Angular

Update angular

npm install --save-dev @angular/cli@latest

Update Node Package Manager

npm install npm@latest -g

Martins-MacBook-Pro:GroceryL Isaac$ git add src/app/\*

Martins-MacBook-Pro:GroceryL Isaac$ git commit -m "version 2"

Make a new file:

cd to destination

ng new name –-style=scss

cd Desktop/

git clone <https://github.com/MartinVAvalos/Envy.git> (url from github; all it does clone something from github)

git commit –m “nameOfVersion” (file committed)

(Before committing a new file, you must add it. If it was already created, you can skip the add step)

f

Martins-MacBook-Pro-2:~ martinvera$ cd Desktop/

Martins-MacBook-Pro-2:Desktop martinvera$ git clone https://github.com/MartinVAvalos/Envy.git

Cloning into 'Envy'...

warning: You appear to have cloned an empty repository.

Martins-MacBook-Pro-2:Desktop martinvera$ cd Envy

Martins-MacBook-Pro-2:Envy martinvera$ gedit read.md

(gedit:1986): Gtk-**WARNING** \*\*: Theme directory 256x256@2/animations of theme hicolor has no size field

^Z

[1]+ Stopped gedit read.md

Martins-MacBook-Pro-2:Envy martinvera$ ls

read.md

Martins-MacBook-Pro-2:Envy martinvera$ git commit

On branch master

Initial commit

Untracked files:

read.md

nothing added to commit but untracked files present

Martins-MacBook-Pro-2:Envy martinvera$ git add read.md

Martins-MacBook-Pro-2:Envy martinvera$ git commit -m "version 1.0" read.md

[master (root-commit) f4e8579] version 1.0

1 file changed, 1 insertion(+)

create mode 100644 read.md

Martins-MacBook-Pro-2:Envy martinvera$ git push

Counting objects: 3, done.

Writing objects: 100% (3/3), 226 bytes | 0 bytes/s, done.

Total 3 (delta 0), reused 0 (delta 0)

To https://github.com/MartinVAvalos/Envy.git

\* [new branch] master -> master

Martins-MacBook-Pro-2:Envy martinvera$

**Updating local Angular cli:** npm install --save-dev @angular/cli@latest

**Sizes:**

Width/Height: Use %

Font: Use em

Side Notes:

~class affects many tags. Id is used affect one tag under that id.

This link will help make a new project:

<https://www.udemy.com/the-complete-guide-to-angular-2/learn/v4/t/lecture/6655614/>

<https://www.youtube.com/watch?v=aiXNKHKWlmY&t=53s>

Force command in terminal

sudo (command)

CHECK (Section3-41)

Install Boostrap version 3

npm install --save bootstrap@3

(download in angular terminal)

(in angular.json, add "node\_modules/bootstrap/dist/css/bootstrap.min.css", below styles.

Ex:

"styles": [

node\_modules/bootstrap/dist/css/bootstrap.min.css",

"src/styles.css"

],

)

or

"../node\_modules/bootstrap/dist/css/bootstrap.min.css",

"src/styles.css"

],

)

Depends where file is located

**Main Course Notes:**

Color : means text is interchangable. Text refers to how it is interchangable.

Color: means text is exactly as written. Used for clarification.

Routing ~ Management of different Urls.

Observables ~ Allows you to work with asynchronus code.

Forms ~ Handles user input.

Pipes ~ Transforms output. Changes what you display on template at runtime.

Http ~ Angular cannot reach out to a web server. Angular cannot connect to a database directly, but it can connect to a server that is able to.

Typescript ~   
A superset of Javascript. Offers more features than javascript. Allows you to you to define if a variable is a number, string, or something else. Javascript allows variables to be anything (dynamic typing). Since Typescript is more strict with the variables, forcing the programmer to write more rubust code. Typescript also checks the code as you write it; not just when you run it. Typescript doesn’t run in the browser; it is compiled into Javascript in the end (which is why we need the CLI. The CLI does this compilation).

Make sure “ng ser

ve” is running in terminal, otherwise the page will not update.

