**Learning Outcomes**

After completing this Elective module a student will be able to:

1. Create Module & export, import
2. Introduction package.json file
3. File Handling
4. Create Event Driven Programming
5. Socket Programming
6. Create Own web server

# AngularJS | Modules

The AngularJS module defines the functionality of the application which is applied on the entire HTML page. It helps to link many components. So it is just a group of related components. It is a container which consists of different parts like controllers, services, directives.

**Note:** This modules should be made in a normal HTML files like index.html and no need to create a new project in VisualStudio for this section.

**How to create a Module:**

var app = angular.module("Module-name", []);

In this [] we can add a list of components needed but we are not including any components in this case. This created module is bound with any tag like div, body, etc by adding it to the list of modules.

|  |
| --- |
| <div ng-app = "module-name">      The code in which the module is required.  </div> |

**Adding a Controller:**

app.controller("Controller-name", function($scope) {

$scope.variable-name= "";

});

Here, we can add any number of variables in controller and use them in the html files, body of the tag in which the controller is added to that tag by writing:

|  |
| --- |
| <body>  <div ng-app="Module-name">      <div ng-controller="Controller-name">          {{variable-name}}      </div>  <!-- This wont get printed since its  not part of the div in which  controller is included -->  {{variable-name}}  </div>  </body> |

**Module and Controllers in Files:** While we can make modules and controllers in the same file along with the HTML file which requiring it however we may want to use this module in some other file. Hence this will lead to redundancy so we will prefer to create Module, Controller and HTML file separately. The Module and Controller are to be stored by using .js files and in order to use them in the HHTML file we have to include them in this way:

**Example:**

**DemoComponent.js**

|  |
| --- |
| // Here the Component name is DemoComponent  // so saving the file as DemoComponent.js  app.controller('DemoController', function($scope) {        $scope.list = ['A', 'E', 'I', 'O', 'U'];      $scope.choice = 'Your choice is: ADIT IBM ';        $scope.ch = function(choice) {          $scope.choice = "Your choice is: " + choice;      };        $scope.c = function() {          $scope.choice = "Your choice is: " + $scope.mychoice;      };  }); |

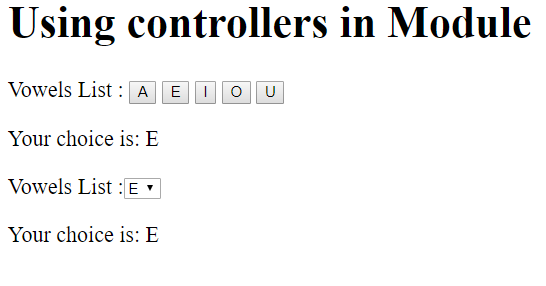
Module-name: **DemoApp.js**

|  |
| --- |
| var app = angular.module('DemoApp', []); |

**index.html file**

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>      <title>          Modules and Controllers in Files      </title>  </head>  <body ng-app="DemoApp">      <h1>          Using controllers in Module      </h1>      <script src=  "<https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js>">      </script>        <script src="DemoApp.js"></script>      <script src="DemoController"></script>      <div ng-app="DemoApp" ng-controller="DemoController">          Vowels List : <button ng-click="ch('A')" >A</button>          <button ng-click="ch('E')" >E</button>          <button ng-click="ch('I')" >I</button>          <button ng-click="ch('O')" >O</button>          <button ng-click="ch('U')" >U</button>          <p>{{ choice }}</p>          Vowels List :          <select  ng-options="option for option in list"              ng-model="mychoice" ng-change="c()">          </select>          <p>{{ choice }}</p>      </div>  </body>    </html> |

**Output:**



**Note:** It makes sure the module and component files are in the same folder otherwise provide the path in which they are saved and run.

**Directives in a Module:** To add a directive in module follow the steps:

Creating a module like we did earlier:

var app = angular.module("DemoApp", []);

Creating a directive:

app.directive("Directive-name", function() {

return {

template : "string or some code which is to be executed"

};

});

**Example:**

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>      <title>          Modules and Controllers in Files      </title>      <script src=  "<https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js>">      </script>  </head>  <body>    <div ng-app="GFG" w3-test-directive></div>    <script>    var gfg\_app = angular.module("GFG", []);    gfg\_app.directive("w3TestDirective", function() {        return {            template : "Welcome to ADIT IBM!"        };    });    </script>  </body>  </html> |

**Output:**

Welcome to ADIT IBM!

**Import Module**

AngularJS supports modular approach. Modules are used to separate logic such as services, controllers, application etc. from the code and maintain the code clean. We define modules in separate js files and name them as per the module.js file. In the following example, we are going to create two modules −

* **Application Module** − used to initialize an application with controller(s).
* **Controller Module** − used to define the controller.

Application Module

Here is a file named *mainApp.js* that contains the following code −

var mainApp = angular.module("mainApp", []);

Here, we declare an application **mainApp** module using angular.module function and pass an empty array to it. This array generally contains dependent modules.

Controller Module

studentController.js

mainApp.controller("studentController", function($scope) {

$scope.student = {

firstName: "Mahesh",

lastName: "Parashar",

fees:500,

subjects:[

{name:'Physics',marks:70},

{name:'Chemistry',marks:80},

{name:'Math',marks:65},

{name:'English',marks:75},

{name:'Hindi',marks:67}

],

fullName: function() {

var studentObject;

studentObject = $scope.student;

return studentObject.firstName + " " + studentObject.lastName;

}

};

});

Here, we declare a controller **studentController** module using mainApp.controller function.

Use Modules

<div ng-app = "mainApp" ng-controller = "studentController">

...

<script src = "mainApp.js"></script>

<script src = "studentController.js"></script>

</div>

Here, we use application module using ng-app directive, and controller using ngcontroller directive. We import the mainApp.js and studentController.js in the main HTML page.

