the application server which is utilized by the web server to do the required processing. This layer interacts with the data layer via the database or some external programs.
* **Data** : This layer contains the databases or any other source of data
* **Creating a Web Server using Node**
* Node.js provides an **HTTP** module which can be used to create an HTTP client of a server. The following is the bare minimum structure of the HTTP server which listens to the 8081 port.
* Create a js file named server.js :
* **File: server.js**
* var http = require('http');
* var fs = require('fs');
* var url = require('url');
* // Create a server
* http.createServer( function (request, response) {
* // Parse the request containing file name
* var pathname = url.parse(request.url).pathname;
* // Print the name of the file for which request is made.
* console.log("Request for " + pathname + " received.");
* // Read the requested file content from file system
* fs.readFile(pathname.substr(1), function (err, data) {
* if (err) {
* console.log(err);
* // HTTP Status: 404 : NOT FOUND
* // Content Type: text/plain
* response.writeHead(404, {'Content-Type': 'text/html'});
* } else {
* //Page found
* // HTTP Status: 200 : OK
* // Content Type: text/plain
* response.writeHead(200, {'Content-Type': 'text/html'});
* // Write the content of the file to response body
* response.write(data.toString());
* }
* // Send the response body
* response.end();
* });
* }).listen(8081);
* // Console will print the message
* console.log('Server running at http://127.0.0.1:8081/');

**What is Express.js ?**

After the big rise of Node.js, there are plenty of frameworks and library that have been created. The most used one for backend rendering is Express. That will be the topic of this Super Skill. So during this chapter we are going to explore:

* What is Express?
* How to setup an Express environment?
* What is routing via Express?
* What are middlewares and what is their use?
* How to work with a template engine?

# What is Express.js ?

The Express [website](https://expressjs.com/) describes Express as a "minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications." What does that really mean, exactly?  
Express provides basic tools that facilitate the creation of Node.js applications without obscuring Node.js features that you know and love.  
It provides a set of methods for processing HTTP requests and provides a middleware system that extends its functionality.  
It also makes it easier to manage the path (URL) of your application and not to mention that it uses templates.

**What is Express.js ?**

If you have wrote any serious apps only using the core Node.js modules, you must have likely found yourself reinventing the wheel by repeatedly writing the same code for the same tasks such as:

* Parsing of HTTP request bodies.
* Parsing of cookies.
* Managing sessions.
* Organizing routes with a chain of **if** conditions based on URL paths and HTTP request methods.
* Determining proper response headers based on data types.

**How Does Express Work ?**

Express.js usually has an entry point or in other words, a main file. In that file, we can perform the following steps:

1. Include third party dependencies.
2. Configure Express.js app settings such as the template engine and its files’ extensions.
3. Define middlewaress such as error handlers, static files folder, cookies and other parsers.
4. Define routes.
5. Connect to databases such as MongoDB, Redis or MySQL.
6. Start the app.

When Express.js app is running, it listens to requests. Each incoming request is processed through a defined chain of middlewares and routes starting from top to bottom. For example, we can have multiple functions handling each request and some of those functions will be in the middle (hence the name middleware):

1. Parse cookie information and go to the next step when done.
2. Parse parameters from the URL and go to the next step when done.
3. Get the information from the database based on the parameter's value, if a user is authorized (cookie/session), and go to the next step if there is a match.
4. Display the data and end the response.
5. **Installation**
6. Assuming you’ve already installed Node.js, you should create a directory to hold your application and make that your working directory.
7. $ mkdir myapp
8. $ cd myapp
9. Once we are inside the project folder, we can create a package.json manually in a text editor or with the npm init terminal command.
10. $ npm init
11. This command prompts you a number of things, such as the name and version of your application. For now, you can simply hit RETURN to accept the defaults for most of them, with the following exception:
12. entry point: (index.js)
13. Enter app.js, or whatever you want the name of the main file to be. If you want it to be index.js, hit RETURN to accept the suggested default file name.  
    Finally, we can install the module utilizing NPM
14. $ npm install express --save

**Express application generator**

Use the application generator tool, express-generator, to quickly create an application skeleton. Install the application generator as a global npm package and then launch it.

$ npm install -g express-generator

$ express

For example, the following creates an Express app named myapp. The app will be created in a folder named myapp in the current working directory and the view engine will be set to Pug:

$ express --view=pug myapp

The generated app has the following directory structure:

.

