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

## Section 1 Getting Started

**Chap 1, 2, 3 Intro**

**Chap 4 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.

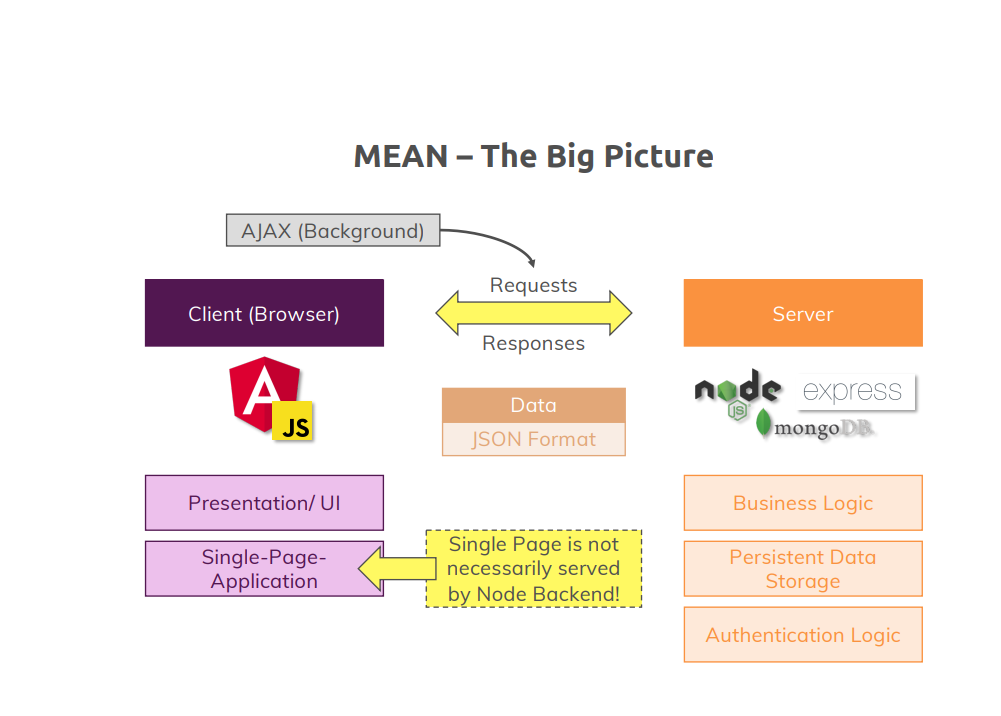
**Chap 5 How Does a MEAN Stack work?**

**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 |

**Chap 6 Must read notes**



**Chap 7 Installing Node and Angular CLI**

Download and install node (see other notes)

Install Angular (see Angular course)