Typescript {*Selecting by “id” won’t work; it is not supported by anguar*}

selector: `\_\_`, (by element)

selector: `[\_\_] (by attribute)

selector: '.\_\_', (by class)

Html

<\_\_></\_\_> (by element)

<div \_\_></div> (by attribute)

<div class="\_\_"></div>

**NG G C Terminal command info:**

Generate a component using terminal:

… martinvera$ ng generate component (name)

or

… martinvera$ ng g c (name)

**Note:** Replace c with **d** to create a directive instead of a component.

ng g c recipe -- spec false

(-- spec false prevents the creation of a testing file; if you don’t need it)

You can also set a path to where you want to generate components

Ex:

… martinvera$ ng g c ./recipes/recipe-list

**Note:** Replace c with **p** to create a pipe instead of a component.

ng g p name

**Note:** We can receive a value (string) of an element of an array.

Line 14 ~ item is an element of an array. We access propName (a string) of an element by using []. (An example can be found in code “pipes-start” file “filter.pipe.ts”

**Data Binding**

Databinding = Communication

--> Output Data -->

String Interpolation: ( {{ data }} )

Property Binding: ( [property] = “data” )

<-- React to (User) Events <--

Event Binding: ( (event) = “expression” )

Combination of Both: **Two-Way-Binding**

( [(ngModel)] = “data” )

Typescript Code (business logic)

Template

(HTML)

Data binding is a communication between Typescript (Business Logic) and HTML (template

<button class = "btn btn-primary" [disabled]>Add Server</button>

(

~Disabled as the name suggests disables the button

~[ ] The square brackets indicate to angular that we are using property binding. That we want to dynamically bind some property.

If you want to output something in your template, use string interpolation.

If you want to change a property on an html element/directive/component, use property binding.

WARNING:

Don’t mix property binding and string interpolation. It will break the code.

( ) signifies event binding.

[ ] signifies property binding.

How do you know to which Properties or Events of HTML Elements you may bind? You can basically bind to all Properties and Events - a good idea is to console.log()  the element you're interested in to see which properties and events it offers.

**Important**:For Two-Way-Binding (covered in the next lecture) to work, you need to enable the ngModel  directive. This is done by adding the FormsModule  to the imports[]  array in the AppModule.

You then also need to add the import from @angular/forms  in the app.module.ts file:

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

**Important**: For events, you don't bind to onclick but only to click (=> (click)).

The MDN (Mozilla Developer Network) offers nice lists of all properties and events of the element you're interested in. Googling for YOUR\_ELEMENT properties  or YOUR\_ELEMENT events  should yield nice results.

Important: For Two-Way-Binding to work, you need to enable the ngModel  directive. This is done by adding the FormsModule  to the imports[]  array in the AppModule.

You then also need to add the import from @angular/forms  in the app.module.ts file:

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

Angular is a javascript framework, changing your DOM (HTML) at runtime.

How to inline your templates? Section 2-19

Directives are instructions in the DOM

The star(\*) in \*ngIf indicates that this is a structural directive.

# is a local reference marker.

**Example of ngIF as well as else statement (structural directive)**

<!—Line bellow is an if statement “ngIf”. The else points to the else statement -->

<p \*ngIf = "serverCreated; else noServer">Server was created, server name is {{ serverName }}</p>

<ng-template #noServer> <!-- Else statement. #noServer a marker of else stment-->

<p>No server was created!</p>

</ng-template>

Unlike stuctural directives, attribute directives do not add or remove elements. The only change the element they were placed on.

**Example of (attribute directive)** (more info may be in slide 2-35)

<cusom-made-attribute></ cusom-made-attribute>

**ngStyle**

ngStyle allows user to change the CSS style itself

**(Recap in Section 7-84)**

**ngClass**

ngClass allows user to user to dynamically add or remove classes

**(Recap in Section 7-84)**

**ngFor (section 2-37)**

ngFor allows user to manipulate the number of components.

Ex: <app-server \*ngFor=”let server of servers”></app-server>

let is the temporary variable

server is the name of let

servers is the property

**NOTE (Applies to section 3-46)**

The way we added it, the Navbar will collapse on smaller screens. Since we didn't implement a Hamburger menu, that means that there's no way of accessing our links on smaller screens.