├── app.js

├── bin

│ └── www

├── package.json

├── public

│ ├── images

│ ├── javascripts

│ └── stylesheets

│ └── style.css

├── routes

│ ├── index.js

│ └── users.js

└── views

├── error.pug

├── index.pug

└── layout.pug

7 directories, 9 files

# Express application generator

Then install dependencies:

$ cd myapp

$ npm install

run the app with this command:

$ npm start

Then load <http://localhost:3000/> in your browser to access the app, you should see this common response



### Command Line Options

This generator can also be configured using command flags:

$ express -h

Usage: express [options] [dir]

Options:

-h, --help output usage information

--version output the version number

-e, --ejs add ejs engine support

--hbs add handlebars engine support

--pug and pug engine support

-H, --hogan and hogan.js engine support

--no-view generate without view engine

-v, --view <engine> add view <engine> support (ejs|hbs|hjs|jade|pug|twig|vash) (defaults to jade)

-c, --css <engine> add stylesheet <engine> support (less|stylus|compass|sass) (defaults to plain css)

--git add .gitignore

-f, --force force on non-empty directory

**Watching for File Changes**

Node.js applications are stored into memory and if we make changes to the source code, we need to restart the process, i.e. node.  
So to make our development process a lot easier, we will install a tool from npm, nodemon. This tool restarts our server as soon as we make a change in any of our files.  
To install nodemon, use the following command:

$ npm install -g nodemon

You can now start working on Express.

# Hello world Example(1/4)

Let’s create our first application using Express.

include this library:

const express = require('express');

Now we can create an application:

const app = express();

The application is a web server that will run locally on port 4000:

const port = 4000;

Let’s define a wildcard route (\*) with app.get() function:

app.get('\*', function(req, res){ res.end('Hello World'); });

The app.get() function above accepts [regular expressions](https://en.wikipedia.org/wiki/Regular_expression) of the URL patterns in a string format. In our example, we’re processing all URLs with the wildcard \* character.

The second parameter to the app.get() is a request handler. A typical Express.js request handler is similar to the one we pass as a callback to the native/core Node.js http.createServer() method.

Lastly, we start the Express.js web server and output a user-friendly terminal message in a callback:

app.listen(port, function() {

console.log('The server is running, ' +

' please, open your browser at http://localhost:%s',

port);

});

# Hello world Example(2/4)

The full code of the index.js file

const express = require('express');

const app = express();

const port = 4000;

app.get('\*', function(req, res){

res.end('Hello World');

});

app.listen(port, function(){

console.log('The server is running, ' +

' please, open your browser at http://localhost:%s',

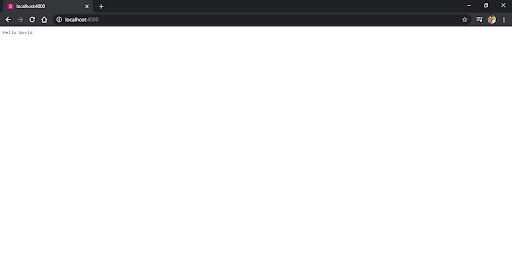
port);

});

To run the script, we execute node index.js from the project folder:

$ node index

Now, if you open your browser at <http://localhost:4000> (same as <http://127.0.0.1:4000>), you should see the “Hello World” message



**Hello world Example(3/4)**

We can make our example a little more interactive by echoing the name that we provide to the server along with the “Hello” phrase. To do so, add the following route before the all-encompassing route from the previous example.

app.get('/name/:user\_name', function(req,res) {

res.status(200);

res.set('Content-type', 'text/html');

res.send('<html><body>' +

'<h1>Hello ' + req.params.user\_name + '</h1>' +

'</body></html>'

);

});

Inside of the /name/:name\_route route, we set the proper HTTP status code (200 means okay), HTTP response headers and wrap our dynamic text in HTML body and h1 tags.

res.send() is a special Express.js method that conveniently goes beyond what our old friend from core HTTP module res.end() does. For example, the former automatically adds a Content-Length HTTP header for us. It also augments Content-Type based on the data provided to it.

