# Angular & NodeJS - The MEAN Stack Guide [2021 Edition]

## Getting Started

### Intro

### Intro

### Intro

### What is a Single Page Application (SPA)?

MEAN provides a dynamic single-page app. It appears to have different pages but this all do to the front end scripting modifying the DOM.

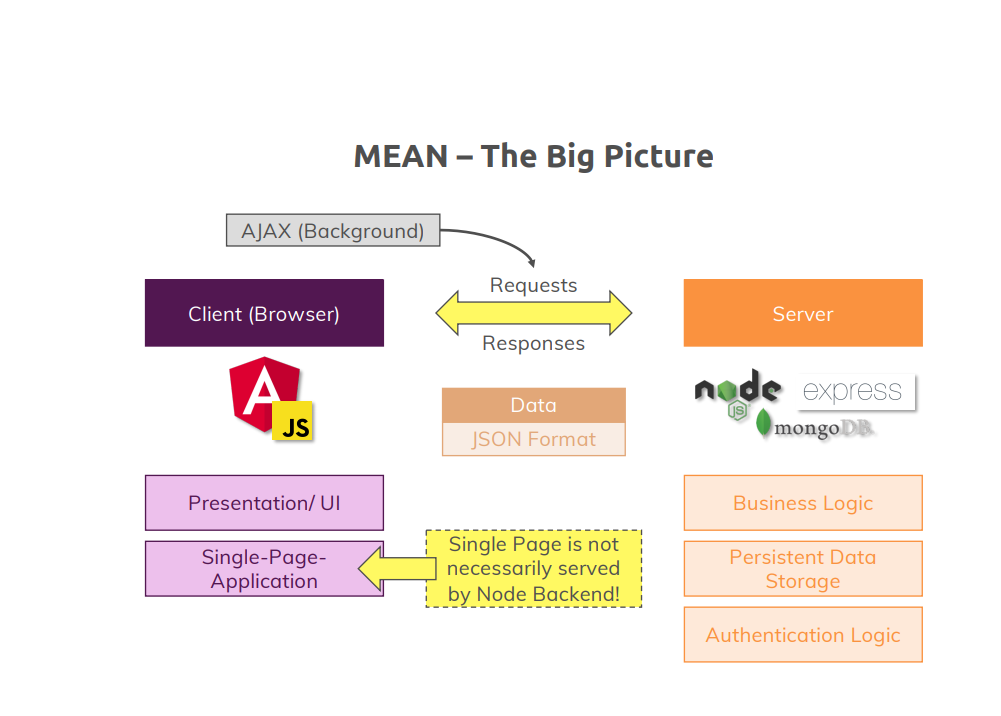
### How Does a MEAN Stack work?

**Mongo**: non-SQL database

**Express**: AJAX is used and data is transmitted using JSON format.

**Angular**: front end. Handles different screens: PC, tablet, and mobile. Made by **Google**.

**Node**: JavaScript runtime. Provides access to server.



Also uses **RxJS**.

|  |  |
| --- | --- |
| ng | This is the angular CLI |

### Must read notes



### Installing Node and Angular CLI

Download and install node (see other notes)

Install Angular (see Angular course)

cd ~

mkdir project-mean

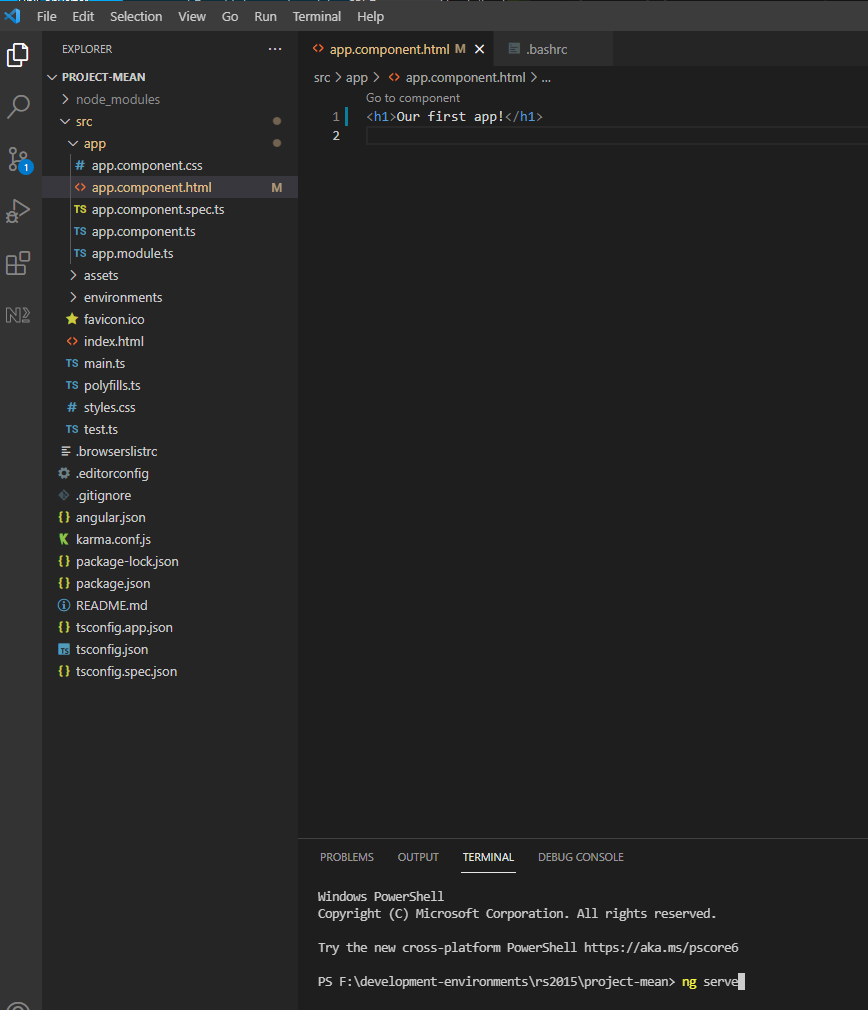
cd project-mean

ng new --no-strict first-project

cd first-project/

click on Terminal->New

then type ng serve



Open [**http://localhost:**4200/](http://localhost:4200/) in chrome

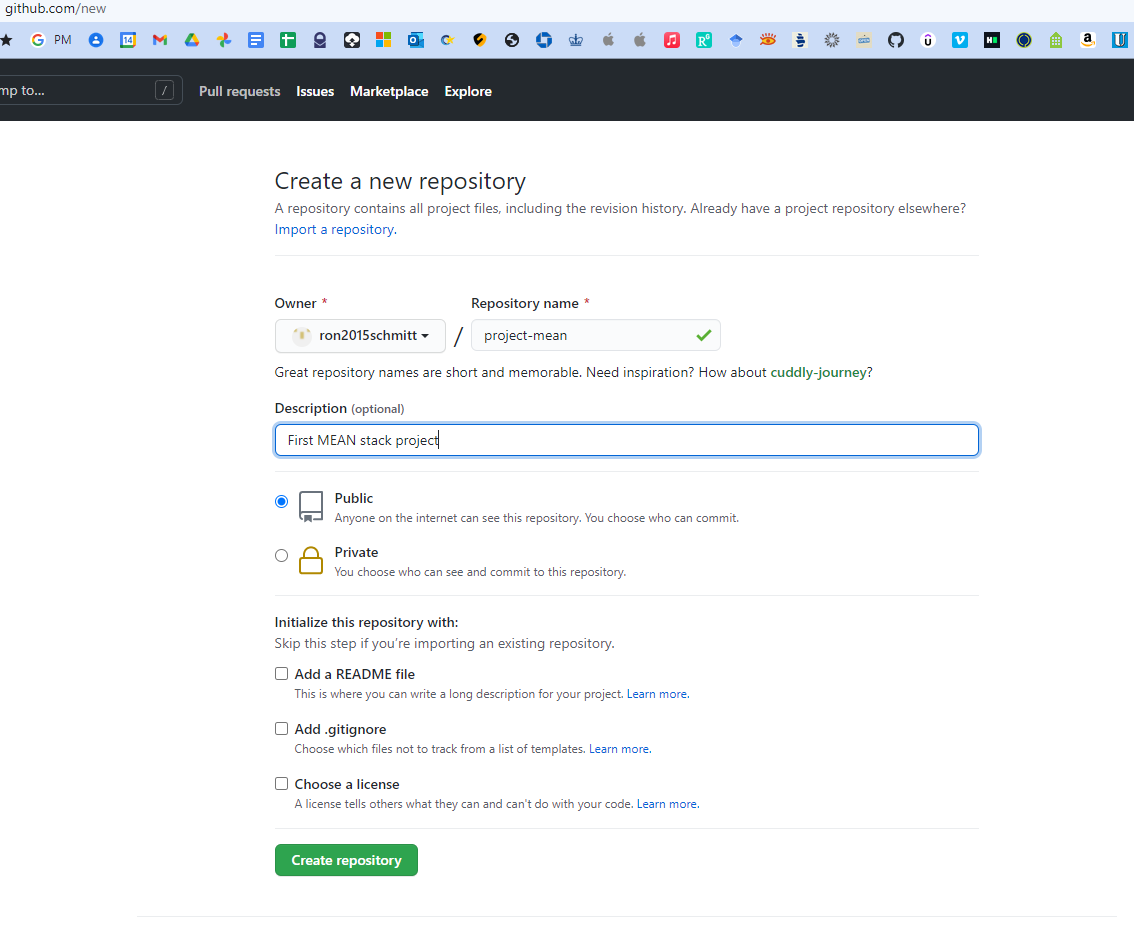
The following worked for github but probably is not best way to do it.