You can either add such a menu on your own, or you replace collapse navbar-collapse  with just navbar-default

**Note:** A model is basically a typescript file.

**Property and Event Binding** (What we can use these tools on)

**- HTML Elements ~** Native properties and events.

**- Directives ~** Custom Properties and Events

**- Customs ~** Custom Property and Events

**Passing data from data from one component down to another component which was implemented there.**

**Note:** (In order to use **@Input()**, you need to import **Input** at the top from @angular/core). To allow parent components to be able to bind to a custom property you must add a decorator (decorators are not only available to classes). The decorator you need is **@Input()**. The String inside the (' ') of @Input is the alias of the property. The String will be how the property is called when using the String outside the component.

**Other direction of passing data (refering to above version of passing data.) We have a component and something changes in there and we want to inform our parent component.**

**Note: $event** can be used to catch data from a method. The captured data can be an object.

Ex: method($event)

**Note:** How do you make a property into an event?

(In order to use **EventEmitter**, you need to import **EventEmitter** at the top from @angular/core) (In order to use **@Output()**, you need to import **Output** at the top from @angular/core) **EventEmitter** is a generic type which is indicated in typescript by using **<>**. In-between **<>**, you define the type of event data you’re going to emit such as {serverName: string, serverConent:string}. Use () at the end to call the constructor of EventEmitter and create a new EventEmitter object. You can also add an alias to **Output**.

We use **EventEmitter** to emit a new event to pass data to the parent component. **(Section 6-81)**

Ex: @Output property = new EventEmitter<{}>();

**Angular Debugging~**

Flow:

Read Error Messages --> Use Source Maps to debug --> Use Augury to understand the application and your dependencies

Step1 – inspect

Step 2 – click sources

Step 3 – click webpack

Step 4 - . (file)

Step 5 src (file)

* You can click line number to make a break point. (then run code by clicking it)
* You can hover over highlighted variable in break point to see more info on how data is being manipulated.
* **Augury** is another tool you can use. Should be located near sources button when inspecting. If not, click on the arrow pointing right >> to find it.

~For more complicated code. You can use the injector graph. This allows you to see dependencies between your components and your services and your app.

**Note:** 3 modes of overwriting encapsulation

encapsulation: ViewEncapsulation.(Native, None, or Emulated)

**Emulated** is is the default of angular, so you wouldn’t have to add ViewEncapsulation.Emulated to use it.

If you use **None**, the css used in that component will be global in order for the html file to exucute the component’s css.

**Native** uses the Shadow Dom technology. This should give you the same result as before with emulated, but only in browsers which support it, which is why in most cases you want to choose Emulated, but be aware that you could switch to None or Native too.

**Note:** How to make a local reference?

Use pound symbol (#) then a name of your choice. This reference will hold a reference to the whole HTML element with all it’s properties. You can use them everywhere in your template. You can only use it in the template, so you can’t use it in the typescript code. We can use a method that calls to typescript code inside the template.

Ex on how to create local reference: #name

Ex on how to use local reference: onAddServer(name)

**Note:** What is decorator **@ViewChild**?

(You must import **ViewChild** into @angular/core before using it. You must also pass an argument within ‘ ’ for it to work. This argument is the selector of the element.)

Ex: @ViewChild('serverContentInput') serverContentInput;

We can pass as a string; the name of a local reference (**serverContentInput**), so **serverContentInput** could now be passed as a string. If you want to select a component instead of passing a string you can do. **Look at next note to see why you may have to add ElementRef** **at the end of the example.**

Ex: @ViewChild(cockpitComponent) serverContentInput ElementRef;

(You would basically emit the ‘ ’ to select a component. This will allow you to access the first occurrence of the component in the app component ~possibly the first occurrence within the previous folder~)

**Note:** When using ViewChild you may end up using **ElementRef**. **ElementRef** is a reference to an element, and must be imported to @angular/core.

Ex: @ViewChild('serverContentInput') serverContentInput: ElementRef;

**Note:** You should use tools like String Interpolation and Property Binding if you want to output something in the DOM

**Note: <ng-content>** is a directive that acts as a marker. This marker tells angular to paste all the contents within <p></p> of app.component.html onto the marker.

**Note:** ngOnInit is a life cycle hook, and angular supports a couple of life cycle hooks. Life cycle details will be in Section 5-22.