# Hello world Example(4/4)

The full source code of the index.js file:

const express = require('express');

const app = express();

const port = 4000;

app.get('/name/:user\_name', function(req,res) {

res.status(200);

res.set('Content-type', 'text/html');

res.send('<html><body>' +

'<h1>Hello ' + req.params.user\_name + '</h1>' +

'</body></html>'

);

});

app.get('\*', function(req, res){

res.end('Hello World');

});

app.listen(port, function(){

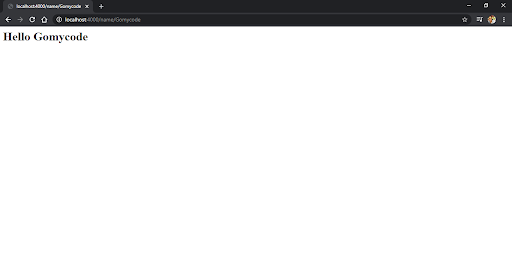
console.log('The server is running, ' +

' please, open your browser at http://localhost:%s',

port);

});

After shutting down the previous server and launching the index.js script, you’ll be able to see the dynamic response, e.g., by entering <http://localhost:4000/name/Gomycode> in your browser yields:



# Routes Introduction

Routes are one of the core concepts in Express and one of the things that make it really useful.  
Routing refers to determining how an application responds to a client request, at a particular endpoint, which is called a URI (or path) and a specific HTTP request method (GET, POST, and so on).  
Each route can have one or more handler functions which are executed when the route is matched.  
Route definition takes the following structure:

app.METHOD(PATH, HANDLER)

* app is an instance of express.
* METHOD is an [HTTP request method](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Request_methods), in lowercase.
* PATH is a path on the server.
* HANDLER is the function executed when the route is matched.

**Route methods (1/2)**

A route method is derived from one of the HTTP methods and it is attached to an instance of the express class.

The following code is an example of routes that are defined for the GET and the POST methods to the root of the app.

// GET method route

app.get('/', function (req, res) {

res.send('GET request to the homepage');

})

// POST method route

app.post('/', function (req, res) {

res.send('POST request to the homepage');

})

**Route methods (2/2)**

There is a special routing method, app.all(), used to load middleware functions at a path for all HTTP request methods. For example, the following handler is executed for requests to the route “/secret” whether using GET, POST, PUT, DELETE, or any other HTTP request method supported in the HTTP module.

app.all('/secret', function (req, res, next) {

console.log('Accessing the secret section ...');

next(); // pass control to the next handler

})

Express allows us to attach the methods to it.

app.get('/', function (req, res) {

res.send('GET request to the homepage');

}).post('/', function (req, res) {

res.send('POST request to the homepage');

}).all('/secret', function (req, res, next) {

console.log('Accessing the secret section ...');

next(); // pass control to the next handler

}).use(function(req, res, next){

res.status(404).send('Page introuvable !');

});

# Route paths (1/2)

The route path is used to define an endpoint where requests can be made. It is like a backend endpoint. In the Express, route paths can be:

* string
* string patterns
* [regular expressions](https://en.wikipedia.org/wiki/Regular_expression).

The characters ?, +, \*, and () are subsets of their regular expression counterparts.  
The hyphen (-) and the dot (.) are literally interpreted by string-based paths.

If you need to use the dollar character ($) in a path string, enclose it within ([ and ]). For example, the path string for requests at “/data/$book”, would be “/data/([$])book”.

* Here are some examples of route paths based on strings.

This route path will match requests to /**about**.

app.get('/about', function (req, res) {

res.send('about');

})

This route path will match requests to /**random.text**.

app.get('/random.text', function (req, res) {

res.send('random.text');

})

**Route paths (2/2)**

* Here are some examples of route paths based on **string patterns.**

This route path will match acd and abcd.

app.get('/ab?cd', function (req, res) {

res.send('ab?cd');

})

This route path will match abcd, abbcd, abbbcd, and so on.

app.get('/ab+cd', function (req, res) {

res.send('ab+cd');

})

This route path will match abcd, abxcd, abRANDOMcd, ab123cd, and so on.

app.get('/ab\*cd', function (req, res) {

res.send('ab\*cd');

})

This route path will match /abe and /abcde.

app.get('/ab(cd)?e', function (req, res) {

res.send('ab(cd)?e');

})

* Examples of route paths based on **regular expressions:**

This route path will match anything with an “a” in it.

app.get(/a/, function (req, res) {

res.send('/a/');

})

This route path will match butterfly and dragonfly, but not butterflyman, dragonflyman, and so on.

app.get(/.\*fly$/, function (req, res) {

res.send('/.\*fly$/');

})

**Route Parameters**

The route parameters are used to capture the values that are assigned to a particular position in the URL. They are called the URL segments.