cd ~/project-mean

git init

git add .

create new repo on github

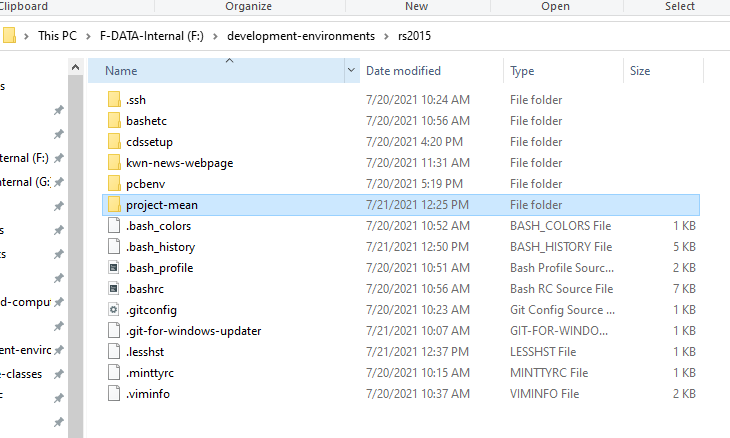


git branch -M main

git remote add origin https://github.com/ron2015schmitt/project-mean.git

git push -u origin main

Right-click folder and open in VSCode



### Installing Our IDE

Install VSCode (see other notes)

Install VSCode extensions:

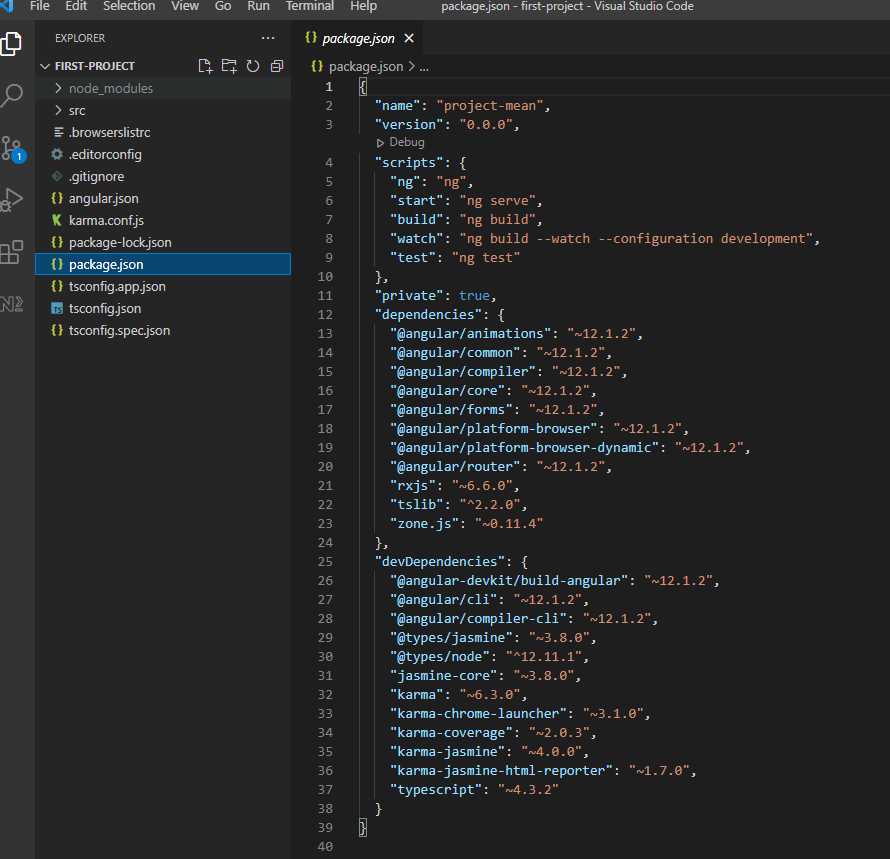
* Angular Essentials
* Material Icon Theme

### Exploring the Project Structure

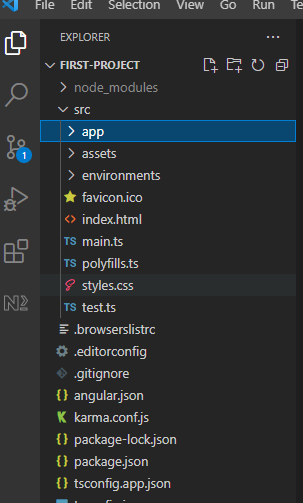
Completed through Chap 8

Open project-mean folder in VSCode

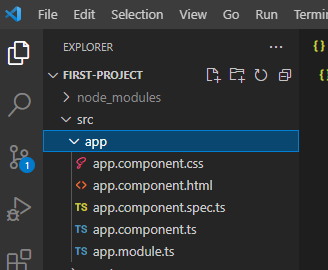
The package.json file list all the node packages needed, our dependencies. When you type npm install these package downloaded and built in the node\_modules folder



Our source code resides in the src folder



The main Angular component is defined in the app folder



### Course Outline

Data Pagination is the process of downloading large amounts data from the database as needed. Example is a grid with thousands of rows.

### How to get the most out of this course

Code along with the video

Ask questions on Q&A

### section resources

## The Angular Front End

### Angular: Introduction

We’re going to build a mini social network

### Understanding the folder structure

See Chap 7 of the Angular course

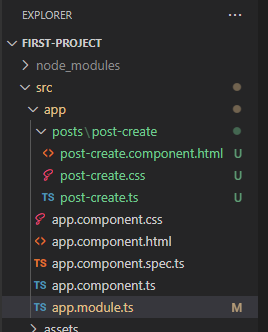
### Understanding Angular Components

Angular constructs web page via components.

Similar to a word processor, the page layout itself is divided up into regions that are defined by components.

### Adding Our Component

Create folder and file structure highlighted below in green text



Leave the CSS blank.

|  |
| --- |
| **post-create.component.html** |
| <p>post-create</p> |

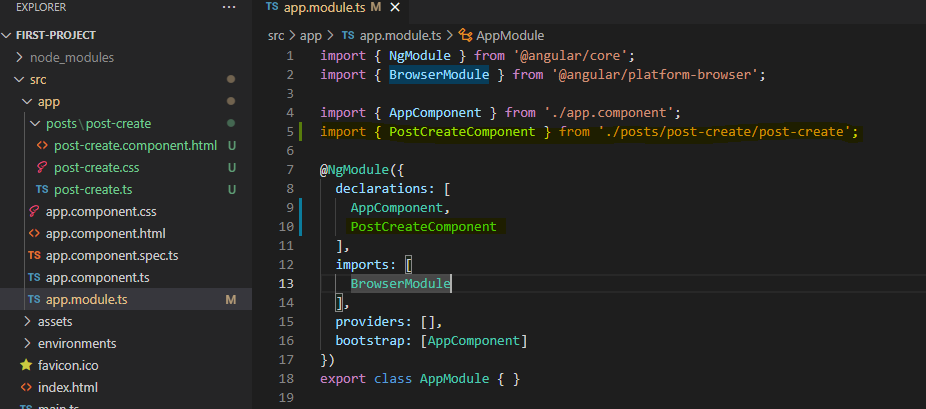
|  |
| --- |
| **post-create.ts** |
| import { Component } from "@angular/core";  @Component({    selector: 'app-post-create',    templateUrl: './post-create.component.html'  })  export class PostCreateComponent {  } |

Add the following line to app.module.ts

import { PostCreateComponent } from './posts/post-create/post-create';

Then also add PostCreateComponent to the declarations

result:



Now in the app html file, write

|  |
| --- |
| **app.component.html** |
| <h1>Our First App!</h1>  <app-post-create></app-post-create> |

|  |
| --- |
| **Our App in Chrome** |
|  |

### Listening to Events

Rename post-create.ts to post-create.component.ts including all its references

Add function onAddPost to definition

|  |
| --- |
| **post-create.component.ts** |
| import { Component } from "@angular/core";  @Component({    selector: 'app-post-create',    templateUrl: './post-create.component.html'  })  export class PostCreateComponent {    onAddPost() {      alert('Post Added!');    }  } |