**Lifecyle (Hooks; all hooks must be imported and exported; copy the name without ng to import/export)**

**ngOnChanges**: called after a bound input property changes.

(It is executed multipletimes. It is executed at the start as soon as a new component is created. It is also called when one of our bound input properties changes. Only hook that requires an argument. Argument must also be imported)

**ngOnInit**: Called once the component is initialized.

(ngOnInit will run after a constructor)

**ngDoCheck**: Called during every change detection run.

(This will run on every check. Example: Suppose you clicked on a button that does absolutely nothing. Angular still needs to check this event, so this will trigger ngDoCheck. It will be called during triggering events such as a timer or an observable was resolved. On these occassions Angular will check your code, and on these occasions ngDoCheck will be executed) ~Side Note: ngDoCheck will run one extra time in development mode. Angular has one extra detection cycle in development mode~

**These lifecycles may sound very inefficient, but angular does this in a** **very efficient way.**

**ngAfterContentInit:** Called after content (ng-content) has been projected into view.

**ngAfterContentChecked:** Called every time the projected content has been checked. (called after each change detection cycle)

**ngAfterViewInit:** Called after the component’s view (and child views) has been initialized.

(called once our view is rendered)

**ngAfterViewChecked:** Called every time the view (and child views) have been checked.

**ngOnDestroy:** Called once the component is about to be destroyed. (used for clean-up work)

“ ..“ is a tool that let’s you go to the father of that file

ex: '../recipe.model';

**Attribute vs Structural (Section 7-82)**

Attribute Directives

~ Look like normal HTML Attribute (possibly with databinding or event binding)

~ Only affect/change the element they are added to.

Structural Directives

~ Look like normal HTML Attribute but have a leading \* (for desugaring)

~ Affect a whole area in the DOM (elements get added/removed)

~ You can’t have more than one Structural Directive on the same element

**NOTE:** The directive unlike a component doesn’t have a view (it doesn’t have a template)

**Directive (Section 7-85)**

* In order to useit, you must import it through @angular/core

**How to use ngOnInit? (Section 7-85)**

**How to build a directive? (Section 7-85)**

**NOTE:** It is not a good practice t o access your elements directly.

**How to use renderer2? How to setStyle? (Section 7-86)**

**setStyle(***element***,** *style* ex: ‘background-color’, *value you want to assign* ex: ‘blue’, *flags object* (optional) **)**

**(Section 7-86)** talks about why the basic way of building a directive is inefficient. The are enviroments where you might not have access to the DOM, so if you try to change the DOM while building a basic directive (by directly accessing the NativeElement and the style of the element), you might get an error in some circumstances. A good practice is to render for the DOM access.

**(Section 7-87)**

In the last lecture, we used the Angular Renderer class to change the style of a HTML element. As explained in that lecture, you should use the Renderer for any DOM manipulations.

Of course, you can do more than simply change the styling of an element via setStyle(). Learn more about the available Renderer methods [here](https://angular.io/docs/ts/latest/api/core/index/Renderer2-class.html).

**How to use HostListner? @HostListener** lets you listen for events on the host element or component.

**(Section 7-88)**

**How to use HostBinding? @HostBinding** lets you set properties on the element or component that hosts the directive

**(Section 7-89)**

**NOTE:** We can bind properties of oour own directives by simply placing them on the same element. Angular checks our own directives before it reaches the custom (native properties)

**Details on why we use \* before structural directives** ex: \*ngIf. **(Section 7-91)**

**How to build a Structural Directive? (Section 7-92)**

**How to use a setter Section? (7-92)** *~ A method that gets executed when the property changes.*

**NOTE:** Use ***class="list-group-item"*** in a list tag for some css bootstrap.

**How to use ngSwitch? (Section 7-93)**

Ex:

<div [ngSwitch] = "value">

<p \*ngSwitchCase="5">Value is 5</p>

<p \*ngSwitchCase="10">Value is 10</p>

<p \*ngSwitchCase="100">Value is 100</p>

<p \*ngSwitchDefault>Value is Default</p>

</div>

**What are services?** Acts as your centeral repository. As a centeral business unit. **(Section 9-95)**

**Hierarchical Injector (How to provide services? Section 9-98 & Section 9-100)** Services are meant to provide methods that will be used repeatedly. Instead of copying the code, can outsource it into a service. The angular dependency injector is actually a hierarchical injector. That means if we provide service in some place of our app, let’s say on one component the angular framework knows how to create an instance of that service component and important child components. That means this component and all the child components and the child components of the child components will receive the same instance of the service.