cd ~

mkdir project-mean

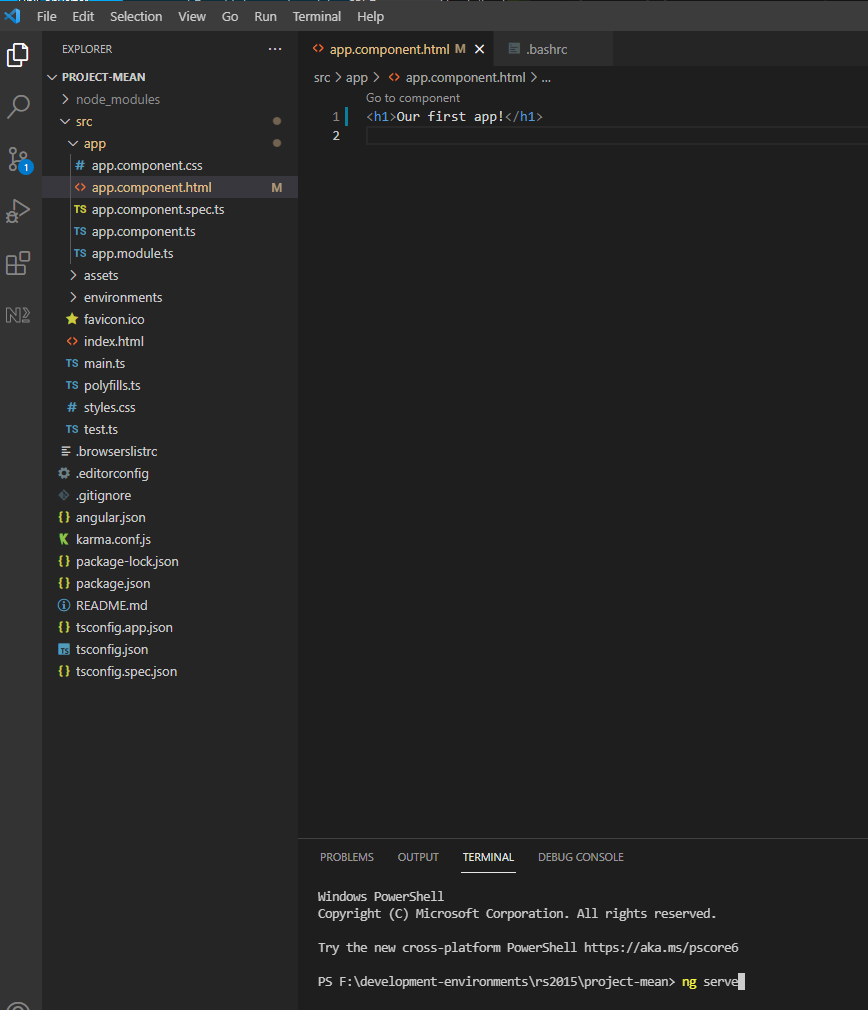
cd project-mean

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

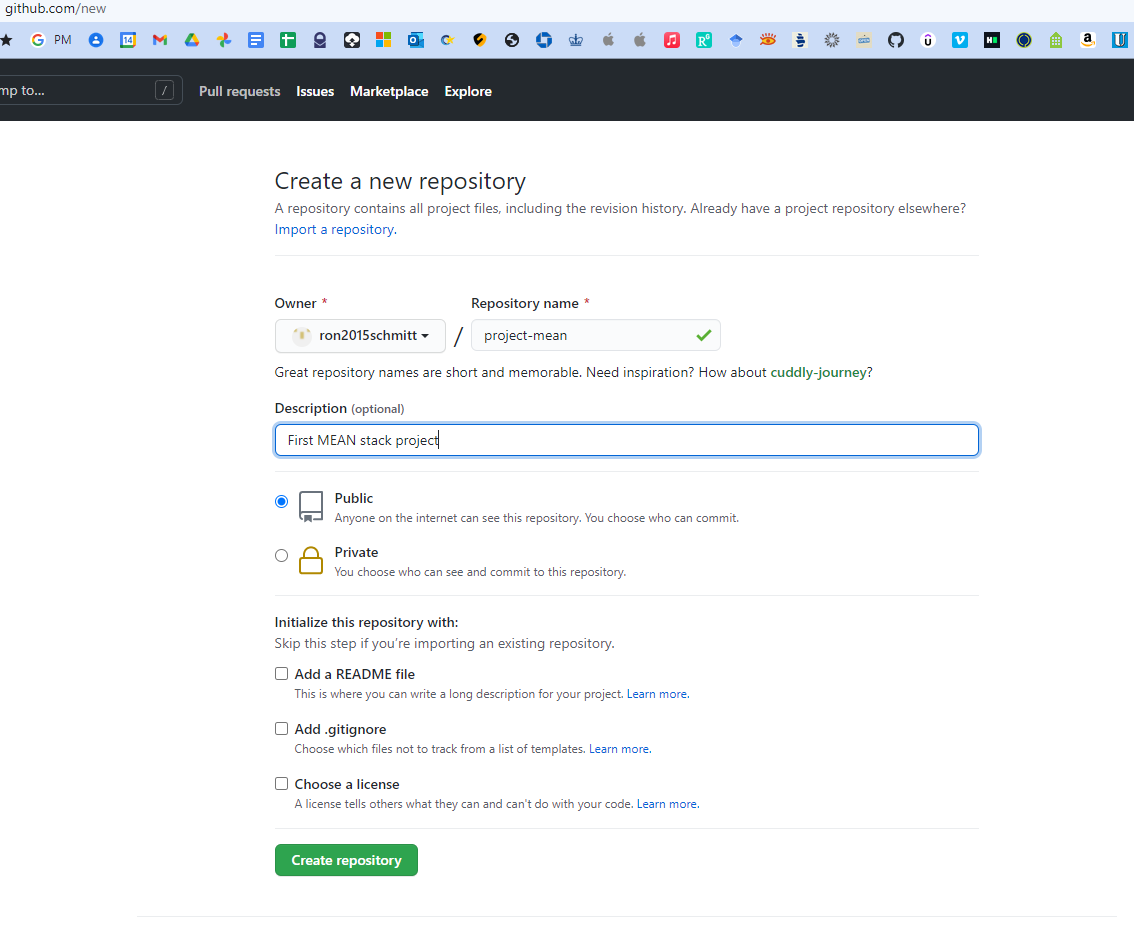
The following worked for github but probably is not best way to do it.

cd ~/project-mean

git init

git add .

create new repo on github

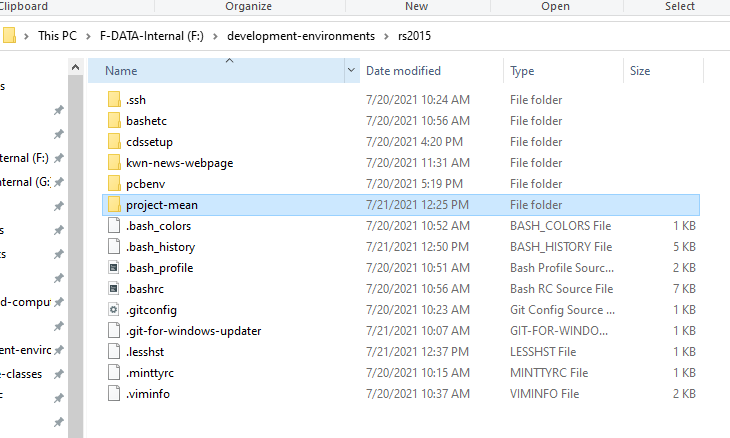


git branch -M main

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

git push -u origin main

Right-click folder and open in VSCode



**Chap 8 Installing Our IDE**

Install VSCode (see other notes)

Install VSCode extensions:

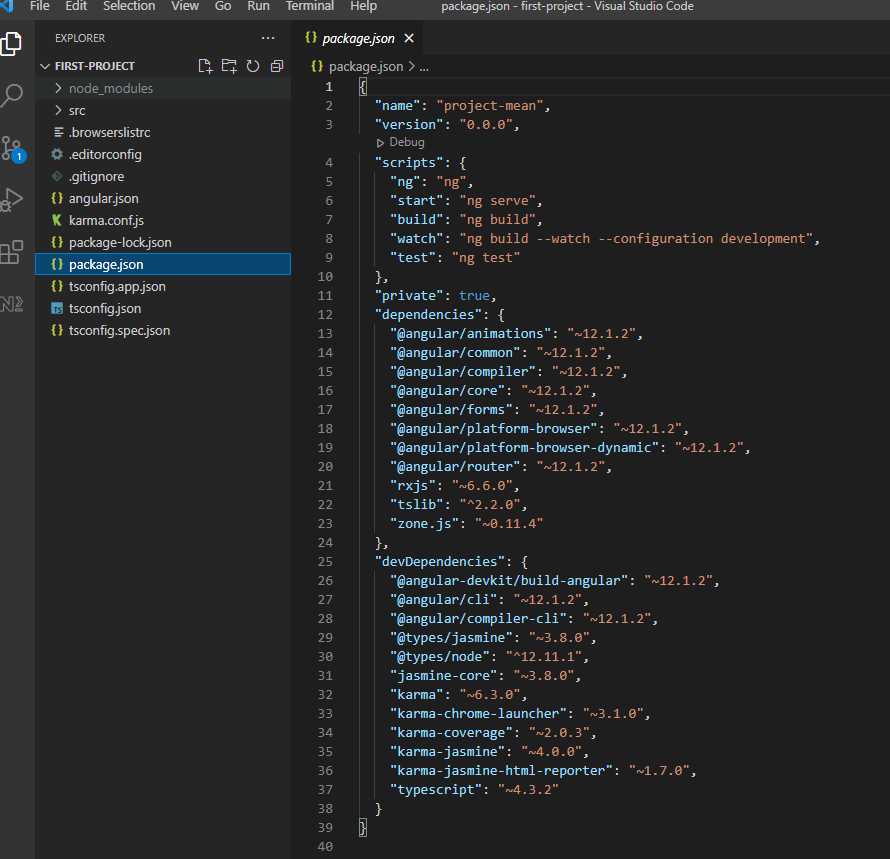
* Angular Essentials
* Material Icon Theme

**Chap 9 Exploring the Project Structure**

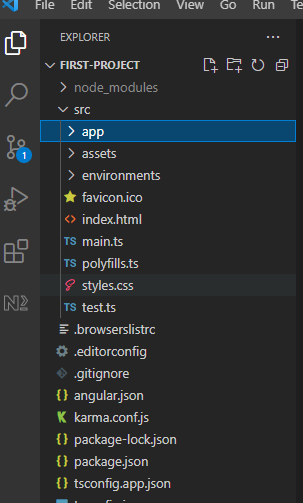
Completed through Chap 8

Open project-mean folder in VSCode

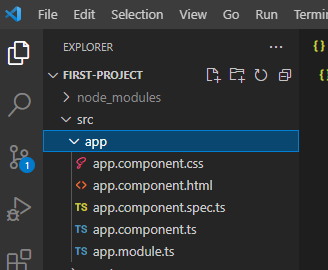
The package.json file list all the node packages needed, our dependencies. When you type npm install these package downloaded and built in the node\_modules folder



Our source code resides in the src folder



The main Angular component is defined in the app folder



**Chap 10 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.

**Chap 11 How to get the most out of this course**

Code along with the video

Ask questions on Q&A

**Chap 12: section resources**

## Section 2 The Angular Front End

**Chap 13: Angular: Introduction**

We’re going to build a mini social network

**Chap 14: Understanding the folder structure**

See Chap 7 of the Angular course

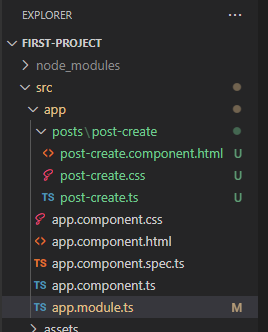
**Chap 15: Understanding Angular Components**

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.

**Chap 16: 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> |

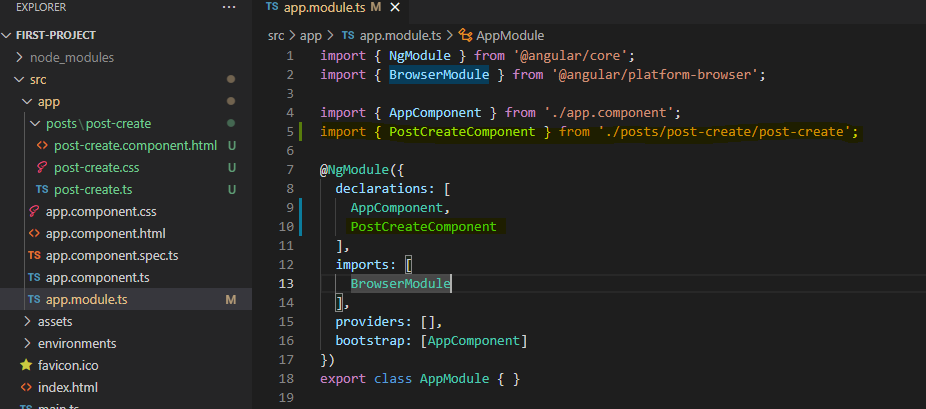
|  |
| --- |
| **post-create.ts** |
| import { Component } from "@angular/core";  @Component({    selector: 'app-post-create',    templateUrl: './post-create.component.html'  })  export class PostCreateComponent {  } |

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** |
|  |

**Chap 17: 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** |
|  |

**Chap 18: 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** |
|  |

**Chap 19: 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** |
|  |

**Chap 20: 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** |
|  |

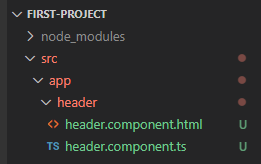
|  |
| --- |
| **post-create.component.css** |
|  |