**Example**

The following example shows use of all the above mentioned modules.

testAngularJS.htm

<html>

<head>

<title>Angular JS Modules</title>

<script src = "https://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"></script>

<script src = "/angularjs/src/module/mainApp.js"></script>

<script src = "/angularjs/src/module/studentController.js"></script>

<style>

table, th , td {

border: 1px solid grey;

border-collapse: collapse;

padding: 5px;

}

table tr:nth-child(odd) {

background-color: #f2f2f2;

}

table tr:nth-child(even) {

background-color: #ffffff;

}

</style>

</head>

<body>

<h2>AngularJS Sample Application</h2>

<div ng-app = "mainApp" ng-controller = "studentController">

<table border = "0">

<tr>

<td>Enter first name:</td>

<td><input type = "text" ng-model = "student.firstName"></td>

</tr>

<tr>

<td>Enter last name: </td>

<td><input type = "text" ng-model = "student.lastName"></td>

</tr>

<tr>

<td>Name: </td>

<td>{{student.fullName()}}</td>

</tr>

<tr>

<td>Subject:</td>

<td>

<table>

<tr>

<th>Name</th>

<th>Marks</th>

</tr>

<tr ng-repeat = "subject in student.subjects">

<td>{{ subject.name }}</td>

<td>{{ subject.marks }}</td>

</tr>

</table>

</td>

</tr>

</table>

</div>

</body>

</html>

mainApp.js

var mainApp = angular.module("mainApp", []);

studentController.js

mainApp.controller("studentController", function($scope) {

$scope.student = {

firstName: "Mahesh",

lastName: "Parashar",

fees:500,

subjects:[

{name:'Physics',marks:70},

{name:'Chemistry',marks:80},

{name:'Math',marks:65},

{name:'English',marks:75},

{name:'Hindi',marks:67}

],

fullName: function() {

var studentObject;

studentObject = $scope.student;

return studentObject.firstName + " " + studentObject.lastName;

}

};

});

**Output**

Open the file *textAngularJS.htm* in a web browser. See the result.

## AngularJS Sample Application

|  |  |
| --- | --- |
| Enter first name: | Mahesh |
| Enter last name: | Parashar |
| Name: | Mahesh Parashar |
| Subject: | |  |  | | --- | --- | | **Name** | **Marks** | | Physics | 70 | | Chemistry | 80 | | Math | 65 | | English | 75 | | Hindi | 67 | |

**Introduction package.json file**

**What Is package.json?**

Learn all about a Node project's package.json, npm’s configuration file housed in the root directory of your project. In this tutorial, learn how to manage metadata in an initial package.json file such as name, version, description, and keywords, as well as dependencies and devDependencies. Learn how to run, develop, and optionally publish your project to NPM.

In this tutorial we'll:

* Learn how *package.json* relates to your project
* Identify important fields and metadata
* Understand how to manage *package.json*

**Goal**

Understand what a package.json file is, how it relates to your project, and what common properties we need to know about.

**Get to know package.json**

If you've worked with Node.js before, you have likely encountered a package.json file. It is a [JSON](http://www.json.org/) file that lives in the root directory of your project. Your package.json holds important information about the project. It contains human-readable metadata about the project (like the project name and description) as well as functional metadata like the package version number and a list of dependencies required by the application.

An example package.json might look like this:

{

"name": "my-project",

"version": "1.5.0",

"description": "Express server project using compression",

"main": "src/index.js",

"scripts": {

"start": "node index.js",

"dev": "nodemon",

"lint": "eslint \*\*/\*.js"

},

"dependencies": {

"express": "^4.16.4",

"compression": "~1.7.4"

},

"devDependencies": {

"eslint": "^5.16.0",

"nodemon": "^1.18.11"

},

"repository": {

"type": "git",

"url": "https://github.com/osiolabs/example.git"

},

"author": "Jon Church",

"contributors": [{

"name": "Amber Matz",

"email": "example@example.com",

"url": "https://www.osiolabs.com/#team"

}],

"keywords": ["server", "osiolabs", "express", "compression"]

}

**What is the purpose of package.json?**

Your project's package.json is the central place to configure and describe how to interact with and run your application. It is used by the npm CLI (and yarn) to identify your project and understand how to handle the project's dependencies. It's the package.json file that enables npm to start your project, run scripts, install dependencies, publish to the NPM registry, and many other useful tasks. The npm CLI is also the best way to manage your package.json because it helps generate and update your package.json file throughout a project's life.

Your package.json fills several roles in the lifecycle of your project, some of which only apply for packages published to NPM. If you're not publishing your project to the NPM registry or otherwise making it publicly available to others, your package.json is still essential to the development flow.

Your project also must include a package.json before any packages can be installed from NPM. This is probably the top reason why you need one in your project.

**Common fields in package.json**

Let's look at some of the most common and important fields that can be in a package.json, to better understand how to use and manage this essential file. Some are required for publishing to NPM, while others help the npm CLI run the application or install dependencies.

There are more fields than the ones we cover, and you can read about the rest in the [documentation](https://docs.npmjs.com/files/package.json), but these are the essential package.json properties to understand.

### name

"name": "my-project"

The name field defines the name of the package. When publishing to the NPM registry, this is the name the package will be listed under. It must be no more than 214 characters, only lowercase letters, and it must be URL-safe (hyphens and underscores allowed, but no spaces or other characters disallowed in URLs).

If publishing your package to NPM, the name property is required and must be unique. You'll receive an error if trying to publish a package under a name that is currently used on the NPM registry. If you aren't developing a package which you'll eventually publish, the name does not have to be unique.

### version

"version": "1.5.0",

The version field is very important for any published package, and required before publishing. It is the current version of the software that the package.json is describing.

You are not required to use [SemVer](https://docs.npmjs.com/about-semantic-versioning), but it is the standard used in the Node.js ecosystem and highly recommended. For an unpublished package, this property isn't strictly required. Typically, the version number is bumped according to SemVer before publishing new versions to NPM.

This workflow isn't typically used when a package is not being relied upon as a dependency, or the package isn't being published to NPM. But if a package is being used as a dependency, keeping the version field up to date is very important to make sure others are using the proper version of a package. [Learn more about semantic versioning](https://heynode.com/tutorial/how-use-semantic-versioning-npm).

### license

This is a very important but often overlooked property. The license field lets us define what license applies to the code the package.json is describing. Again, this is very important when publishing a project to the NPM registry, as the license may limit the use of your software by some developers or organizations. Having a clear license in place helps clearly define what terms the software is able to be used under.