**NOTE:** We can inject services into services.

**Hierarchical Injector** (effects on where it is applied):

AppModule: Same instance of Service is available *Application-wide.*

AppComponent: Same instance of Service is available for *all Components* (but *not for other services*)

Any other component: Same instance of Service is available for *the Component and all its child comonents.* (**NOTE:** this can overwrite a service that was stated at a higher level.)

**How to use different instances of the service?** Review with Emmanuel. **(Section 9-101)** Good exmple on 2:13 **(Assignment 5 Instructor example)**

**How to inject a service into a service? (Section 9-102)**

**NOTE:** If you inject a service into something, the something must contain meta data. A component has meta data through @Component. A directive has meta data through @Directive. A service does not contain meta data. You have to attatch meta data to a service. You use attach @Injectable to services; it must be imported from @angular/core. You use @Injectable at the receiving service. **(Section 9-12)**

If you're using **Angular 6+** (check your package.json  to find out), you can provide application-wide services in a different way.

Instead of adding a service class to the providers[]  array in AppModule , you can set the following config in @Injectable() :

1. @Injectable({providedIn: 'root'})
2. export class MyService { ... }

This is exactly the same as:

1. export class MyService { ... }

and

1. import { MyService } from './path/to/my.service';
3. @NgModule({
4. ...
5. providers: [MyService]
6. })
7. export class MyService { ... }

Using this new syntax is **completely optional**, the traditional syntax (using providers[] ) will still work. The "new syntax" does offer one advantage though: Services **can be loaded lazily**by Angular (behind the scenes) and redundant code can be removed automatically. This can lead to a better performance and loading speed - though this really only kicks in for bigger services and apps in general.

**HUGE NOTE: (Section 10-107)**

getRecipes() {

return this.recipes.slice(); *//slice with no argument will return a new array, //which is an exact copy of the current array (recipes)*

}

**How to use subrscribe() (Section 10-108, 4:10)**

**How to use spread operator?** Helps spread an array into single units. **(Section 10-112, 5:20)**

**Routing (Section 11)**

In our app, we got three sections (referse to **routing-start**):

* Home
* Servers: - View and Edit Servers - A Service is used to load and update Servers
* Users - View Users

This app will be improved by adding routing but definitely feel free to play around with it - besides routing, everything should be working fine.

**How to import Routes? (Section 11-116) NOTE:** Section 11-122 tells us how to add parameters to Routes.import { Routes } from '@angular/router';

const appRoutes: Routes = [

{ path: 'users', component: UsersComponent } // when added it will be seen as ex: localhost:4200/users

]; //each route is a javascript object in the array

**NOTE:** forRoot() allows you to register some routes for the main application

**How to use** <router-outlet></router-outlet>? This directive marks the place in your document where you want angular to load the component currently selected route. (Section 11-116 7:20)

**How to have one page click to another:** After you set up the app-routing.module (set up the paths), use routerLink (Section 12-117, 1:56).

ex:

<li routerLinkActive="active"><a routerLink="/recipes">Recipes</a></li>

<li routerLinkActive="active"><a routerLink="[‘/users]">Users </a></li>

Another ex

<li role=”presentation> <a [routerLink]="/shopping-list">Shopping List</a></li>

<a [routerLink]="[‘/users’, 10, ‘Anna’]">Shopping List’]</a>

*SideNote:* Using “/users” is an absolute path ~ http://localhost:4200/users

Using “users” is a relative path ~ <http://localhost:4200/users/users>

You can use “../users” to go up a level and then place “users”

More details about this can be found on (Section 12-118)

**Styling Active Router Links** (Section 112-119)

**How to implement navigation?** Use routerLink=”/”. This will tell anuglar; that the element which the router link is placed will serve as a link (refers to string inbetween “/ ”), but it will handle the link differently. (Section 11-117, 2:00) The different paths you can use. (Section 11-118, 4:00)

**How to use** exact **&** routerLinkActiveOptions? (Section 11-119, 4:26) Ex: [routerLinkActiveOptions]="{exact: true}"> *//is true only if the link we are using is exactly like the link in routerLink=”/”*