Change html to have a text area and a button tied to that function

|  |
| --- |
| **post-create.component.html** |
| <textarea rows="6"></textarea>  <hr>  <button (click)="onAddPost()">Save Button</button> |

|  |
| --- |
| **Our App in Chrome** |
|  |

### Outputting the content

Add function onAddPost to definition

|  |
| --- |
| **post-create.component.ts** |
| import { Component } from "@angular/core";  @Component({    selector: 'app-post-create',    templateUrl: './post-create.component.html'  })  export class PostCreateComponent {    onAddPost() {      alert('Post Added!');    }  } |

Change html to have a text area and a button tied to that function, with initial value as shown.

|  |
| --- |
| **post-create.component.html** |
| <textarea rows="6" value="hello"></textarea>  <hr>  <button (click)="onAddPost()">Save Button</button> |

|  |
| --- |
| **Our App in Chrome** |
|  |

Now, let’s use Angular in the **textarea**: We use [value] and "'hello'" for the value.

|  |
| --- |
| **post-create.component.html** |
| <textarea rows="6" [value]="'hello'"></textarea>  <hr>  <button (click)="onAddPost()">Save Button</button> |

Lastly, we put a variable inside the method

|  |
| --- |
| **post-create.component.ts** |
| import { Component } from "@angular/core";  @Component({    selector: 'app-post-create',    templateUrl: './post-create.component.html'  })  export class PostCreateComponent {    newPost = 'type here';    onAddPost() {    }  } |

Change html to have a text area and a button tied to that function.

|  |
| --- |
| **post-create.component.html** |
| <textarea rows="6" [value]="newPost"></textarea>  <hr>  <button (click)="onAddPost()">Save Button</button> |

|  |
| --- |
| **Our App in Chrome** |
|  |

### Getting the user input

First version: use 2 one-way bindings.

We bind the **textarea** value to **#postInput** and then feed to the onAddPost method

|  |
| --- |
| **post-create.component.html** |
| <textarea rows="6" [value]="newPost" #postInput></textarea>  <hr>  <button (click)="onAddPost(postInput)">Save Button</button>  <p>{{ newPost }}</p> |

|  |
| --- |
| **post-create.component.ts** |
| import { Component } from "@angular/core";  @Component({    selector: 'app-post-create',    templateUrl: './post-create.component.html'  })  export class PostCreateComponent {    newPost = 'type here';    onAddPost(postInput: HTMLTextAreaElement) {      console.log(postInput);      this.newPost = postInput.value;    }  } |

|  |
| --- |
| **Our App in Chrome** |
|  |

Type and hit save and the value at the bottom now updates

Second version: use two-way binding.

Add import line for NgModule and add to imports array to **app.module.ts**

|  |
| --- |
| **app.module.ts** |
| import { NgModule } from '@angular/core';  import { BrowserModule } from '@angular/platform-browser';  import { FormsModule } from '@angular/forms';  import { AppComponent } from './app.component';  import { PostCreateComponent } from './posts/post-create/post-create.component';  @NgModule({    declarations: [      AppComponent,      PostCreateComponent    ],    imports: [      BrowserModule,      FormsModule    ],    providers: [],    bootstrap: [AppComponent]  })  export class AppModule { } |

Use the forms modules. Note this **updates the variable for every key stroke** not just when hitting the button

|  |
| --- |
| **post-create.component.html** |
| <textarea rows="6" [(ngModel)]="userValue"></textarea>  <hr>  <button (click)="onAddPost()">Save Button</button>  <p>{{ newPost }}</p> |

|  |
| --- |
| **post-create.component.ts** |
| import { Component } from "@angular/core";  @Component({    selector: 'app-post-create',    templateUrl: './post-create.component.html'  })  export class PostCreateComponent {    newPost = 'NO CONTENT';    userValue = '';    onAddPost() {      console.log(this.userValue);      this.newPost = this.userValue;    }  } |

|  |
| --- |
| **Our App in Chrome** |
|  |

### Installing Angular Material

We will use [Angular Material](Angular%20Material) instead of Bootstrap

Add the material library

|  |
| --- |
| ng add @angular/material |

Go to [Form Field](https://material.angular.io/components/form-field/overview) controls

Add imports

|  |
| --- |
| <https://angular.io/api/core/NgModule> |
|  |
|  |
| These components must be used somewhere or else you get a runtime error |

|  |
| --- |
| app.module.ts |
|  |

Use the forms modules. Note this **updates the variable for every key stroke** not just when hitting the button

|  |
| --- |
| post-create.component.ts |
|  |

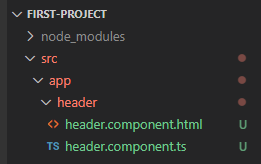
|  |
| --- |
| post-create.component.css |
|  |

|  |
| --- |
| post-create.component.html |
|  |

|  |
| --- |
| Our App in Chrome |
|  |

### Adding a Toolbar

Add folder and files



Use the forms modules. Note this **updates the variable for every key stroke** not just when hitting the button

|  |
| --- |
| header.component.ts |
|  |

|  |
| --- |
| header.component.html |
|  |

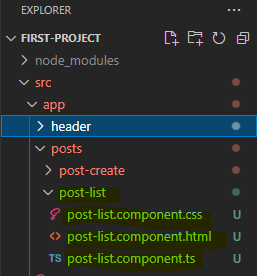
|  |
| --- |
| app.module.ts |
|  |

|  |
| --- |
| app.module.html |
|  |

|  |
| --- |
| Our App in Chrome |
|  |

### Outputting Posts (post-list component)

Add folder and files



Use the forms modules. Note this **updates the variable for every key stroke** not just when hitting the button

|  |
| --- |
| post-list.component.ts |
|  |

We will now user an expansion panel. Copy example code from the following link

<https://material.angular.io/components/expansion/overview>

|  |
| --- |
| post-list.component.html |
|  |

|  |
| --- |
| post-list.component.css |
|  |

|  |
| --- |
| app.module.ts |
|  |

|  |
| --- |
| app.module.html |
|  |

|  |
| --- |
| app.module.css |
|  |

|  |
| --- |
| Our App in Chrome |
|  |

### Diving Into Structural Directives

|  |
| --- |
| post-list.component.ts |
|  |

|  |
| --- |
| post-list.component.html |
|  |

|  |
| --- |
| post-list.component.css |
|  |

|  |
| --- |
| Our App in Chrome |
|  |

### Creating Posts with Property & Event Binding

|  |
| --- |
| app.component.ts |
|  |

|  |
| --- |
| app.component.html |
|  |

|  |
| --- |
| post-list.component.ts |
|  |

|  |
| --- |
| post-create.component.ts |
|  |

|  |
| --- |
| post-create.component.html |
|  |

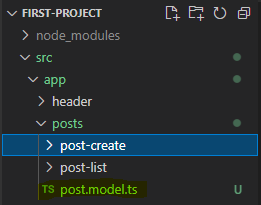
|  |
| --- |
| post-create.component.css |
|  |

|  |
| --- |
| Our App in Chrome |
|  |

### Creating a Post Model

Here we create a data type for a Post

Create new file as shown



|  |
| --- |
| post.model.ts |
|  |

|  |
| --- |
| app.component.ts |
|  |

|  |
| --- |
| post-list.component.ts |
|  |

|  |
| --- |
| post-create.component.ts |
|  |

### Adding Forms

Use [Forms](https://material.angular.io/components/form-field/overview) objects instead of two-way binding. Scroll down error messages to get code for error handling.

|  |
| --- |
| post-create.component.ts |
|  |

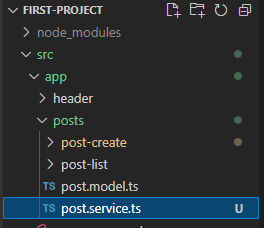
|  |
| --- |
| post-create.component.html |
|  |

|  |
| --- |
| Our App in Chrome |
|  |

### Getting Posts from Post-Create to Post-List

Here we create a PostsService class and use dependency injection, @Injectable, to access the posts.

Create new file as shown



The use of [...this.posts] create a copy of the array.