The value of this field will usually be the license's identifier code -- a string like "MIT" or "ISC" for the [MIT](https://opensource.org/licenses/MIT) license and [ISC](https://opensource.org/licenses/ISC) license respectively. If you don't wish to provide a license, or explicitly do not want to grant use of a private or unpublished package, you can put "UNLICENSED" as the license. [Choose a License](https://choosealicense.com/) is a helpful resource if you're not sure which license to use.

### author and contributors

"author": "Jon Church jon@example.com https://www.osioslabs.com/#team",

"contributors": [{

"name": "Amber Matz",

"email": "example@example.com",

"url": "https://www.osiolabs.com/#team"

}],

The author and contributor fields function similarly. They are both "people" fields which can be either a string in the format of "Name <email> <url>" , or an object with fields name, email, url. The email and url are both optional.

Author is for a single person, and contributors is an array of people.

These fields are a useful way to list contacts for a public project, as well as share credit with contributors.

### description

The description field is used by the NPM registry for published packages, to describe the package in search results and on the [npmjs.com](https://www.npmjs.com/) website.

This string is used to help surface packages when users search the NPM registry. This should be a short summary of what the package is for.

It can also be useful as simple documentation for your project, even if you aren't publishing it to the NPM registry.

### keywords

"keywords": ["server", "osiolabs", "express", "compression"]

The keywords field is an array of strings, and serves a similar purpose to the description. This field is indexed by the NPM registry to help find packages when someone searches for them. Each value in the array is one keyword associated with your package.

This field doesn't have much use if you're not publishing to the NPM registry, and you can feel free to omit it.

### main

"main": "src/index.js",

The main field is a functional property of your package.json. This defines the entry point into your project, and commonly the file used to start the project.

If your package (let's say its name is foo-lib) is installed by a user, then when a user does require('foo-lib'), it is the module.exports property of the file listed in the main field that is returned by require.

This is commonly an index.js file in the root of your project, but it can be any file you choose to use as the main entry-point to your package.

### scripts

"scripts": {

"start": "node index.js",

"dev": "nodemon"

}

The scripts field is another functional piece of metadata in your package.json. The scripts property takes an object with its keys being scripts we can run with npm run <scriptName>, and the value is the actual command which is run. These are typically terminal commands, which we put into the scripts field so we can both document them and reuse them easily.

Scripts are powerful tools that the npm CLI can use to run tasks for your project. They can do the job of most task runners used during development. [Learn more about npm scripts](https://heynode.com/tutorial/what-are-npm-scripts).

### repository

"repository": {

"type": "git",

"url": "https://github.com/osiolabs/example.git"

}

You can record the repository the code for a project lives in by providing the repository field. This field is an object which defines the url where the source code is located, and what type of version control system it uses. For open source projects, this is likely GitHub or Bitbucket with Git as the version control system.

An important note is that the URL field is meant to point to where the version control can be accessed from, not just the released code base.

### dependencies

"dependencies": {

"express": "^4.16.4",

"compression": "~1.7.4"

}

This is one of the most important fields in your package.json, and likely the entire reason you need one. All of the dependencies your project uses (the external code that the project relies on) are listed here. When a package is installed using the npm CLI, it is downloaded to your node\_modules/ folder and an entry is added to your dependencies property, noting the name of the package and the installed version.

The dependency field is an object with package names as keys, and a version or version range as a value. From this list, npm knows what packages to fetch and install (and at what versions) when npm install is run in the directory. The dependency field of your package.json is at the heart of your project, and defines the external packages your project requires.

The carets (^) and tildes (~) you see in the dependency versions are notation for version ranges defined in SemVer. [Learn more about dependency versions and SemVer](https://heynode.com/tutorial/how-use-semantic-versioning-npm).

### devDependencies

"devDependencies": {

"nodemon": "^1.18.11"

}

Similar to the dependencies field, but for packages which are only needed during development, and aren't needed in production.

An example would be using a tool to reload your project during development, like [nodemon](https://www.npmjs.com/package/nodemon), which we have no use for once the application is deployed and in production.

The devDependencies property lets us explicitly note which dependencies aren't needed in production. When installing your app in a production environment, you can use npm install --production to only install what is listed in the dependency field of package.json.

Recording a devDependency is a great way to document what tools are needed for the app during development. To install a package from npm as a devDependency, you can run npm install --save-dev <package>.

There is another way the devDependencies property is useful to us, and that's by using them in our npm scripts. [Learn more about using devDependencies in npm scripts](https://heynode.com/tutorial/what-are-npm-scripts).

**Manage your package.json**

A package.json file must be valid JSON. This means any missing commas, unclosed quotes, or other formatting errors will prevent npm from interacting with the package.json. If you do introduce an error, the next time you run an npm command you will see an error from npm. It's recommended to use the npm CLI for updating and managing your package.json when possible, to avoid accidentally introducing errors to your package.json, and to make managing your dependencies easier.

Using npm init to [create your package.json](https://heynode.com/tutorial/create-packagejson-file) will help to ensure you generate a valid file.

Dependencies are best managed by using npm's commands npm install, npm uninstall, and npm update, so your package.json and node\_modules/ folder are kept in sync. Manually adding a dependency listing will require you to run npm install before the dependency is actually installed to your project.

Because our package.json is only where we record dependencies, and our node\_modules/ folder is where the actual code for dependencies is installed, manually updating the dependency field of package.json does not immediately reflect the state of our node\_modules/ folder. That's why you want to use npm to help manage dependencies, because it will update both the package.json and node\_modules/ folder in tandem.

You can always edit your package.json manually in your text editor and make changes. That works well for most fields, so long as you're careful not to introduce any JSON formatting errors. We recommend you use the npm CLI commands wherever you can, however.

**Recap**

The package.json file is the heart of any Node project. It records important metadata about a project which is required before publishing to NPM, and also defines functional attributes of a project that npm uses to install dependencies, run scripts, and identify the entry point to our package.

Not all fields available in package.json will apply to you, but we can achieve some powerful benefits by recording information about our application in its package.json file. Understanding the role of package.json and how it relates to npm is an important part of developing Node.js apps, and increasingly an important part of the JavaScript ecosystem.