The values obtained are made available in a req.params object, using the name of the route parameter specified in the path as the values' keys.

Route path: /users/:userId/books/:bookId

Request URL: http://localhost:3000/users/34/books/8989

req.params: { "userId": "34", "bookId": "8989" }

To define routes with route parameters, simply specify the route parameters in the path of the route as shown below.

app.get('/users/:userId/books/:bookId', function (req, res) {

res.send(req.params)

})

To have more control over the exact string that can be matched by a route parameter, you can append a regular expression in parentheses (()):

Route path: /user/:userId(\d+)

Request URL: http://localhost:3000/user/42

req.params: {"userId": "42"}

# Route handlers

We can have multiple request handlers, hence the name middleware. They accept the third parameter/function next and calling which (next()) to be executed will switch the execution flow from one handler to the next:  
For Example :

app.get('/api/v1/stories/:id', function(req,res, next) {

//do authorization

//if not authorized or there is an error

//return next(error);

//if authorized and no errors

return next();

}), function(req,res, next) {

//extract id and fetch the object from the database

//assuming no errors, save story in the request object

req.story = story;

return next();

}), function(req,res) {

//output the result of the database search

res.send(res.story);

});

Another useful technique is to pass callbacks as items of an array, thanks to the inner workings of arguments. [JavaScript mechanism](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments):

const authAdmin = function (req, res, next) {

...

return next();

}

const getUsers = function (req, res, next) {

...

return next();

}

const renderUsers = function (req, res) {

res.end();

}

const admin = [authAdmin, getUsers, renderUsers];

app.get('/admin', admin);

**app.route()**

You can create chainable route handlers for a route path by using app.route(). Because the path is specified at a single location, creating modular routes is helpful, as well as reducing redundancies and typos.  
Here is an example of chained route handlers that are defined by using app.route().

app.route('/book')

.get(function (req, res) {

res.send('Get a random book');

})

.post(function (req, res) {

res.send('Add a book');

})

.put(function (req, res) {

res.send('Update the book');

})

**app.route()**

The express.Router middleware allows us to group the route handlers for a particular part of a site together and access them using a common route-prefix.

The code below provides a concrete example of how we can create a route module and then use it in an *Express application*.  
First, we create routes for a **post** in a module named **post.js**. Then, the code imports the Express application object, uses it to get a Router object, and adds a couple of routes to it using the get() method. Lastly, the module exports the Router object.

**post.js** - **post** route module.

const express = require('express');

const router = express.Router();

// post page route.

router.route('/post/:slug')

.all(function(req, res, next) {

// runs each time

// we can fetch the post by id from the database

})

.get(function(req, res, next) {

//render post

})

.put(function(req, res, next) {

//update post

})

.post(function(req, res, next) {

//create new comment

})

.del(function(req, res, next) {

//remove post

});

module.exports = router

To use the router module in our main app file we first require() the route module (post.js). We then call use() on the Express application to add the router to the middleware handling path while specifying the URL path of '/blog'.

const post = require(‘./post’);

app.use(‘/blog, post);

The app will now be able to handle requests to /blog/post.

# Middleware Introduction

Middleware functions are functions that have access to the [request object](http://expressjs.com/en/4x/api.html#req) (req), [the response object](http://expressjs.com/en/4x/api.html#res) (res), and the next function in the application’s request-response cycle.

The next function is a function in the Express router that, when invoked, executes the middleware succeeding the current middleware.  
Middleware functions can perform the following tasks:

* Execute any code.
* Make changes to the request and the response objects.
* End the request-response cycle.
* Call the next middleware in the stack.

If the current middleware function does not end the request-response cycle, it must call next() to pass control to the next middleware function. Otherwise, the request will be left hanging.

**Writing middleware (1/2)**

Here is a simple example of a middleware function

//Simple request time logger

const myLogger = function (req, res, next) {

console.log("A new request received at " + Date.now());

next();

}

To load the middleware function, call app.use(), specifying which middleware function. For example, the following code loads the myLogger middleware function before the route to the root path (/).

const express = require('express');

const app = express();

//Simple request time logger

const myLogger = function (req, res, next) {

console.log("A new request received at " + Date.now());

next();

}

app.use(myLogger);

app.get('/', function (req, res) {

res.send('Hello World!')