<a routerLink="/">Home</a></li>

**How to navigate programatically?** (Section 11-121)

**NOTE:** Unlike routerLink; the navigate method doesn’t know which route you are currently on. The router link always knows in which component it sits in which components template, and therefore it knows what the currently loaded route is. (Section 11-121, 2:30, tells how to inform the navigate method of the current loaded route. You will need to import Activated route at @angular/router)

**How to use snapshot and params** (“Fetching Route Parameters Reactively”, Section 11-124)

**NOTE:** Observables are a feature added by some third party package not by a angular, but heavy used by angular, which allow you to work with asynchronous tasks. An observable is an easy way to subscribe to some event which might happen in the future to then execute some code when it happens without having to wait for now. **How to subscribe?** (Section 11-124, 4:30) **How to import Subscription** Section 11-125, 1:30, ex: import { Subscription } from 'rxjs/subscription';) **How to unsubscribe?** (Section 11-125, 2:00)

**NOTE:** Angular cleans up a subscription (behind the scenes) you set up whenever the component is destroyed, doing this lets the subscription live on in memory even if you left the component and later come back(leaving the component causes it to be destroyed; when coming back, a new one will be created). **How to use OnDestroy?** (Section 11-125) (*Angular cleans up subscriptions by default, but you have to manually do this for observables you created*)

**How to pass query parameters and fragments?** (Section 11-126)

**How to use queryParams and fragment properties?** (Section 11-126, 3:00, 3:50)

**How to retrieve Query Parameters and fragments?** (Section 11-127)

**Common Gatchas** (Section 11-128)

**How to set up child (nested) routes?** Child routes need a separate outlet from the parent outlet, because they can’t override the parent outlet. Instead the child route should be nested in the parent route.(Section 11-129)

**NOTE:** Use + in front of variables to make sure the variable is viewed as a number.

**HUGE NOTE:** How to catch all possible routes? Use this wild card ‘\*\*’. (Section 11-132, 3:20)

**Important: Redirection Path Matching**

**Section 11, Lecture 133**

In our example, we didn't encounter any issues when we tried to redirect the user. But that's not always the case when adding redirections.

By default, Angular matches paths by prefix. That means, that the following route will match both /recipes  and just /

{ path: '', redirectTo: '/somewhere-else' }

Actually, Angular will give you an error here, because that's a common gotcha: This route will now **ALWAYS** redirect you! Why?

Since the default matching strategy is "prefix" , Angular checks if the path you entered in the URL does **start with the path** specified in the route. Of course every path starts with ''  (Important: That's no whitespace, it's simply "nothing").

To fix this behavior, you need to change the matching strategy to"full" :

{ path: '', redirectTo: '/somewhere-else', pathMatch: 'full' }

Now, you only get redirected, if the full path is ''  (so only if you got NO other content in your path in this example).

**NOTE:** Typically if you have more than two or three routes, you don’t add it in the app.module.ts. Instead you add a new file, which is for the application as a whole. Typically is called “app-routing.module.ts”.

How to use CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, & Observable? (Section11-136)

How to protect child (nested) routes with CanActivateChild? (Section 11-137)

Authentification checks (Section 11-138)

How to use CanDeactivate (Section 11-139)

How to make an optional argument? How to implement interface? (Section 11-139, 5:50, ex: nextState?: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean; , 5:50 )

**Types of guards:** CanActivate, CanActivateChild, CanDeactivate.

How to get static or dynamic data once a route is loaded? (Section 11-140)

How to use resolver? (Section 11-141, 9:20)

Understanding Location Strategies. How to get code to work on older servers. (Section 11-142)

Details on how to use forRoot() (Section 11-142)

Routes are basically just an array of objects where each object represents a route.

How to use pathMatch (overrides the default of prefix) for setting up routes (Section 12- 145, 6:30)

You can make a hard copy of an object with object assign.

There's one thing I forgot to clean up here (will be cleaned up later in the course). Feel free to do the cleanup right now though.