# How to get file content and other details in AngularJS?

We can get the file content by using some basic angular functions and other details like the name and size of the file in AngularJS. To understand it look into the below example where both HTML and JS files are implemented.

**Note:**Consider below two files are of same component in angular.

**app.module.html:**

|  |
| --- |
| <!-- Script for display  data in particular format -->  <!DOCTYPE html>  <html>  <script src=  "<https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js>">    </script>  <body ng-app="myApp">      <div ng-controller="MyCtrl">          <input type="file" id="myFileInput" />          <button ng-click="submit()"> Submit</button>          <br /><br />          <h1>              Filename: {{ fileName }}          </h1>          <h2>              File size: {{ fileSize }} Bytes          </h2>          <h2>              File Content: {{ fileContent }}          </h2>      </div>  </body>  </html> |

**Output:**

Graphical user interface

Description automatically generated with medium confidence

In the above HTML file we have simply made a structure to how it should be looked on the webpage. For that, we have used some angular stuff like ‘ng-controller’ and also doubly curly brackets which we will implement in the below javascript code.

**app.module.ts:**

|  |
| --- |
| import { BrowserModule } from          '@angular/platform-browser';  import { NgModule } from '@angular/core';  import { AppComponent } from './app.component';  import { BrowserAnimationsModule } from          '@angular/platform-browser/animations';  import { FormsModule, ReactiveFormsModule }          from '@angular/forms';  import { MatInputModule } from          '@angular/material/input';  import { MatDialogModule } from          '@angular/material/dialog';  import { MatFormFieldModule } from          '@angular/material/form-field';  import { MatIconModule } from          '@angular/material/icon';    @NgModule({      declarations: [          AppComponent,      ],      imports: [          BrowserModule,          FormsModule,          BrowserAnimationsModule,          MatInputModule,          MatFormFieldModule,          MatIconModule,          MatDialogModule,      ],      bootstrap: [AppComponent]  })  export class AppModule { } |

**app.component.ts:**

|  |
| --- |
| // Code to get file content  // and other data  import { Component, OnInit }          from '@angular/core';  // Imports  import { FormGroup, FormControl,            } from '@angular/forms';  @Component({      selector: 'app-root',      templateUrl: './app.component.html',      styleUrls: ['./app.component.scss']  })  export class AppComponent implements OnInit {      constructor() { }      ngOnInit() {      }        var myApp = angular.module('myApp', []);      myApp.controller('MyCtrl', function ($scope) {          // Intially declaring empty string          // and assigning size to zero          $scope.fileContent = '';          $scope.fileSize = 0;          $scope.fileName = '';          // Implementing submit function          $scope.submit = function () {              var file = document.getElementById("myFileInput")                                            .files[0];              if(file) {                  var Reader = new FileReader();                  Reader.readAsText(file, "UTF-8");                  Reader.onload = function (evt) {                        // Getting required result                      // of the file                      $scope.fileContent = Reader.result;                      $scope.fileName = document.getElementById(                                       "myFileInput").files[0].name;                      $scope.fileSize = document.getElementById(                                        "myFileInput").files[0].size;;                  }         // Printing error if data         //is not proper              Reader.onerror = function (evt) {                  $scope.fileContent = "error";              }            }         }      }  }); |

**Output:**  
Text, letter

Description automatically generated

# Explain Event-Driven Programming in Node.js

**Event-Driven Programming in Node.js:**Node.js makes extensive use of events which is one of the reasons behind its speed when compared to other similar technologies. Once we start a Node.js server, it initializes the variables and functions and then listens for the occurrence of an event.

Event-driven programming is used to synchronize the occurrence of multiple events and to make the program as simple as possible. The basic components of an Event-Driven Program are:

* A callback function ( called an event handler) is called when an event is triggered.
* An event loop that listens for event triggers and calls the corresponding event handler for that event.

Diagram

Description automatically generated

Ref : https://media.geeksforgeeks.org/wp-content/uploads/20211017211104/EDP1drawio.png

A function that listens for the triggering of an event is said to be an ‘Observer’. It gets triggered when an event occurs. Node.js provides a range of events that are already in-built. These ‘events’ can be accessed via the ‘events’ module and the EventEmitter class. Most of the in-built modules of Node.js inherit from the EventEmitter class

**EventEmitter:**The EventEmitter is a Node module that allows objects to communicate with one another. The core of Node’s asynchronous event-driven architecture is EventEmitter. Many of Node’s built-in modules inherit from EventEmitter.

The idea is simple – emitter objects send out named events, which trigger listeners that have already been registered. Hence, an emitter object has two key characteristics:

* **Emitting name events:** The signal that something has happened is called emitting an event. A status change in the emitting object is often the cause of this condition.
* **Registering and unregistering listener functions:** It refers to the binding and unbinding of the callback functions with their corresponding events.

**Event-Driven Programming Principles:**

* A suite of functions for handling the events. These can be either blocking or non-blocking, depending on the implementation.
* Binding registered functions to events.
* When a registered event is received, an event loop polls for new events and calls the matching event handler(s).

**Implementation:  Filename: app.js**

|  |
| --- |
| // Import the 'events' module  const events = require('events');    // Instantiate an EventEmitter object  const eventEmitter = new events.EventEmitter();    // Handler associated with the event  const connectHandler = function connected() {      console.log('Connection established.');        // Trigger the corresponding event      eventEmitter.emit('data\_received');  }    // Binds the event with handler  eventEmitter.on('connection', connectHandler);    // Binds the data received  eventEmitter.on(      'data\_received', function () {          console.log('Data Transfer Successful.');      });    // Trigger the connection event  eventEmitter.emit('connection');    console.log("Finish"); |

The above code snippet binds the handler named ‘connectHandler’ with the event ‘connection’’. The callback function is triggered when the event is emitted.

Run the *app.js* file using the following command:

node app.js

**Output:**

Connection established.

Data Transfer Successful.