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

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

**Chap 21: 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** |
|  |

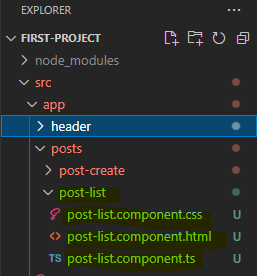
|  |
| --- |
| **app.module.ts** |
|  |

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

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

**Chap 22: 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** |
|  |

**Chap 23: Diving Into Structural Directives**

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

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

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

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

**Chap 24: Creating Posts with Property & Event Binding**

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

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

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

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

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

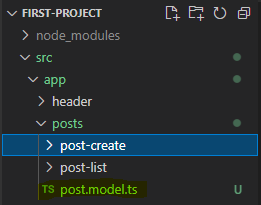
|  |
| --- |
| **post-create.component.css** |
|  |

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

**Chap 25: Creating a Post Model**

Here we create a data type for a Post

Create new file as shown



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

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

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

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

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

**Chap 26: 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** |
|  |

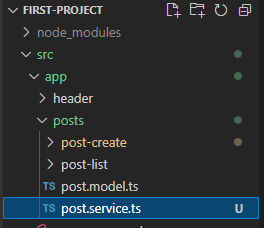
|  |
| --- |
| **post-create.component.html** |
|  |

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

**Chap 27: 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** |
|  |

**Chap 28: 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.

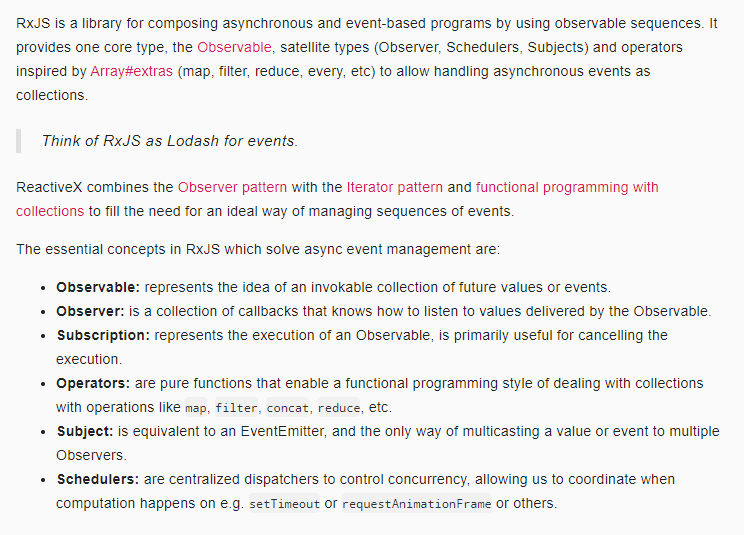
|  |
| --- |
| **post-list.component.ts** |
|  |

Now use PostsService in the create code.

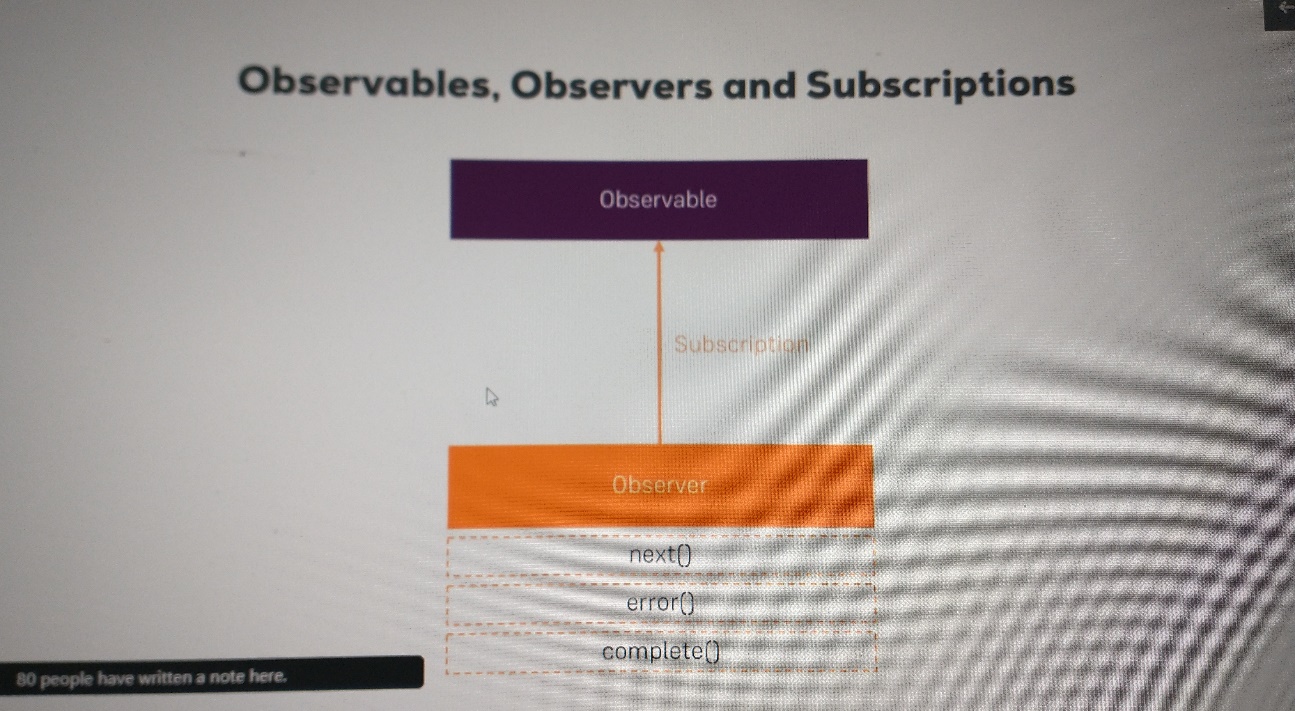
|  |
| --- |
| **post-create.component.ts** |
|  |