The use of @Injectable({ providedIn: 'root' }) makes this available in the @NgModule.providers array in file app.module.ts. In other words, this is shorthand for

import { PostsService } from './posts/posts.service';

@NgModule({

  declarations: [

    HeaderComponent,

    AppComponent,

    PostCreateComponent,

    PostListComponent

  ],

  imports: [

    BrowserModule,

    FormsModule,

    BrowserAnimationsModule,

    MatInputModule,

    MatCardModule,

    MatButtonModule,

    MatDividerModule,

    MatToolbarModule,

    MatExpansionModule,

  ],

  providers: [PostsService],

  bootstrap: [AppComponent]

})

export class AppModule { }

With this code, an instance of PostsService is created at init and is fed to every class constructor that has a PostsService argument.

|  |
| --- |
| posts.service.ts |
|  |

And lastly, use of public

  constructor(public postsService: PostsService) {

  }

is shorthand for

postsService: PostsService;

  constructor(postsService: PostsService) {

this.postsService = postsService;

  }

|  |
| --- |
| posts.service.ts |
|  |

|  |
| --- |
| post-create.component.css |
|  |

### RxJS: Calling GET Post

Here we use **Observables** using rxjs to manage posts.

Can remove all the Angular code from the app html!

|  |
| --- |
| app.component.html |
|  |

Implement a Subject observable in the posts service.

|  |
| --- |
| posts.service.ts |
|  |

Now subscribe to the PostsService, as well as implementing the OnInit interface.

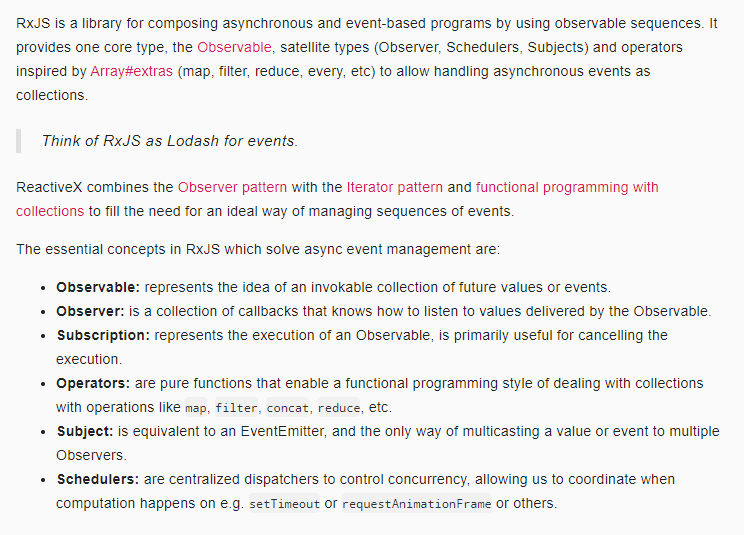
|  |
| --- |
| post-list.component.ts |
|  |

Now use PostsService in the create code.

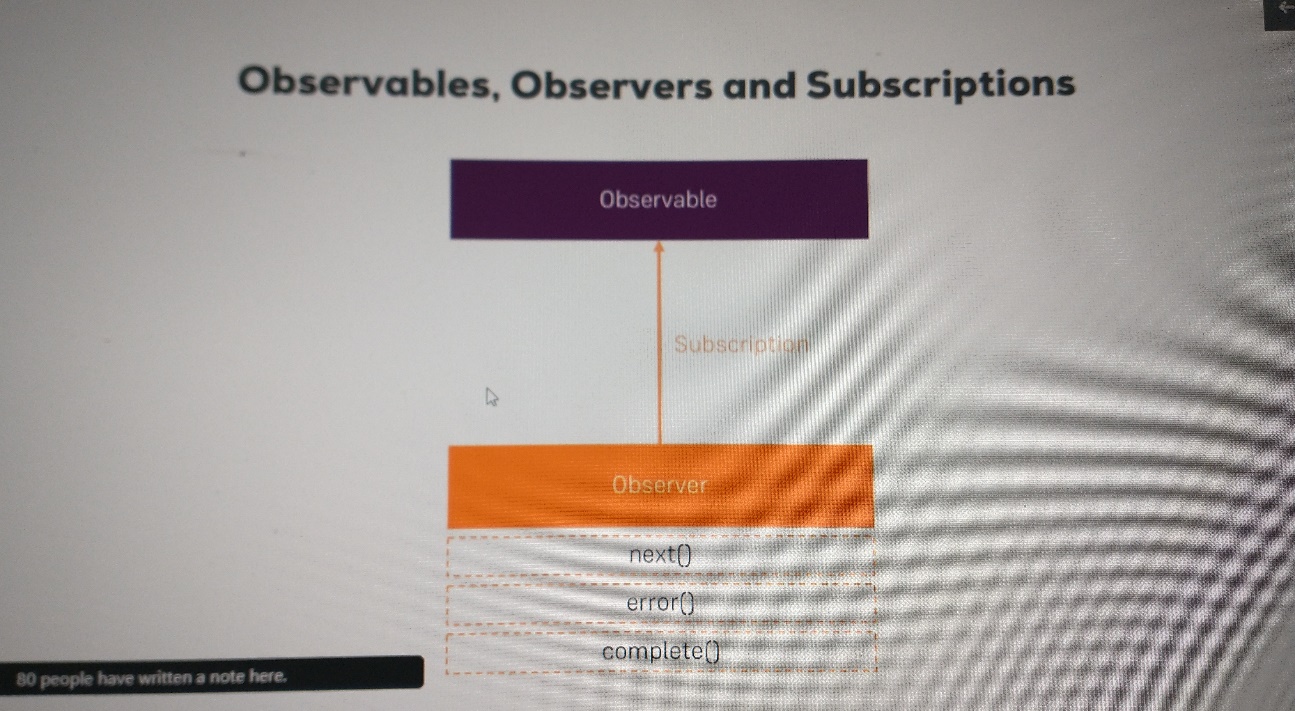
|  |
| --- |
| post-create.component.ts |
|  |