Finish

**Advantages of Event-Driven Programming:**

* **Flexibility:** It is easier to alter sections of code as and when required.
* **Suitability for graphical interfaces:** It allows the user to select tools (like radio buttons etc.) directly from the toolbar
* **Programming simplicity:** It supports predictive coding, which improves the programmer’s coding experience.
* **Easy to find natural dividing lines:**Natural dividing lines for unit testing infrastructure are easy to come by.
* **A good way to model systems:**Useful method for modeling systems that must be asynchronous and reactive.
* **Allows for more interactive programs:**It enables more interactive programming. Event-driven programming is used in almost all recent GUI apps.
* **Using hardware interrupts:**It can be accomplished via hardware interrupts, lowering the computer’s power consumption.
* **Allows sensors and other hardware:**It makes it simple for sensors and other hardware to communicate with software.

**Disadvantages of Event-Driven Programming:**

* **Complex:** Simple programs become unnecessarily complex.
* **Less logical and obvious:**The flow of the program is usually less logical and more obvious
* **Difficult to find error:**Debugging an event-driven program is difficult
* **Confusing:** Too many forms in a program might be confusing and/or frustrating for the programmer.
* **Tight coupling:** The event schema will be tightly coupled with the consumers of the schema.
* **Blocking:**Complex blocking of operations.

**Relation between Event-Driven Programming and Object-Oriented Programming:**We can combine Object-oriented Programming (OOP) and Event-driven programming (EDP) and use them together in the same code snippet.

**When OOP is used with EDP:**

* All OOP fundamentals (encapsulation, inheritance, and polymorphism) are preserved.
* Objects get the ability to post-event notifications and subscribe to event notifications from other objects.

**How to distinguish between OOP with and without EDP:**The control flow between objects is the distinction between OOP with and without EDP. On a method call in OOP without EDP, control flows from one object to another. The primary function of an object is to call the methods of other objects.

However, on event notification, control in OOP with EDP moves from one object to another. Object subscribes to notifications from other objects, waits for notifications from those objects, performs work based on the notification, and then publishes its own notifications.

# How To Create a Real-Time App with Socket.IO, Angular, and Node.js

### **Introduction**

WebSocket is the internet protocol that allows for full-duplex communication between a server and clients. This protocol goes beyond the typical HTTP request and response paradigm. With WebSockets, the server may send data to a client without the client initiating a request, thus allowing for some very interesting applications.

In this tutorial, you will build a real-time document collaboration application (similar to Google Docs). We’ll be using the [Socket.IO](https://socket.io/) Node.js server framework and [Angular 7](https://www.digitalocean.com/community/tutorials/angular-angular-7) to accomplish this.

You can find the complete [source code for this example project on GitHub](https://github.com/alligatorio/socket-example).

## Prerequisites

To complete this tutorial, you will need:

* Node.js installed locally, which you can do by following [How to Install Node.js and Create a Local Development Environment](https://www.digitalocean.com/community/tutorial_series/how-to-install-node-js-and-create-a-local-development-environment).
* A modern web browser that [supports WebSocket](https://caniuse.com/#search=websocket).

This tutorial was originally written in an environment consisting of Node.js v8.11.4, npm v6.4.1, and Angular v7.0.4.

This tutorial was verified with Node v14.6.0, npm v6.14.7, Angular v10.0.5, and [Socket.IO](http://socket.io/) v2.3.0.

## Step 1 — Setting Up the Project Directory and Creating the Socket Server

First, open your terminal and create a new project directory that will hold both our server and client code:

1. mkdir socket-example

Next, change into the project directory:

1. cd socket-example

Then, create a new directory for the server code:

1. mkdir socket-server

into the server directory.

1. cd socket-server

Then, initialize a new npm project:

1. npm init -y

Now, we will install our package dependencies:

1. npm install express@4.17.1 socket.io@2.3.0 @types/socket.io@2.1.10 --save

These packages include Express, [Socket.IO](http://socket.io/), and @types/socket.io.

Now that you have completed setting up the project, you can move on to writing code for the server.

First, create a new src directory:

1. mkdir src

Now, create a new file called app.js in the src directory, and open it using your favorite text editor:

1. nano src/app.js

Start with the require statements for Express and [Socket.IO](http://socket.io/):

socket-server/src/app.js

const app = require('express')();

const http = require('http').Server(app);

const io = require('socket.io')(http);

As you can tell, we’re using Express and [Socket.IO](http://socket.io/) to set up our server. [Socket.IO](http://socket.io/) provides a layer of abstraction over native WebSockets. It comes with some nice features, such as a fallback mechanism for older browsers that do not support WebSockets, and the ability to create rooms. We’ll see this in action in a minute.

For the purposes of our real-time document collaboration application, we will need a way to store documents. In a production setting, you would want to use a database, but for the scope of this tutorial, we will use an in-memory store of documents:

socket-server/src/app.js

const documents = {};

Now, let’s define what we want our socket server to actually do:

socket-server/src/app.js

io.on("connection", socket => {

// ...

});

Let’s break this down. .on('...') is an event listener. The first parameter is the name of the event, and the second one is usually a callback executed when the event fires, with the event payload.

The first example we see is when a client connects to the socket server (connection is a reserved event type in [Socket.IO](http://socket.io/)).

We get a socket variable to pass to our callback to initiate communication to either that one socket or to multiple sockets (i.e., broadcasting).

**safeJoin**

We will set up a local function (safeJoin) that takes care of joining and leaving rooms:

socket-server/src/app.js

io.on("connection", socket => {

let previousId;

const safeJoin = currentId => {

socket.leave(previousId);

socket.join(currentId, () => console.log(`Socket ${socket.id} joined room ${currentId}`));

previousId = currentId;

};

// ...

});

In this case, when a client has joined a room, they are editing a particular document. So if multiple clients are in the same room, they are all editing the same document.

Technically, a socket can be in multiple rooms, but we don’t want to let one client edit multiple documents at the same time, so if they switch documents, we need to leave the previous room and join the new room. This little function takes care of that.

There are three event types that our socket is listening for from the client:

* getDoc
* addDoc
* editDoc

And two event types that are emitted by our socket to the client:

* document
* documents

**getDoc**

Let’s work on the first event type - getDoc:

socket-server/src/app.js

io.on("connection", socket => {

// ...

socket.on("getDoc", docId => {

safeJoin(docId);

socket.emit("document", documents[docId]);

});

// ...

});

When the client emits the getDoc event, the socket is going to take the payload (in our case, it’s just an id), join a room with that docId, and emit the stored document back to the initiating client only. That’s where socket.emit('document', ...) comes into play.