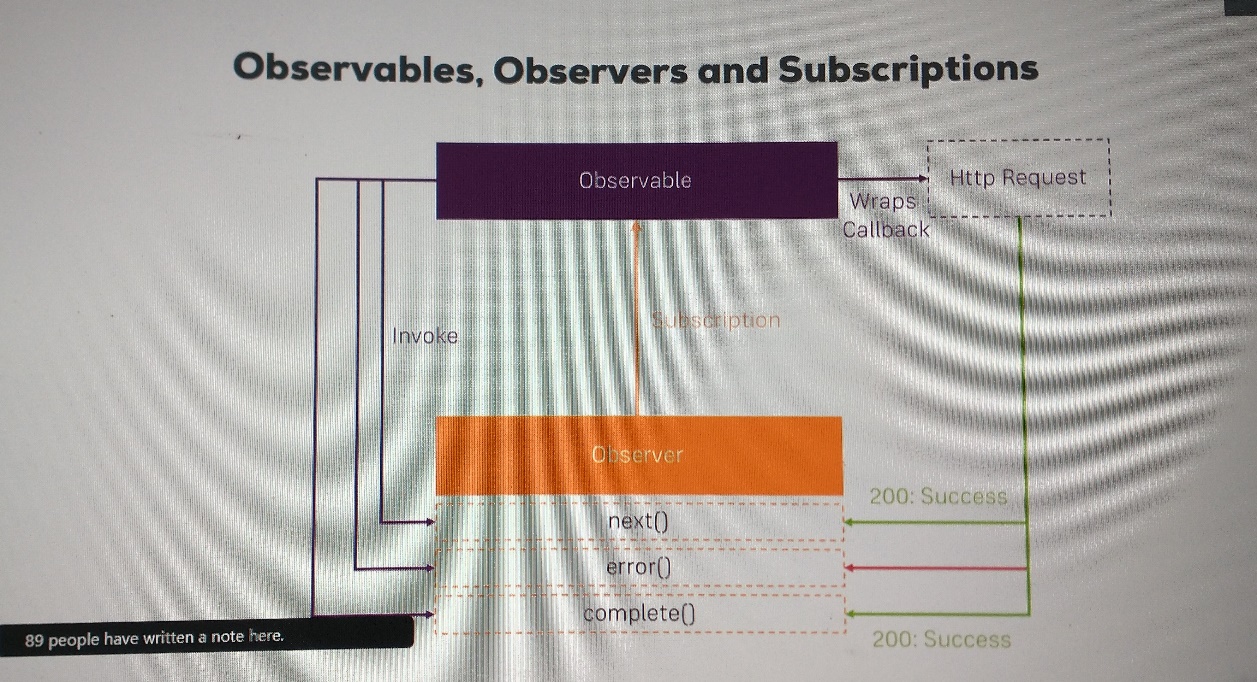
**Chap 29: More About Observables**

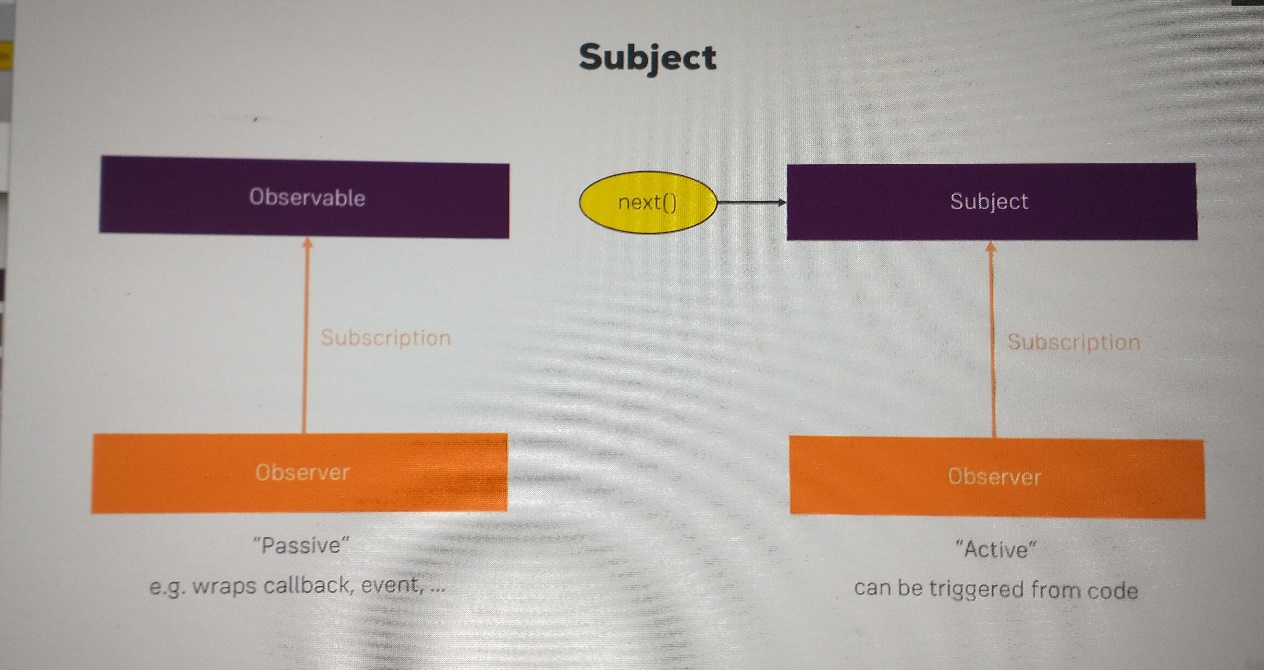
We use RxJS

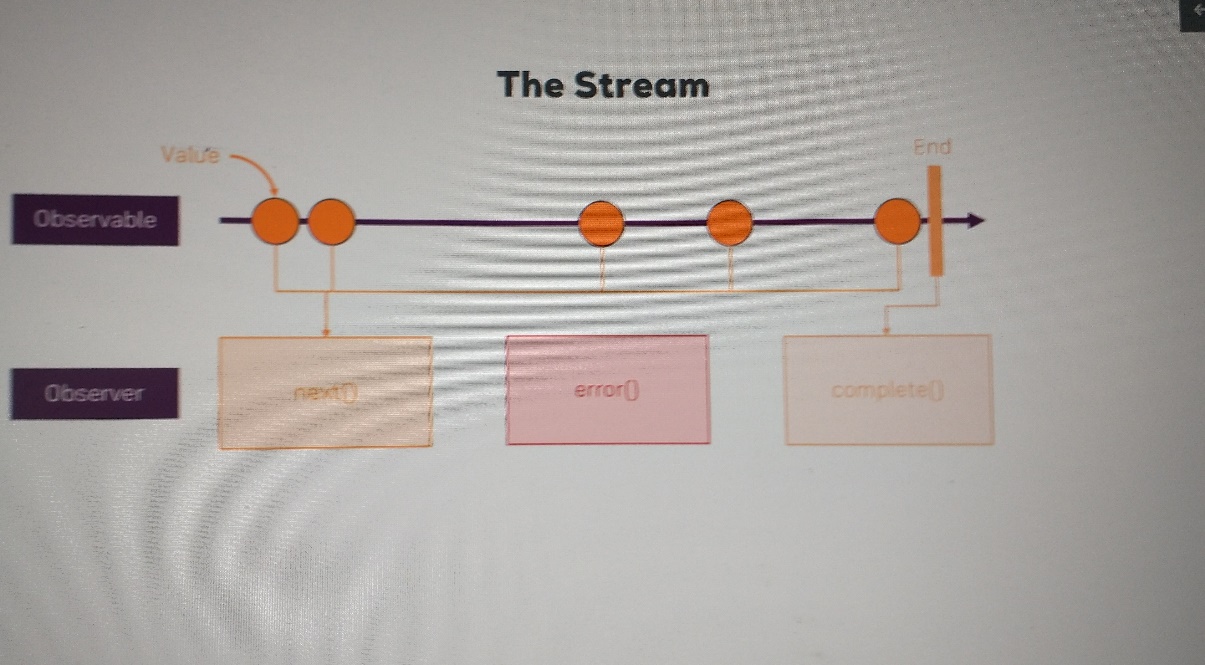


<https://www.youtube.com/watch?v=T9wOu11uU6U>