Our app.component.html file looks like that:

1. <app-header (featureSelected)="onNavigate($event)"></app-header>
2. <div class="container">
3. <div class="row">
4. <div class="col-md-12">
5. <router-outlet></router-outlet>
6. </div>
7. </div>
8. </div>

The (featureSelected)="..."  event listener is a relict of our "old" navigation approach using ngIf. We no longer need it, so feel free to change this template to:

1. <app-header></app-header>
2. <div class="container">
3. <div class="row">
4. <div class="col-md-12">
5. <router-outlet></router-outlet>
6. </div>
7. </div>
8. </div>

Observable: Various Data Scources (Section 13-159) ex: (User Input) Events, Http Requests, Triggered in Code, …

Observer (You write this part, which gets executed): Three ways to handle data packages~ Handle Data, Handle Error, Handle Completion. NOTE: Observable doesn’t have to complete. (Section 13-159)

~ Imports needed to use Observables. (Section 13-162)

import { Observable } from 'rxjs/Observable';

import 'rxjs/Rx';

~ Creating an observable from scratch (Section 13-163)

.interval

.create

.next emmits a normal data package

~ Memory Leaks (Section 13 – 164, 0:47)

~ Observables built into angular clean up themselves, but it is a good practice to clean up all observables (so you don’t forget to clean up an observable where no automatic clean up is provided).

~ For more info on rxjs (observables)

<https://rxjs-dev.firebaseapp.com/api>

~Subject (Section 13-166)

Simlar to an observable, but you can use it to emit data.

Subject is observable and observer at the same time. Don’t forget to unsubscribe.

~ Use Subject for cross component communication instead of the Event Emmiter.

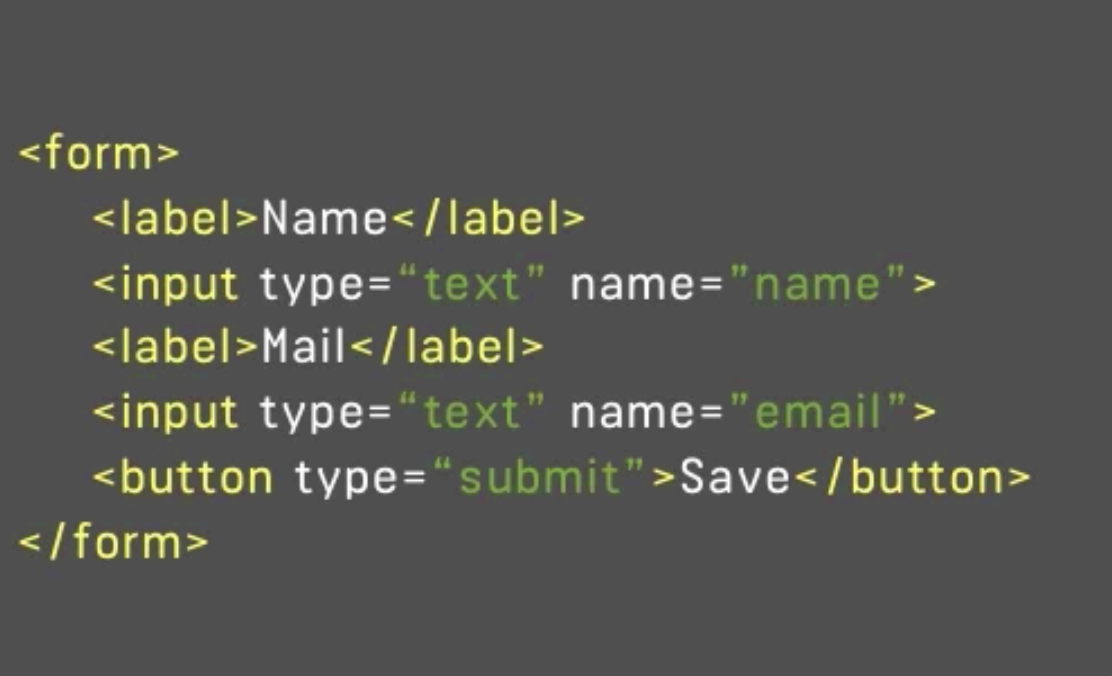
~ Maps (.maps() You must import 'rxjs/Rx' in order to use it):

simply maps the data you get back into a new observable with any transformations of your choice.

~Since Operators simply return new Observables, you can of course also chain these Operators! (Section 13-167)

~ RxJS 6 without rxjs-compat (Section 13-168)

~Example of form:



~**Angular offers two was to approach forms** (Section 15-173):

- Template driven approach – Angular infers the Form Object from the DOM.

- Reactive – Form is created programmatically and sychronized with the DOM.