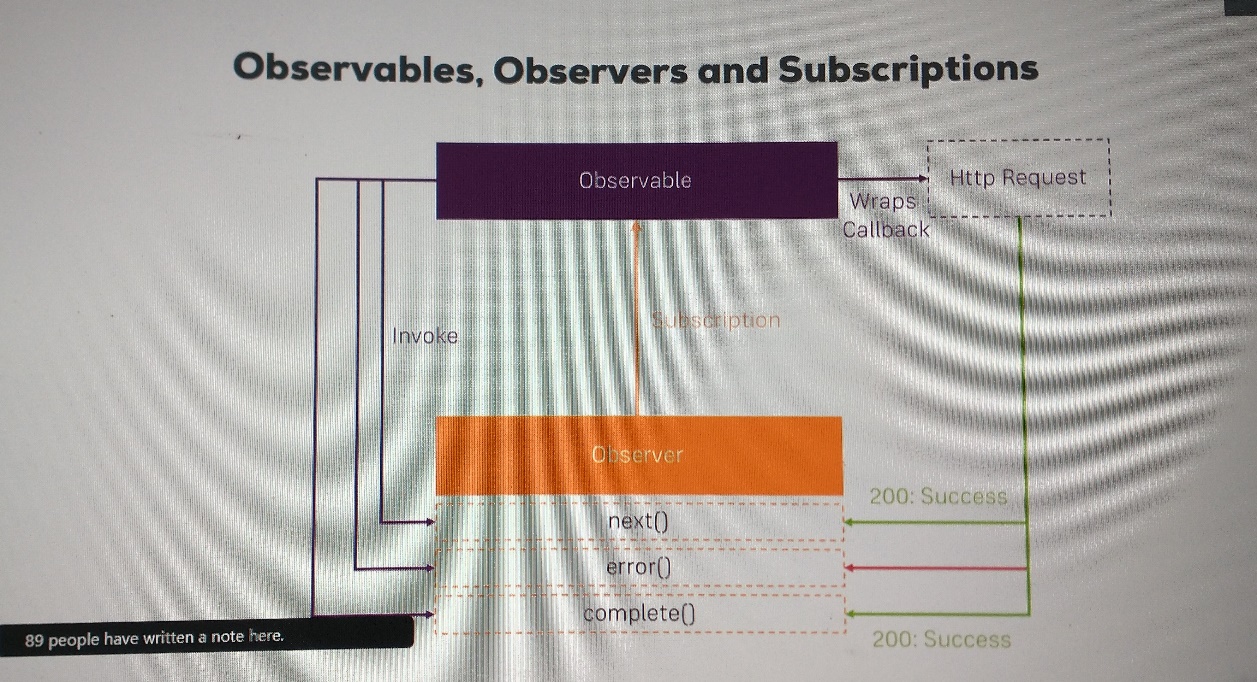
### RxJS: More About Observables

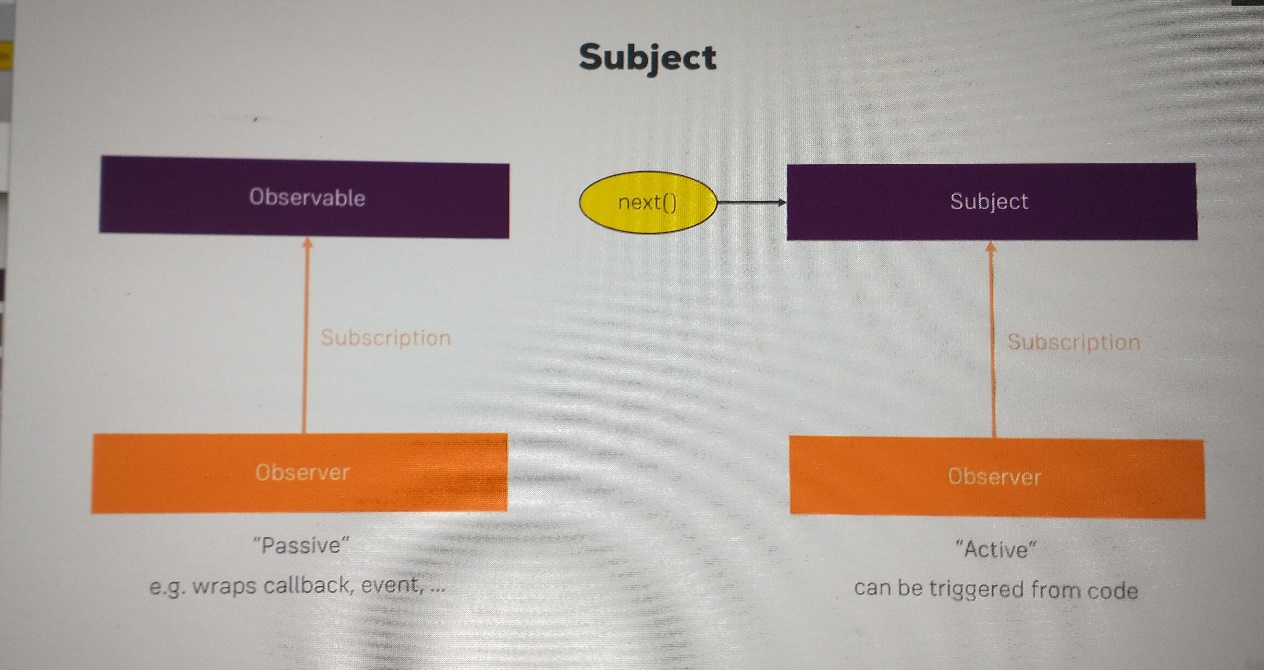
We use RxJS

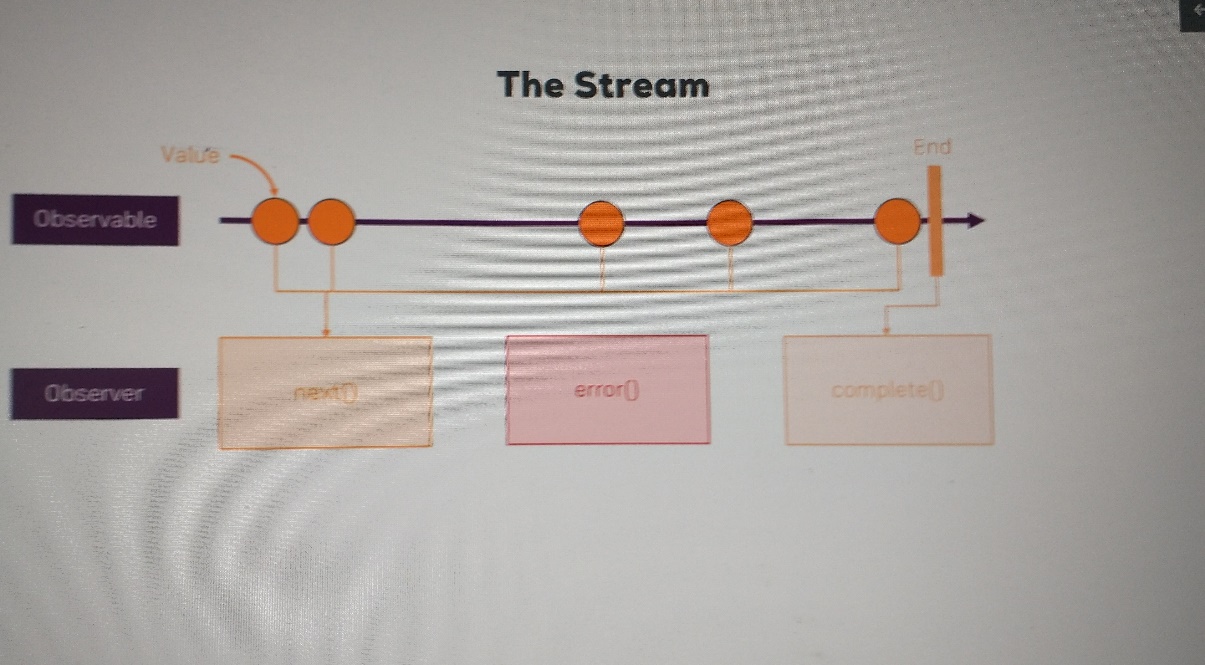


<https://www.youtube.com/watch?v=T9wOu11uU6U>









### Working On Our Form

Add labels to the form using placeholder in html.

|  |
| --- |
| post-create.component.html |
|  |

Add buttons for Delete and Edit in the html for each post in list.

|  |
| --- |
| post-list.component.html |
|  |

Clear form after Submit is pressed using form.resetForm().

|  |
| --- |
| post-create.component.ts |
|  |

|  |
| --- |
| Our App in Chrome |
|  |

### Angular Section Resources

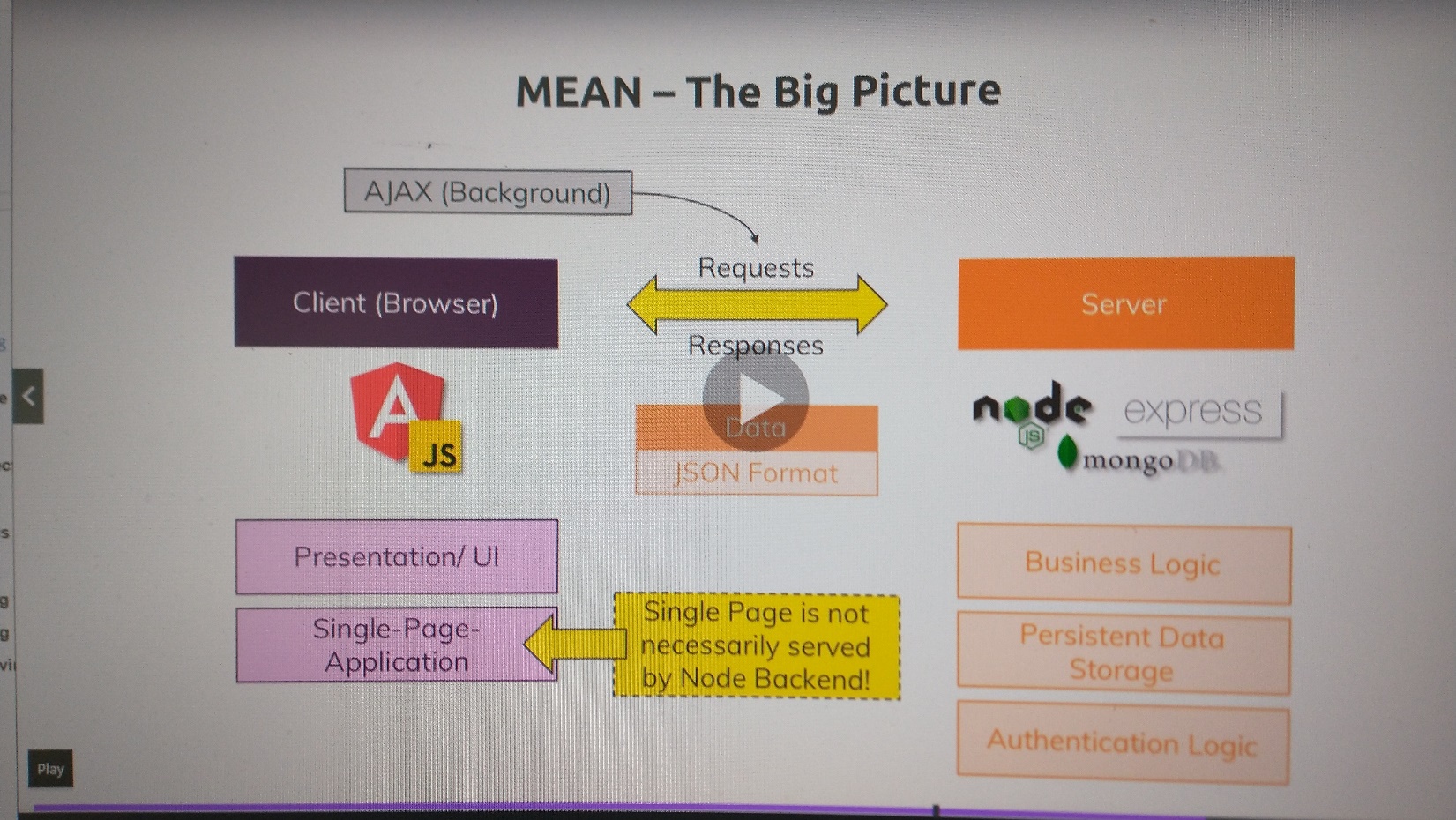
Attached to this lecture, you find code snapshots for the course section. In addition, this links might be helpful as well:

* Learn everything about Angular: <https://academind.com/learn/angular>
* Angular Material Tutorial: <https://academind.com/learn/angular/angular-material-a-thorough-guide/>
* Angular Material Docs: <https://material.angular.io/>
* Reference vs Primitive Types in JS: <https://academind.com/learn/javascript/reference-vs-primitive-values/>
* RxJS Tutorial: <https://academind.com/learn/javascript/understanding-rxjs/>

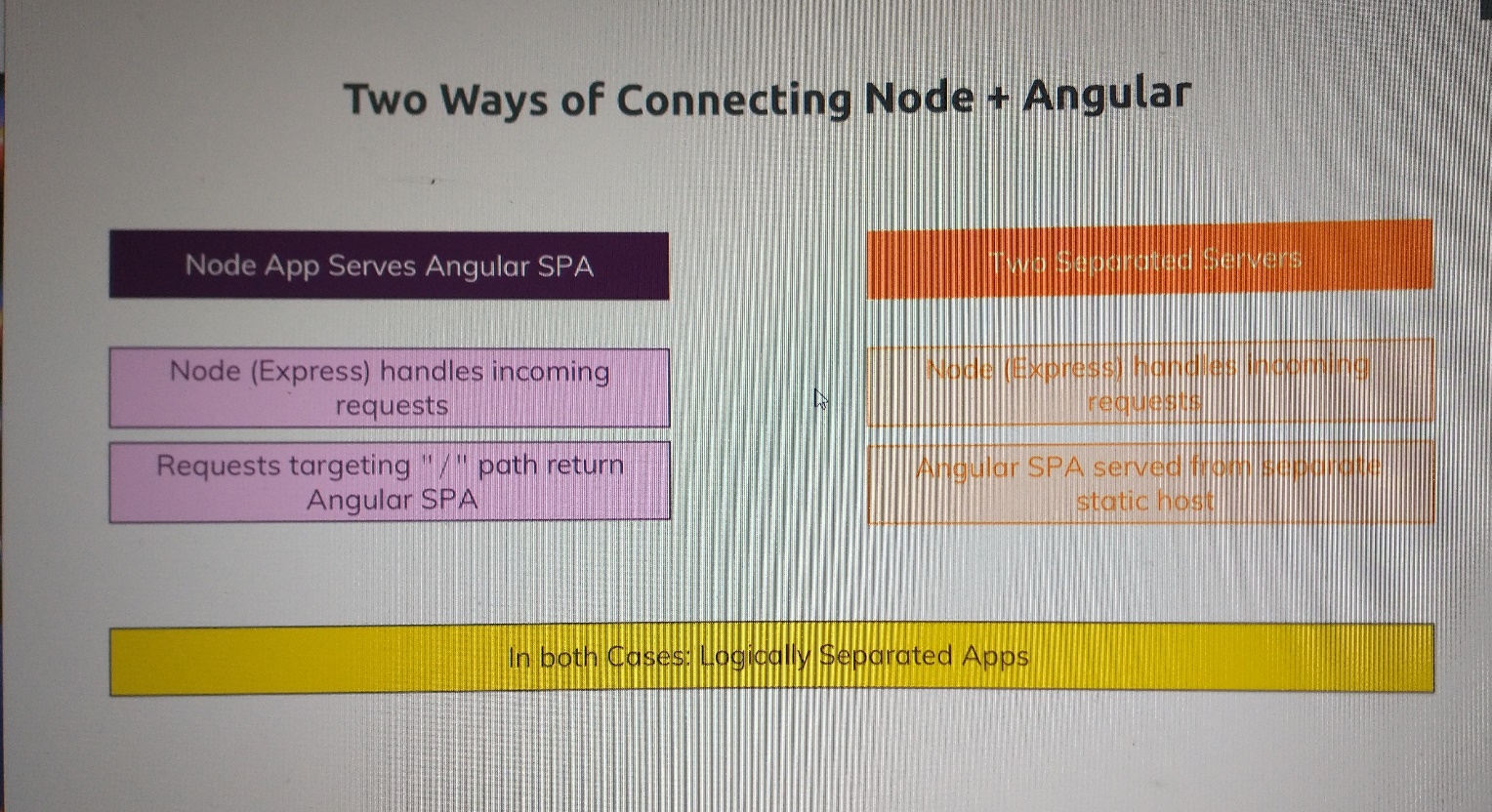
[RxJS: Reactive Extensions Library for JavaScript](https://rxjs.dev/)

## Adding NodeJS and Express to Our Project

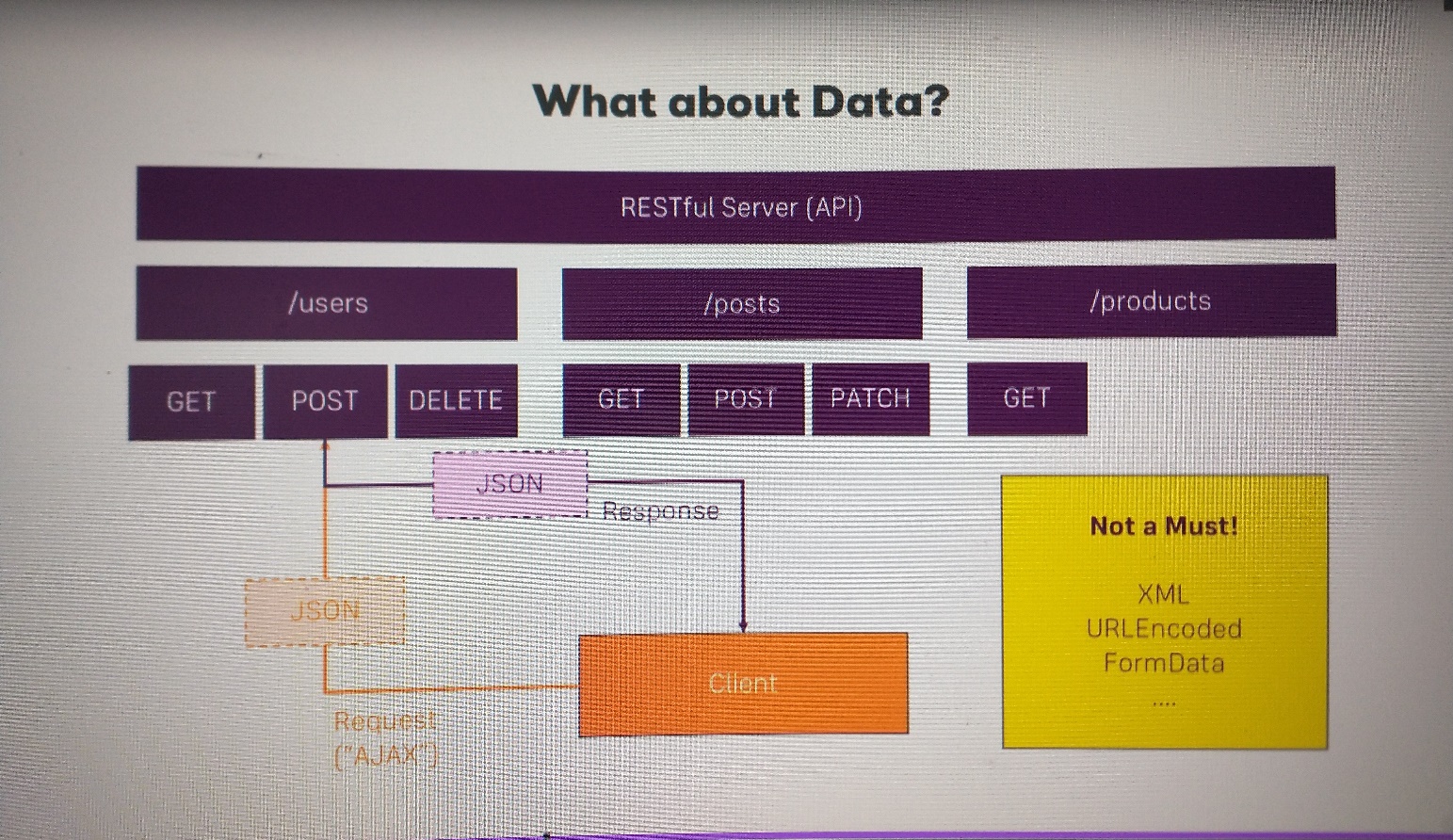
### Module Introduction



### Connecting Node & Angular - Theory



### What is a RESTful API?

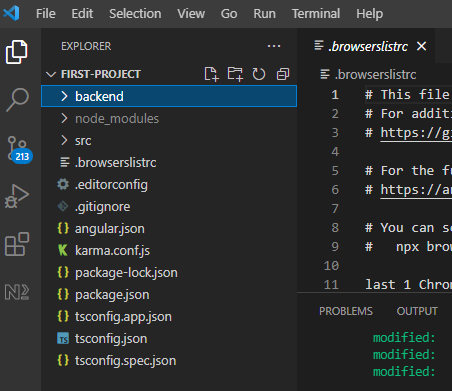


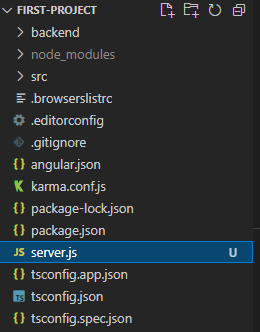
### Adding the Node Backend

We will go with two separate server apps for now:

1. **Angular**: ng serve
2. **NodeJS**: node server.js

We could develop into two separate VSCode/git projects, but we will just add a separate folder





|  |
| --- |
| server.js |
|  |

In bash terminal type node server.js

Or open a Node terminal

|  |
| --- |
| Run Node Terminal in VSCode |
| 1. Make sure you have node installed. 2. Shift + Command + ` to open the command pallete. 3. Or Shift + Command + ` and type "Terminal: Create New Integrated Terminal" 4. type node -v |

then type .load server.js

Node.js: What is require?

<https://nodejs.org/en/knowledge/getting-started/what-is-require/>

2011-08-26

**Node.js** follows the **CommonJS** module system, and the builtin require function is the easiest way to include modules that exist in separate files.

The basic functionality of require is that it reads a JavaScript file, executes the file, and then proceeds to return the exports object.

An example module:

|  |
| --- |
| Example.js |
| console.log("evaluating example.js");  var invisible = function () {  console.log("invisible");  }  exports.message = "hi";  exports.say = function () {  console.log(exports.message);  } |

So if you run var example = require('./example.js'), then example.js will get evaluated and then example be an object equal to:

{

message: "hi",

say: [Function]

}

Normally, exports starts as an empty object, {}. If you want to set the exports object to a function or a new object, you have to use the module.exports object. So for an example:

module.exports = function () {

console.log("hello world")

}

require('./example2.js')() //require itself and run the exports object

It is worth noting that each time you subsequently require an already-required file, the exports object is cached and reused. To illustrate this point:

node> require('./example.js')

evaluating example.js

{ message: 'hi', say: [Function] }

node> require('./example.js')

{ message: 'hi', say: [Function] }

node> require('./example.js').message = "hey" //set the message to "hey"

'hey'

node> require('./example.js') //One might think that this "reloads" the file...

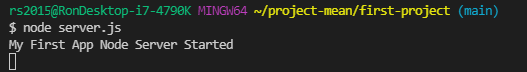
{ message: 'hey', say: [Function] } //...but the message is still "hey" because of the module cache.

As you can see from the above, example.js is evaluated the first time, but all subsequent calls to require() only invoke the module cache, rather than reading the file again. As seen above, this can occasionally produce side effects.

The rules of where require finds the files can be a little complex, but a simple rule of thumb is that if the file doesn't start with "./" or "/", then it is either considered a core module (and the local Node.js path is checked), or a dependency in the local node\_modules folder. If the file starts with "./" it is considered a relative file to the file that called require. If the file starts with "/", it is considered an absolute path. NOTE: you can omit ".js" and require will automatically append it if needed. For more detailed information, see [the official docs](https://nodejs.org/docs/v0.4.2/api/modules.html#all_Together...)

An extra note: if the filename passed to require is actually a directory, it will first look for package.json in the directory and load the file referenced in the main property. Otherwise, it will look for an index.js.

Now run the server code



### Adding the Express Framework

Express.js: Using middleware

<https://expressjs.com/en/guide/using-middleware.html>

Express is a routing and middleware web framework that has minimal functionality of its own: An Express application is essentially a series of middleware function calls.

***Middleware*** functions are functions that have access to the [request object](https://expressjs.com/en/4x/api.html#req) (req), the [response object](https://expressjs.com/en/4x/api.html#res) (res), and the next middleware function in the application’s request-response cycle. The next middleware function is commonly denoted by a variable named next.

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 function 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.

An Express application can use the following types of middleware:

* [Application-level middleware](https://expressjs.com/en/guide/using-middleware.html#middleware.application)
* [Router-level middleware](https://expressjs.com/en/guide/using-middleware.html#middleware.router)
* [Error-handling middleware](https://expressjs.com/en/guide/using-middleware.html#middleware.error-handling)
* [Built-in middleware](https://expressjs.com/en/guide/using-middleware.html#middleware.built-in)
* [Third-party middleware](https://expressjs.com/en/guide/using-middleware.html#middleware.third-party)

You can load application-level and router-level middleware with an optional mount path. You can also load a series of middleware functions together, which creates a sub-stack of the middleware system at a mount point.

Application-level middleware

Bind application-level middleware to an instance of the [app object](https://expressjs.com/en/4x/api.html#app) by using the app.use() and app.METHOD() functions, where METHOD is the HTTP method of the request that the middleware function handles (such as GET, PUT, or POST) in lowercase.

This example shows a middleware function with no mount path. The function is executed every time the app receives a request.

var express = require('express')

var app = express()

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

console.log('Time:', Date.now())

next()

})

This example shows a middleware function mounted on the /user/:id path. The function is executed for any type of HTTP request on the /user/:id path.

app.use('/user/:id', function (req, res, next) {

console.log('Request Type:', req.method)

next()

})

This example shows a route and its handler function (middleware system). The function handles GET requests to the /user/:id path.

app.get('/user/:id', function (req, res, next) {

res.send('USER')

})

Here is an example of loading a series of middleware functions at a mount point, with a mount path. It illustrates a middleware sub-stack that prints request info for any type of HTTP request to the /user/:id path.

app.use('/user/:id', function (req, res, next) {

console.log('Request URL:', req.originalUrl)

next()

}, function (req, res, next) {

console.log('Request Type:', req.method)

next()

})

Route handlers enable you to define multiple routes for a path. The example below defines two routes for GET requests to the /user/:id path. The second route will not cause any problems, but it will never get called because the first route ends the request-response cycle.

This example shows a middleware sub-stack that handles GET requests to the /user/:id path.

app.get('/user/:id', function (req, res, next) {

console.log('ID:', req.params.id)

next()

}, function (req, res, next) {

res.send('User Info')

})

// handler for the /user/:id path, which prints the user ID

app.get('/user/:id', function (req, res, next) {

res.send(req.params.id)

})

To skip the rest of the middleware functions from a router middleware stack, call next('route') to pass control to the next route. **NOTE**: next('route') will work only in middleware functions that were loaded by using the app.METHOD() or router.METHOD() functions.

This example shows a middleware sub-stack that handles GET requests to the /user/:id path.

app.get('/user/:id', function (req, res, next) {

// if the user ID is 0, skip to the next route

if (req.params.id === '0') next('route')

// otherwise pass the control to the next middleware function in this stack

else next()

}, function (req, res, next) {

// send a regular response

res.send('regular')

})

// handler for the /user/:id path, which sends a special response

app.get('/user/:id', function (req, res, next) {

res.send('special')

})

Middleware can also be declared in an array for reusability.

This example shows an array with a middleware sub-stack that handles GET requests to the /user/:id path

function logOriginalUrl (req, res, next) {

console.log('Request URL:', req.originalUrl)

next()

}

function logMethod (req, res, next) {

console.log('Request Type:', req.method)

next()

}

var logStuff = [logOriginalUrl, logMethod]

app.get('/user/:id', logStuff, function (req, res, next) {

res.send('User Info')

})

Router-level middleware

Router-level middleware works in the same way as application-level middleware, except it is bound to an instance of express.Router().

var router = express.Router()

Load router-level middleware by using the router.use() and router.METHOD() functions.

The following example code replicates the middleware system that is shown above for application-level middleware, by using router-level middleware:

var express = require('express')

var app = express()

var router = express.Router()

// a middleware function with no mount path. This code is executed for every request to the router

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

console.log('Time:', Date.now())

next()

})

// a middleware sub-stack shows request info for any type of HTTP request to the /user/:id path

router.use('/user/:id', function (req, res, next) {

console.log('Request URL:', req.originalUrl)

next()

}, function (req, res, next) {

console.log('Request Type:', req.method)

next()

})

// a middleware sub-stack that handles GET requests to the /user/:id path

router.get('/user/:id', function (req, res, next) {

// if the user ID is 0, skip to the next router

if (req.params.id === '0') next('route')

// otherwise pass control to the next middleware function in this stack

else next()

}, function (req, res, next) {

// render a regular page

res.render('regular')

})

// handler for the /user/:id path, which renders a special page

router.get('/user/:id', function (req, res, next) {

console.log(req.params.id)

res.render('special')

})

// mount the router on the app

app.use('/', router)

To skip the rest of the router’s middleware functions, call next('router') to pass control back out of the router instance.