**Chap 30: 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** |
|  |

**Chap 31: Angular Section Resources**

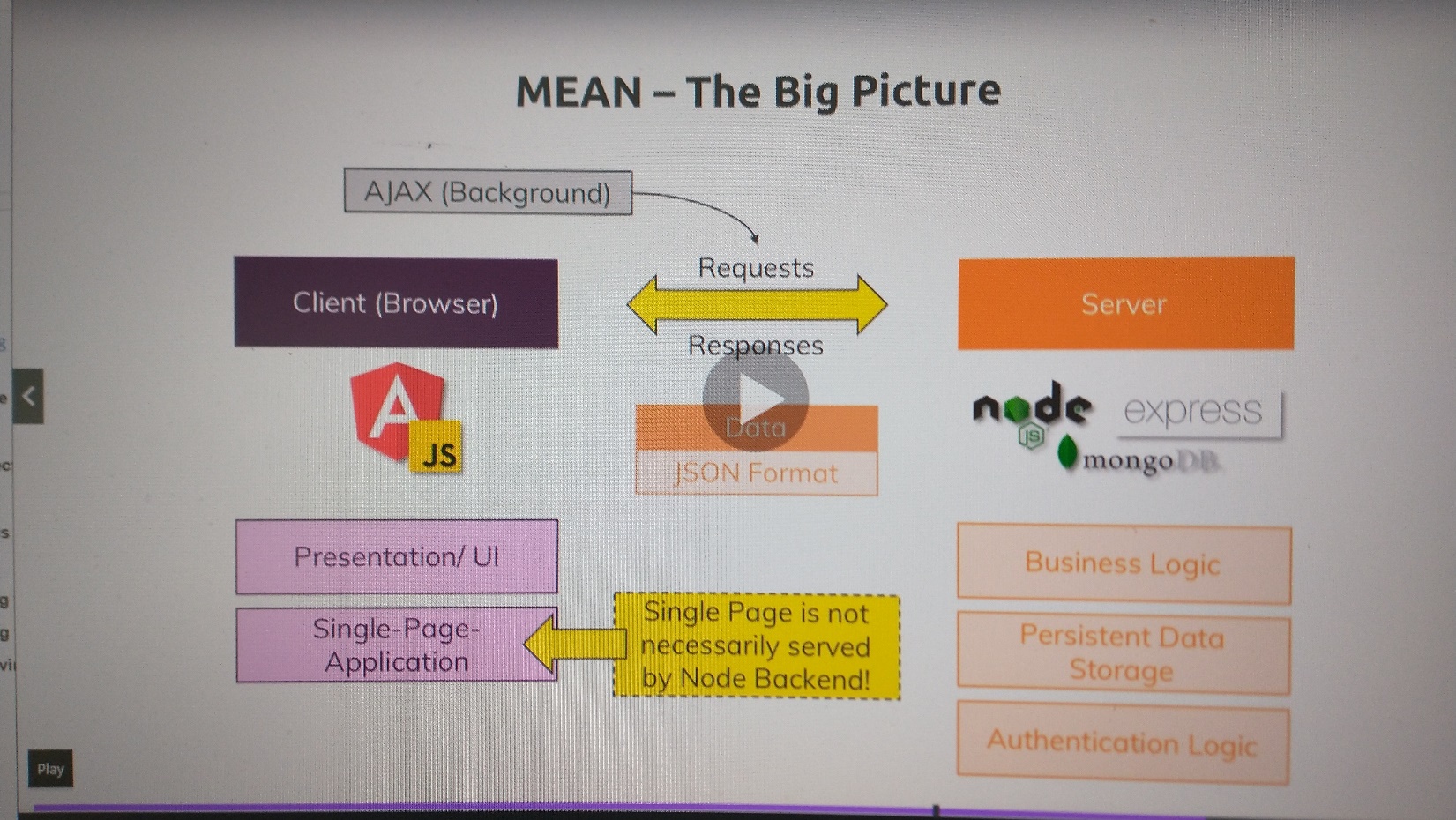
Attached to this lecture, you find code snapshots for the course section. In addition, this links might be helpful as well:

* 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/>

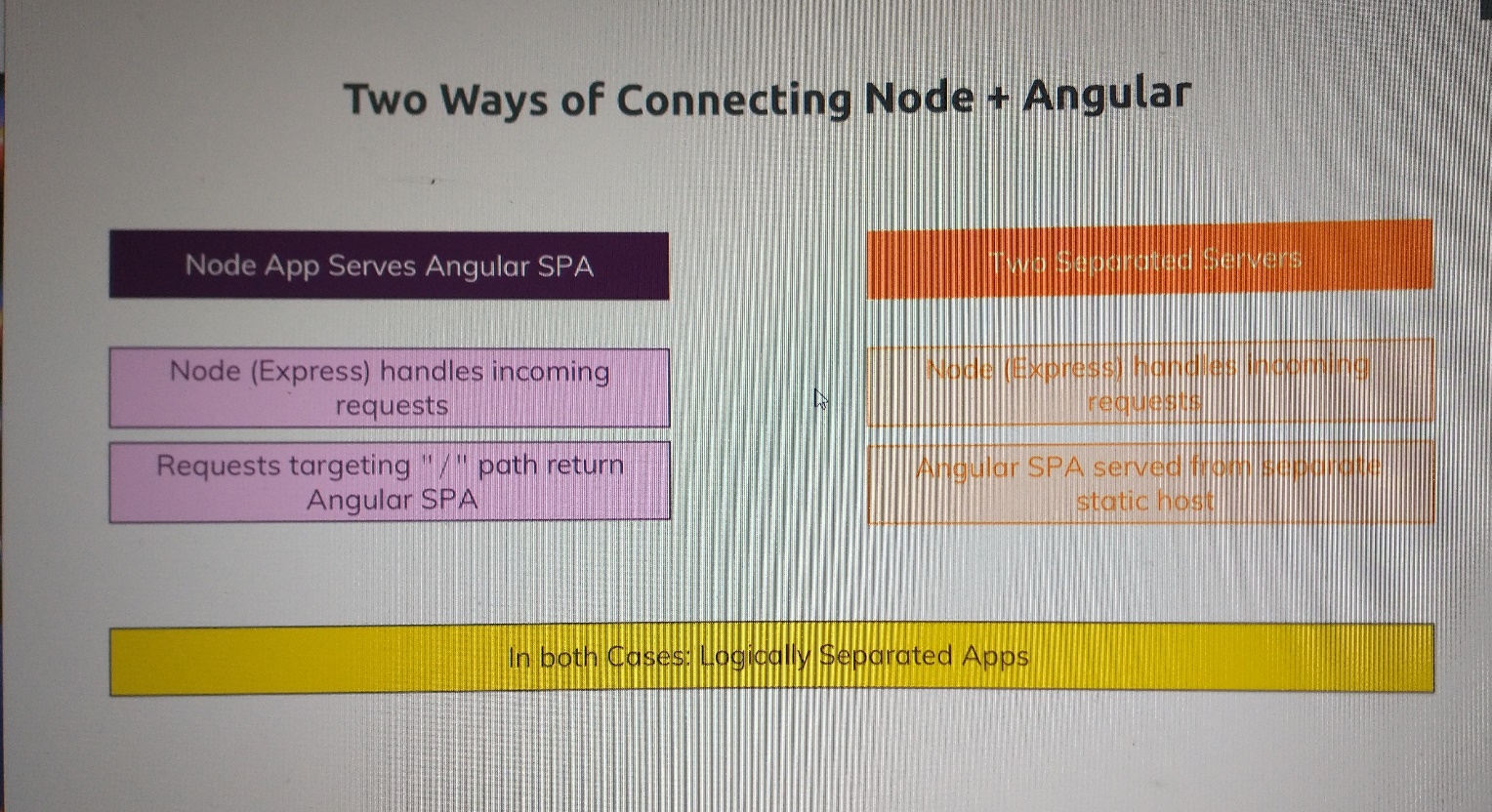
[RxJS: Reactive Extensions Library for JavaScript](https://rxjs.dev/)