})

app.listen(3000);

The above middleware is called for every request on the server. So after every request, we will get the following message in the console:

A new request received at 1584954785016

**Writing middleware (2/2)**

To restrict it to a specific route (and all its subroutes), provide that route as the first argument of app.use(). For Example,

const express = require('express');

const app = express();

//Middleware function to log request protocol

app.use('/things', function(req, res, next){

console.log("A request for things received at " + Date.now());

next();

});

// Route handler that sends the response

app.get('/things', function(req, res){

res.send('Things');

});

app.listen(3000);

Now, whenever you request any subroute of '/things', that will be the instance where it will log the time.

**Error-handling middleware**

Express JS comes with default error handling parameters. We can define error-handling middleware functions in the same way as other middleware functions, except error-handling functions have four arguments instead of three:

app.use(function (err, req, res, next) {

console.error(err.stack)

res.status(500).send('Something broke!')

})

In order to call an error-handling middleware, you simply pass the error to next(), like this:

app.get('/', (req, res, next) => {

next(new Error('I am passing you an error!'));

});

If you pass anything to the next() function (except the string 'route'), Express regards the current request as being an error and will skip any remaining non-error handling routing and middleware functions.

# Third Party Middlewares

A list of third party middlewares for Express are available [here](http://expressjs.com/en/resources/middleware.html). The following are some of the most commonly used middleware. We will also learn how to mount and use them.

**body-parser:** parse incoming request bodies in a middleware before your handlers. It's available under the req.body property.  
To mount body parser, we need to install it using

npm install --save body-parser

and to mount it, include the following lines in your index.js

const bodyParser = require('body-parser');

//To parse URL encoded data

app.use(bodyParser.urlencoded({ extended: false }))

//To parse json data

app.use(bodyParser.json())

To view all available options for the body-parser middleware, visit its GitHub page.

**cookie-parser:** It parses Cookie header and populates req.cookies with objects that have cookie names as keys. To mount cookie parser, we need to install it using

npm install --save cookie-parser

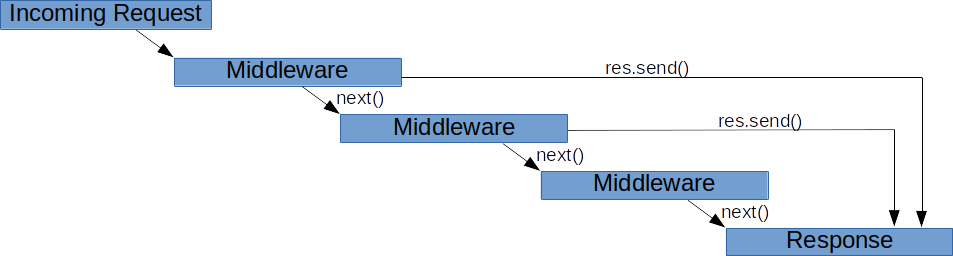
and to mount it, include the following lines in your index.js

var cookieParser = require('cookie-parser');

app.use(cookieParser())

**Middleware Order is Important**

When a request is received by Express, each middleware that matches the request is run in the order it is initialized in until there is a concluding action (like a response being sent).



So if an error occurs, all middlewares that are assigned to handle errors will be called in order until one of them does not call the next() function call.

**Express Routing RECAP**

Kudos to you for making it this far!

We'll try and do a little recap before we move on to other things.

After diving deep into this Super Skill, we have covered the basics of Express.js. It is a super powerful framework for backend rendering, but that is not all.

For the time being, we have only seen the server part in backend development. There are still more tools to learn. Let's jump into the next Super Skill.

# Tools - To update your resume

[Zety](https://zety.com/)  
[Resumonk](https://www.resumonk.com/)  
[Resume dot com](https://www.resume.com/)  
[VisualCV](https://www.visualcv.com/)  
[Cvmaker](https://cvmkr.com/)  
[Resume Genius](https://resumegenius.com/)  
[Resume builder](https://resumegenius.com/)  
[Resume Baker](http://www.resumebaker.com/)  
[Enhancv](https://enhancv.com/)

# Tools - To prepare your interview

[Ambitionbox](https://www.ambitionbox.com/)  
[AceTheInterview](https://acetheinterview.herokuapp.com/)  
[Geeksforgeeks](https://www.geeksforgeeks.org/)  
[Leetcode](https://leetcode.com/)  
[Gainlo](http://www.gainlo.co/#!/)  
[Careerup](https://www.careerup.com/)  
[Indiabix](https://www.indiabix.com/)