**addDoc**

Let’s work on the second event type - addDoc:

socket-server/src/app.js

io.on("connection", socket => {

// ...

socket.on("addDoc", doc => {

documents[doc.id] = doc;

safeJoin(doc.id);

io.emit("documents", Object.keys(documents));

socket.emit("document", doc);

});

// ...

});

With the addDoc event, the payload is a document object, which, at the moment, consists only of an id generated by the client. We tell our socket to join the room of that ID so that any future edits can be broadcast to anyone in the same room.

Next, we want everyone connected to our server to know that there is a new document to work with, so we broadcast to all clients with the io.emit('documents', ...) function.

Note the difference between socket.emit() and io.emit() - the socket version is for emitting back to only initiating the client, the io version is for emitting to everyone connected to our server.

**editDoc**

Let’s work on the third event type - editDoc:

socket-server/src/app.js

io.on("connection", socket => {

// ...

socket.on("editDoc", doc => {

documents[doc.id] = doc;

socket.to(doc.id).emit("document", doc);

});

// ...

});

With the editDoc event, the payload will be the whole document at its state after any keystroke. We’ll replace the existing document in the database and then broadcast the new document to only the clients that are currently viewing that document. We do this by calling socket.to(doc.id).emit(document, doc), which emits to all sockets in that particular room.

Finally, whenever a new connection is made, we broadcast to all the clients to ensure the new connection receives the latest document changes when they connect:

socket-server/src/app.js

io.on("connection", socket => {

// ...

io.emit("documents", Object.keys(documents));

console.log(`Socket ${socket.id} has connected`);

});

After the socket functions are all set up, pick a port and listen on it:

socket-server/src/app.js

http.listen(4444, () => {

console.log('Listening on port 4444');

});

Run the following command in your terminal to start the server:

node src/app.js

We now have a fully-functioning socket server for document collaboration!

## Step 2 — Installing @angular/cli and Creating the Client App

Open a new terminal window and navigate to the project directory.

Run the following commands to install the Angular CLI as a devDependency:

1. npm install @angular/cli@10.0.4 --save-dev

Now, use the @angular/cli command to create a new Angular project, with no Angular Routing and with SCSS for styling:

1. ng new socket-app --routing=false --style=scss

Then, change into the server directory:

1. cd socket-app

Now, we will install our package dependencies:

1. npm install ngx-socket-io@3.2.0 --save

ngx-socket-io is an Angular wrapper over [Socket.IO](http://socket.io/) client libraries.

Then, use the @angular/cli command to generate a document model, a document-list component, a document component, and a document service:

1. ng generate class models/document --type=model
2. ng generate component components/document-list
3. ng generate component components/document
4. ng generate service services/document

Now that you have completed setting up the project, you can move on to writing code for the client.

### App Module

Open app.modules.ts:

1. nano src/app/app.module.ts

And import FormsModule, SocketioModule, and SocketioConfig:

socket-app/src/app/app.module.ts

// ... other imports

import { FormsModule } from '@angular/forms';

import { SocketIoModule, SocketIoConfig } from 'ngx-socket-io';

And before your @NgModule declaration, define config:

socket-app/src/app/app.module.ts

const config: SocketIoConfig = { url: 'http://localhost:4444', options: {} };

You’ll notice that this is the port number that we declared earlier in the server’s app.js.

Now, add to your imports array, so it looks like:

socket-app/src/app/app.module.ts

@NgModule({

// ...

imports: [

// ...

FormsModule,

SocketIoModule.forRoot(config)

],

// ...

})

This will fire off the connection to our socket server as soon as AppModule loads.

### Document Model and Document Service

Open document.model.ts:

1. nano src/app/models/document.model.ts

And define id and doc:

socket-app/src/app/models/document.model.ts

export class Document {

id: string;

doc: string;

}

Open document.service.ts:

1. nano src/app/services/document.service.ts

**And add the following in the class definition:**

socket-app/src/app/services/document.service.ts

import { Injectable } from '@angular/core';

import { Socket } from 'ngx-socket-io';

import { Document } from 'src/app/models/document.model';

@Injectable({

providedIn: 'root'

})

export class DocumentService {

currentDocument = this.socket.fromEvent<Document>('document');

documents = this.socket.fromEvent<string[]>('documents');

constructor(private socket: Socket) { }

getDocument(id: string) {

this.socket.emit('getDoc', id);

}

newDocument() {

this.socket.emit('addDoc', { id: this.docId(), doc: '' });

}

editDocument(document: Document) {

this.socket.emit('editDoc', document);

}

private docId() {

let text = '';

const possible = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789';

for (let i = 0; i < 5; i++) {

text += possible.charAt(Math.floor(Math.random() \* possible.length));

}

return text;

}

}

The methods here represent each emit the three event types that the socket server is listening for. The properties currentDocument and documents represent the events emitted by the socket server, which is consumed on the client as an Observable.

You may notice a call to this.docId(). This is a little private method that generates a random string to assign as the document id.

### Document List Component

Let’s put the list of documents in a sidenav. Right now, it’s only showing the docId - a random string of characters.

Open document-list.component.html:

1. nano src/app/components/document-list/document-list.component.html

And replace the contents with the following:

socket-app/src/app/components/document-list/document-list.component.html

<div class='sidenav'>

<span

(click)='newDoc()'

>

New Document

</span>

<span

[class.selected]='docId === currentDoc'

(click)='loadDoc(docId)'

\*ngFor='let docId of documents | async'

>

{{ docId }}

</span>

</div>

Open document-list.component.scss:

1. nano src/app/components/document-list/document-list.component.scss

And add some styles:

socket-app/src/app/components/document-list/document-list.component.scss

.sidenav {

background-color: #111111;

height: 100%;

left: 0;

overflow-x: hidden;

padding-top: 20px;

position: fixed;

top: 0;

width: 220px;

span {

color: #818181;

display: block;

font-family: 'Roboto', Tahoma, Geneva, Verdana, sans-serif;

font-size: 25px;

padding: 6px 8px 6px 16px;

text-decoration: none;

**&**.selected {

color: #e1e1e1;

}

**&**:hover {

color: #f1f1f1;

cursor: pointer;