## Section 3 Adding NodeJS and Express to Our Project

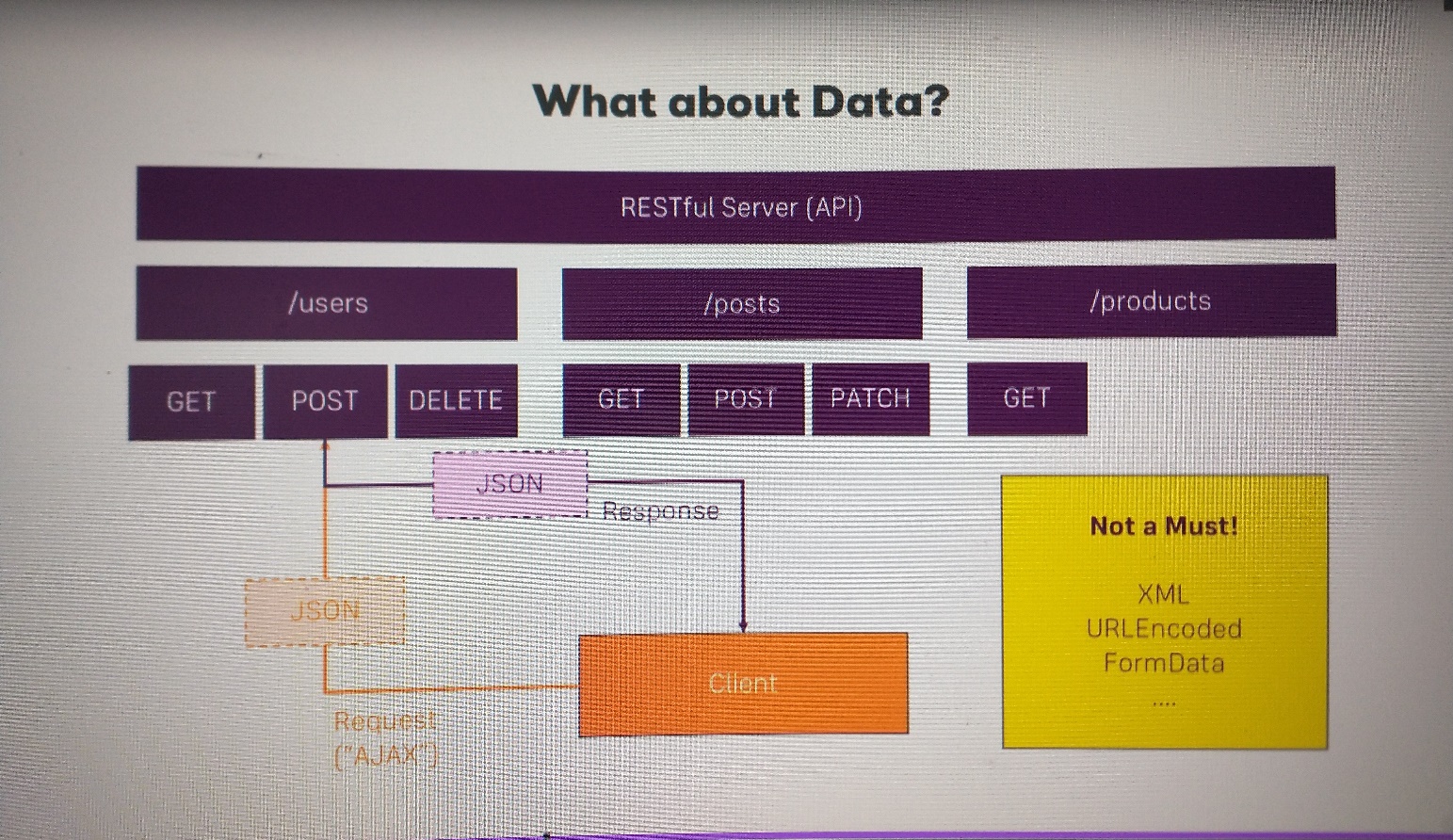
**Chap 32: Module Introduction**



**Chap 33: Connecting Node & Angular - Theory**



**Chap 34: What is a RESTful API?**

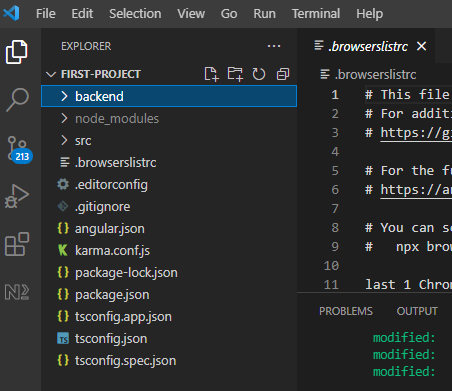


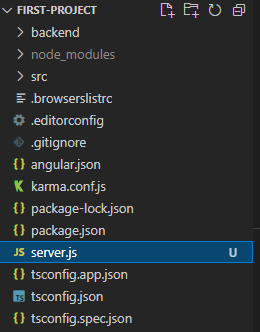
**Chap 35: Adding the Node Backend**

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

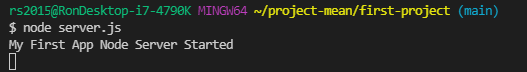




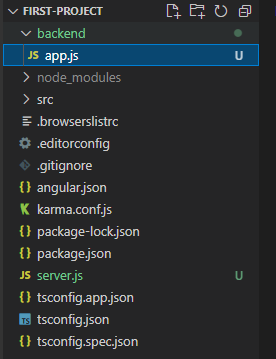
|  |
| --- |
| **server.js** |
|  |

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



**Chap 36: Adding the Express Framework**



**Chap 37: Improving the server.js Code**

**Chap 35:**

**Chap 35:**

**Chap 35:**

**Chap 35:**

Completed thru Chap 31