This example shows a middleware sub-stack that handles GET requests to the /user/:id path.

var express = require('express')

var app = express()

var router = express.Router()

// predicate the router with a check and bail out when needed

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

if (!req.headers['x-auth']) return next('router')

next()

})

router.get('/user/:id', function (req, res) {

res.send('hello, user!')

})

// use the router and 401 anything falling through

app.use('/admin', router, function (req, res) {

res.sendStatus(401)

})

Error-handling middleware

Error-handling middleware always takes ***four*** arguments. You must provide four arguments to identify it as an error-handling middleware function. Even if you don’t need to use the next object, you must specify it to maintain the signature. Otherwise, the next object will be interpreted as regular middleware and will fail to handle errors.

Define error-handling middleware functions in the same way as other middleware functions, except with four arguments instead of three, specifically with the signature (err, req, res, next)):

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

console.error(err.stack)

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

})

For details about error-handling middleware, see: [Error handling](https://expressjs.com/en/guide/error-handling.html).

Built-in middleware

Starting with version 4.x, Express no longer depends on [Connect](https://github.com/senchalabs/connect). The middleware functions that were previously included with Express are now in separate modules; see [the list of middleware functions](https://github.com/senchalabs/connect#middleware).

Express has the following built-in middleware functions:

* [express.static](https://expressjs.com/en/4x/api.html#express.static) serves static assets such as HTML files, images, and so on.
* [express.json](https://expressjs.com/en/4x/api.html#express.json) parses incoming requests with JSON payloads. **NOTE: Available with Express 4.16.0+**
* [express.urlencoded](https://expressjs.com/en/4x/api.html#express.urlencoded) parses incoming requests with URL-encoded payloads. **NOTE: Available with Express 4.16.0+**

Third-party middleware

Use third-party middleware to add functionality to Express apps.

Install the Node.js module for the required functionality, then load it in your app at the application level or at the router level.

The following example illustrates installing and loading the cookie-parsing middleware function cookie-parser.

$ npm install cookie-parser

var express = require('express')

var app = express()

var cookieParser = require('cookie-parser')

// load the cookie-parsing middleware

app.use(cookieParser())

For a partial list of third-party middleware functions that are commonly used with Express, see: [Third-party middleware](https://expressjs.com/en/resources/middleware.html).

Express.js: The app object

<https://expressjs.com/en/api.html#app>

The core part of an Express app is the Application object. It’s the application itself.

app object

The app object conventionally denotes the Express application. Create it by calling the top-level express() function exported by the Express module:

var express = require('express')

var app = express()

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

res.send('hello world')

})

app.listen(3000)

The app object has methods for

* Routing HTTP requests; see for example, [app.METHOD](https://expressjs.com/en/api.html#app.METHOD) and [app.param](https://expressjs.com/en/api.html#app.param).
* Configuring middleware; see [app.route](https://expressjs.com/en/api.html#app.route).
* Rendering HTML views; see [app.render](https://expressjs.com/en/api.html#app.render).
* Registering a template engine; see [app.engine](https://expressjs.com/en/api.html#app.engine).

It also has settings (properties) that affect how the application behaves; for more information, see [Application settings](https://expressjs.com/en/api.html#app.settings.table).

The Express application object can be referred from the [request object](https://expressjs.com/en/api.html#req) and the [response object](https://expressjs.com/en/api.html#res) as req.app, and res.app, respectively.

app.use

app.use([path,] callback [, callback...])

Mounts the specified [middleware](https://expressjs.com/guide/using-middleware.html) function or functions at the specified path: the middleware function is executed when the base of the requested path matches path.

Arguments

|  |  |  |
| --- | --- | --- |
| **Argument** | **Description** | **Default** |
| path | The path for which the middleware function is invoked; can be any of:   * A string representing a path. * A path pattern. * A regular expression pattern to match paths. * An array of combinations of any of the above.   For examples, see [Path examples](https://expressjs.com/en/api.html#path-examples). | '/' (root path) |
| callback | Callback functions; can be:   * A middleware function. * A series of middleware functions (separated by commas). * An array of middleware functions. * A combination of all of the above.   You can provide multiple callback functions that behave just like middleware, except that these callbacks can invoke next('route') to bypass the remaining route callback(s). You can use this mechanism to impose pre-conditions on a route, then pass control to subsequent routes if there is no reason to proceed with the current route.  Since [router](https://expressjs.com/en/api.html#router) and [app](https://expressjs.com/en/api.html#application) implement the middleware interface, you can use them as you would any other middleware function.  For examples, see [Middleware callback function examples](https://expressjs.com/en/api.html#middleware-callback-function-examples). | None |

Description

A route will match any path that follows its path immediately with a “/”. For example: app.use('/apple', ...) will match “/apple”, “/apple/images”, “/apple/images/news”, and so on.

Since path defaults to “/”, middleware mounted without a path will be executed for every request to the app.  
For example, this middleware function will be executed for **every** request to the app:

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

console.log('Time: %d', Date.now())

next()

})

**NOTE**

Sub-apps will:

* Not inherit the value of settings that have a default value. You must set the value in the sub-app.
* Inherit the value of settings with no default value.

For details, see [Application settings](https://expressjs.com/en/4x/api.html#app.settings.table).

Middleware functions are executed sequentially, therefore the order of middleware inclusion is important.

// this middleware will not allow the request to go beyond it

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

res.send('Hello World')

})

// requests will never reach this route

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

res.send('Welcome')

})

****Error-handling**** middleware

Error-handling middleware always takes **four** arguments. You must provide four arguments to identify it as an error-handling middleware function. Even if you don’t need to use the next object, you must specify it to maintain the signature. Otherwise, the next object will be interpreted as regular middleware and will fail to handle errors. For details about error-handling middleware, see: [Error handling](https://expressjs.com/en/guide/error-handling.html).

Define error-handling middleware functions in the same way as other middleware functions, except with four arguments instead of three, specifically with the signature (err, req, res, next)):

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

console.error(err.stack)

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

})

Express.js: request object

<https://expressjs.com/en/api.html#req>

The req object represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, and so on. In this documentation and by convention, the object is always referred to as req (and the HTTP response is res) but its actual name is determined by the parameters to the callback function in which you’re working.

For example:

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

res.send('user ' + req.params.id)

})

But you could just as well have:

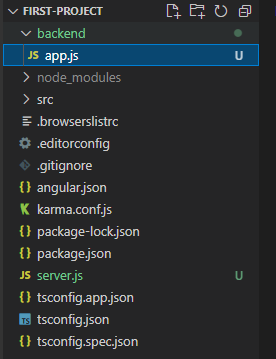
app.get('/user/:id', function (request, response) {

response.send('user ' + request.params.id)

})

The req object is an enhanced version of Node’s own request object and supports all [built-in fields and methods](https://nodejs.org/api/http.html#http_class_http_incomingmessage).

Add file for express server backend code



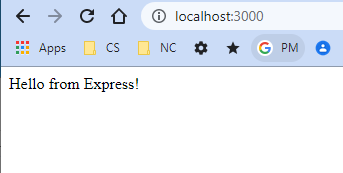
|  |
| --- |
| app.js |
|  |

|  |
| --- |
| server.js |
|  |

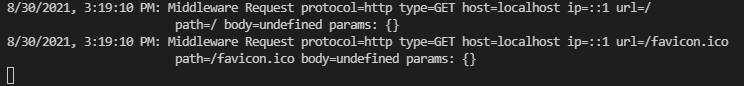
Now run the server



Open port 3000 in Chrome (reload page if necessary)



You should see something like the following appear in the node terminal. Notice that two messages are received.



### Improving the server.js Code

Install nodemon

npm install --save-dev nodemon

This allows our running node server to automatically update whenever the source file changes.

|  |
| --- |
| package.json |
|  |

Now instead of node server.js we type

npm run start:server

### Fetching Initial Posts

### Using the Angular HTTP Client

### Understanding CORS

### Adding the POST Backend Point

### Adding Angular

### Node + Express Section Resources

Section Resources

* Learn Node + Express from Scratch (for free!):

<https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs>

* Creating a REST API with Node + Express (+ MongoDB):

<https://academind.com/learn/node-js/building-a-restful-api-with/>

## Working with MongoDB

Completed thru Chap 37