}

}

}

Open document-list.component.ts:

1. nano src/app/components/document-list/document-list.component.ts

And add the following in the class definition:

socket-app/src/app/components/document-list/document-list.component.ts

import { Component, OnInit, OnDestroy } from '@angular/core';

import { Observable, Subscription } from 'rxjs';

import { DocumentService } from 'src/app/services/document.service';

@Component({

selector: 'app-document-list',

templateUrl: './document-list.component.html',

styleUrls: ['./document-list.component.scss']

})

export class DocumentListComponent implements OnInit, OnDestroy {

documents: Observable<string[]>;

currentDoc: string;

private \_docSub: Subscription;

constructor(private documentService: DocumentService) { }

ngOnInit() {

this.documents = this.documentService.documents;

this.\_docSub = this.documentService.currentDocument.subscribe(doc => this.currentDoc = doc.id);

}

ngOnDestroy() {

this.\_docSub.unsubscribe();

}

loadDoc(id: string) {

this.documentService.getDocument(id);

}

newDoc() {

this.documentService.newDocument();

}

}

Let’s start with the properties. documents will be a stream of all available documents. currentDocId is the id of the currently selected document. The document list needs to know what document we’re on, so we can highlight that doc id in the sidenav. \_docSub is a reference to the Subscription that gives us the current or selected doc. We need this so we can unsubscribe in the ngOnDestroy lifecycle method.

You’ll notice the methods loadDoc() and newDoc() don’t return or assign anything. Remember, these fire off events to the socket server, which turns around and fires an event back to our Observables. The returned values for getting an existing document or adding a new document are realized from the Observable patterns above.

### Document Component

This will be the document editing surface.

Open document.component.html:

1. nano src/app/components/document/document.component.html

And replace the contents with the following:

socket-app/src/app/components/document/document.component.html

<textarea

[(ngModel)]='document.doc'

(keyup)='editDoc()'

placeholder='Start typing...'

></textarea>

Open document.component.scss:

1. nano src/app/components/document/document.component.scss

And change some styles on the default HTML textarea:

socket-app/src/app/components/document/document.component.scss

textarea {

border: none;

font-size: 18pt;

height: 100%;

padding: 20px 0 20px 15px;

position: fixed;

resize: none;

right: 0;

top: 0;

width: calc(100% - 235px);

}

Open document.component.ts:

1. src/app/components/document/document.component.ts

And add the following in the class definition:

socket-app/src/app/components/document/document.component.ts

import { Component, OnInit, OnDestroy } from '@angular/core';

import { Subscription } from 'rxjs';

import { startWith } from 'rxjs/operators';

import { Document } from 'src/app/models/document.model';

import { DocumentService } from 'src/app/services/document.service';

@Component({

selector: 'app-document',

templateUrl: './document.component.html',

styleUrls: ['./document.component.scss']

})

export class DocumentComponent implements OnInit, OnDestroy {

document: Document;

private \_docSub: Subscription;

constructor(private documentService: DocumentService) { }

ngOnInit() {

this.\_docSub = this.documentService.currentDocument.pipe(

startWith({ id: '', doc: 'Select an existing document or create a new one to get started' })

).subscribe(document => this.document = document);

}

ngOnDestroy() {

this.\_docSub.unsubscribe();

}

editDoc() {

this.documentService.editDocument(this.document);

}

}

Similar to the pattern we used in the DocumentListComponent above, we’re going to subscribe to the changes for our current document, and fire off an event to the socket server whenever we change the current document. This means that we will see all the changes if any other client is editing the same document we are, and vice versa. We use the RxJS startWith operator to give a little message to our users when they first open the app.

### AppComponent

Open app.component.html:

1. nano src/app.component.html

And compose the two custom components by replacing the contents with the following:

socket-app/src/app.component.html

<app-document-list></app-document-list>

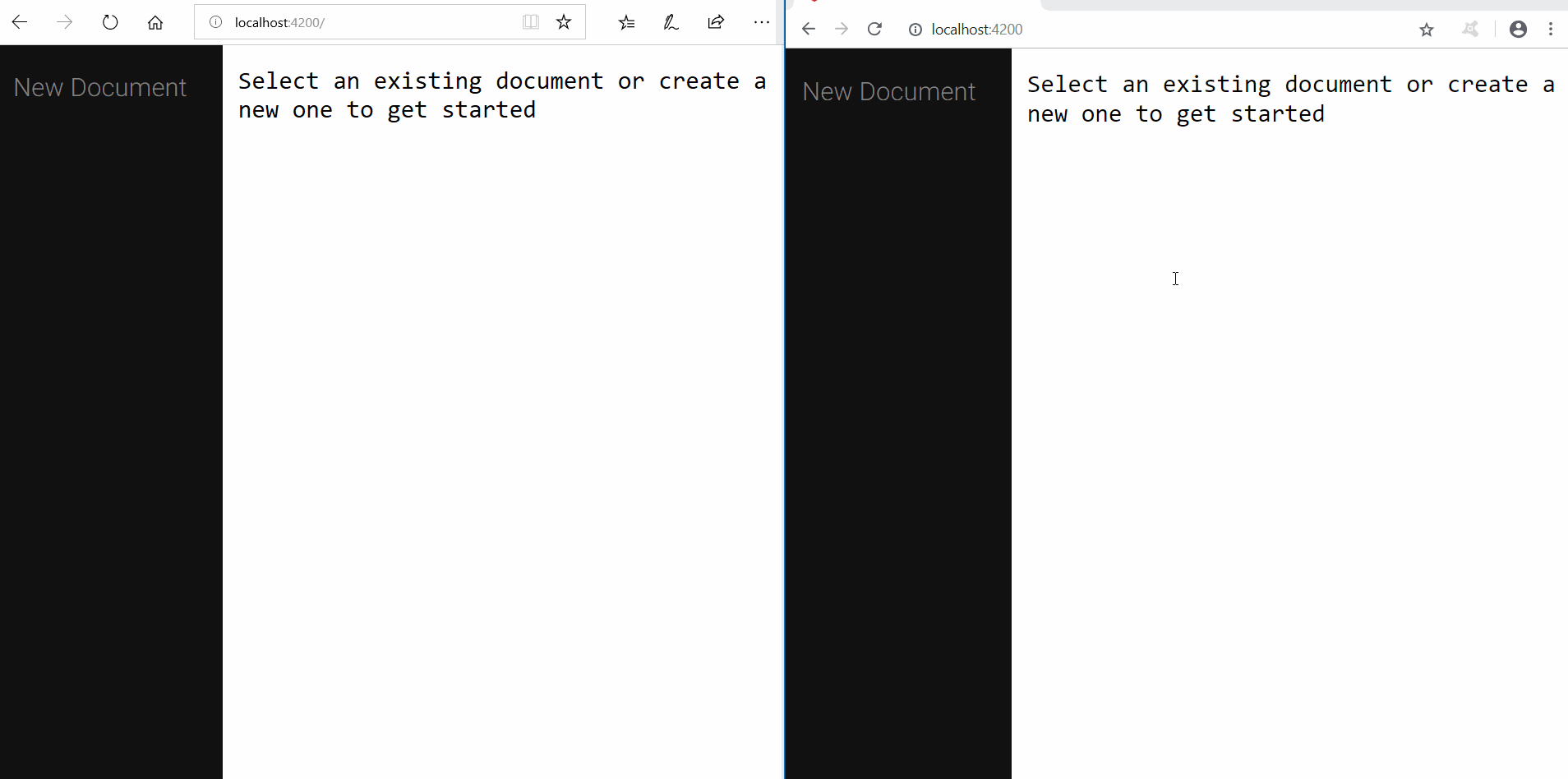
<app-document></app-document>

## Step 3 — Viewing the App in Action

With our socket server still running in a terminal window, let’s open a new terminal window and start our Angular app:

1. ng serve

Open more than one instance of http://localhost:4200 in separate browser tabs and see it in action.



Now, you can create new documents and see them update in both browser windows. You can make a change in one browser window and see the change reflected in the other browser window.

## Conclusion

In this tutorial, you have completed an initial exploration into using WebSocket. You used it to build a real-time document collaboration application. It supports multiple browser sessions to connect to a server and update and modify multiple documents.

If you’d like to learn more about Angular, check out [our Angular topic page](https://www.digitalocean.com/community/tags/angularjs) for exercises and programming projects.

If you’d like to learn more about [Socket.IO](http://socket.io/), check out [Integrating Vue.js and Socket.IO](https://www.digitalocean.com/community/tutorials/vuejs-vue-socketio).

Further WebSocket projects include real-time chat applications. See [How To Build a Realtime Chat App with React and GraphQL](https://www.digitalocean.com/community/tutorials/how-to-build-a-realtime-chat-app-with-react-and-graphql).

# Node.js Web Server

In this section, we will learn how to create a simple Node.js web server and handle HTTP requests.

To access web pages of any web application, you need a [web server](https://en.wikipedia.org/wiki/Web_server). The web server will handle all the http requests for the web application e.g IIS is a web server for ASP.NET web applications and Apache is a web server for PHP or Java web applications.

Node.js provides capabilities to create your own web server which will handle HTTP requests asynchronously. You can use IIS or Apache to run Node.js web application but it is recommended to use Node.js web server.

## Create Node.js Web Server

Node.js makes it easy to create a simple web server that processes incoming requests asynchronously.

The following example is a simple Node.js web server contained in server.js file.

**server.js**

var http = require('http'); // 1 - Import Node.js core module

var server = http.createServer(function (req, res) { // 2 - creating server

//handle incomming requests here..

});

server.listen(5000); //3 - listen for any incoming requests

console.log('Node.js web server at port 5000 is running..')

In the above example, we import the http module using require() function. The http module is a core module of Node.js, so no need to install it using NPM. The next step is to call createServer() method of http and specify callback function with request and response parameter. Finally, call listen() method of server object which was returned from createServer() method with port number, to start listening to incoming requests on port 5000. You can specify any unused port here.

Run the above web server by writing node server.js command in command prompt or terminal window and it will display message as shown below.

C:\> node server.js  
Node.js web server at port 5000 is running..

This is how you create a Node.js web server using simple steps. Now, let's see how to handle HTTP request and send response in Node.js web server.

## Handle HTTP Request

The http.createServer() method includes [request](https://nodejs.org/api/http.html#http_http_incomingmessage) and [response](https://nodejs.org/api/http.html#http_class_http_serverresponse) parameters which is supplied by Node.js. The request object can be used to get information about the current HTTP request e.g., url, request header, and data. The response object can be used to send a response for a current HTTP request.

The following example demonstrates handling HTTP request and response in Node.js.

**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..')

In the above example, req.url is used to check the url of the current request and based on that it sends the response. To send a response, first it sets the response header using writeHead() method and then writes a string as a response body using write() method. Finally, Node.js web server sends the response using end() method.

Now, run the above web server as shown below.

C:\> node server.js  
Node.js web server at port 5000 is running..

To test it, you can use the command-line program curl, which most Mac and Linux machines have pre-installed.

curl -i http://localhost:5000

You should see the following response.

HTTP/1.1 200 OK  
Content-Type: text/plain  
Date: Tue, 8 Sep 2015 03:05:08 GMT  
Connection: keep-alive  
This is home page.

For Windows users, point your browser to *http://localhost:5000* and see the following result.

[Graphical user interface, text, application, Word

Description automatically generated](https://www.tutorialsteacher.com/Content/images/nodejs/webserver-response.png)Node.js Web Server Response

The same way, point your browser to *http://localhost:5000/student* and see the following result.

[Graphical user interface, text, application

Description automatically generated](https://www.tutorialsteacher.com/Content/images/nodejs/webserver-response2.png)Node.js Web Server Response

It will display "Invalid Request" for all requests other than the above URLs.

## Sending JSON Response

The following example demonstrates how to serve JSON response from the Node.js web server.

**server.js**

var http = require('http');

var server = http.createServer(function (req, res) {

if (req.url == '/data') { //check the URL of the current request

res.writeHead(200, { 'Content-Type': 'application/json' });

res.write(JSON.stringify({ message: "Hello World"}));

res.end();

}

});

server.listen(5000);

console.log('Node.js web server at port 5000 is running..')

So, this way you can create a simple web server that serves different responses.