~ To make use of Form functionalities, import FormsModule at app.module.ts

(Section 15-175)

ex:

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

imports: [

FormsModule

],

~ When using ngModel in a form tag, you are stating to angular that this input is a control of my form. You must include a name for the control in order to use the control. You can use this aproach for input and select tags.

* ngModel is used without binding or with one-way binding, depending on what you are using it for.
* It is still possible for you to use two-way binding on ngModel. An example is found here (Section15-184)

[(ngModel)]="variable" <!— ngModel will grab the user’s input and save it into variabe. You can then use string interpolation to print variable —>

~ If you want to run an event when a user clicks submit, use (ngSubmit) on the form tag. (Selection 15-176, 1:37)

~ ngForm tells angular to give you access to the form it created automatically. (Section 15-176, 4:28)

~NgForm object (values within this object):

* dirty – field must be changed in order for this value to be true
* touched – value will be true if form registers a click.

~How to validate user e-mail? (Section 15-179 through 15-183)

* Use required directive in the input tag to mark if the tag is empty.
* Use email directive in the tag to mark if the tag has a valid email.

Which Validators do ship with Angular?

Check out the Validators class: <https://angular.io/docs/ts/latest/api/forms/index/Validators-class.html> - these are all built-in validators, though that are the methods which actually get executed (and which you later can add when using the reactive approach).

For the template-driven approach, you need the directives. You can find out their names, by searching for "validator" in the official docs: <https://angular.io/api?type=directive> - everything marked with "D" is a directive and can be added to your template.

Additionally, you might also want to enable HTML5 validation (by default, Angular disables it). You can do so by adding the ngNativeValidate  to a control in your template.

~ To group inputs together use ngModelGroup (Section 15-185)

- You can use the **required** directive to force the user to click on something before being allowed to submit their form. (Section 16-186, 2:56)

**How to use .setValue() and how to use .patchValue()** (Section16-187)

~.setValue() changes a form according to it’s specific value.

.patchValue() only changes parts of the form that have been specified.

- How to get ngModel to work (Section 15 Assignment 6, 1:00)

- form-control is a bootstrap class that is merely to make the form look better.

**Reactive Form** approach (16-191)

-Made programatically (this does not mean it’s made from scratch)

-You don’t need the FormsModule to make a reactive form.

-You must import ReactiveFormsModule from ‘@angular/forms’ (Section 16-193)

-Synhing HTML and Form (Section 16-193)

-import { **FormGroup** } from “'@angular/forms';

~This imports form classes as well as ngForm.

~You can have FormGroup inside a FormGroup (Section 16-197)

- **Adding validation to Reactive Form**.

Import Validators from ‘@angular/forms’ (Section 16-195)

-**How to use formControlName** for elements in an array. (Section 16-198, 6:25)

-Creating custom validators (Section 16-199)

-Reactive Form, creating a custom asycnronous validator. (Section 16-201)

-**NOTE**: You can pass an object to reset() to reset to specific values.

- When providing a service to a component, all the components within that component share the same instance. When navigating away from these components, the previous component is destroyed, destroying the instance of the service. This causes a loss in memory for the program. Fix this by providing the service in app.module.ts. Remember to import it as well.

- Pipes are a feature built in angular 2, which allows you to transform output in your template.

Ex. <p>{{ username | uppercase }}</p>

- Using pipes and how to use parameters. (Section 17-227-229)

Ex: pipe parameter

{{ server.started | date:'fullDate' }}

NOTE: You can add a second parameter by adding a colon and a second parameter. Ex: …‘fullDate’:

NOTE: You can add a second pipe by adding another | after a parameter.

Ex: …date | uppercase

- More details on pipes here

https://angular.io/api?query=pipe

- Details on how angular processes pipes. It is read from left to right. (Section 17-229)

- You can force angular to update its pipes whenever data is changed, however, this might lead to performance issues (updating large lists will cause the performance issue). (Section 17-233)

NOTE: pure is true by default. Changing this to false will cause angular to update whenever data (anything) changes.

- async is a pipe that helps with handling asychronous data. Data that transforms into something else. This is needed when wanting to catch changing data, because angular by default will output the old data in order to save performance.

- How to reverse a string. (Section 17-Assignment 8, 0:37)

(Section 18-235)

**Angular 6**is currently the latest version of Angular and it **deprecates** the Http-access method taught in this module.

**What does this mean?**

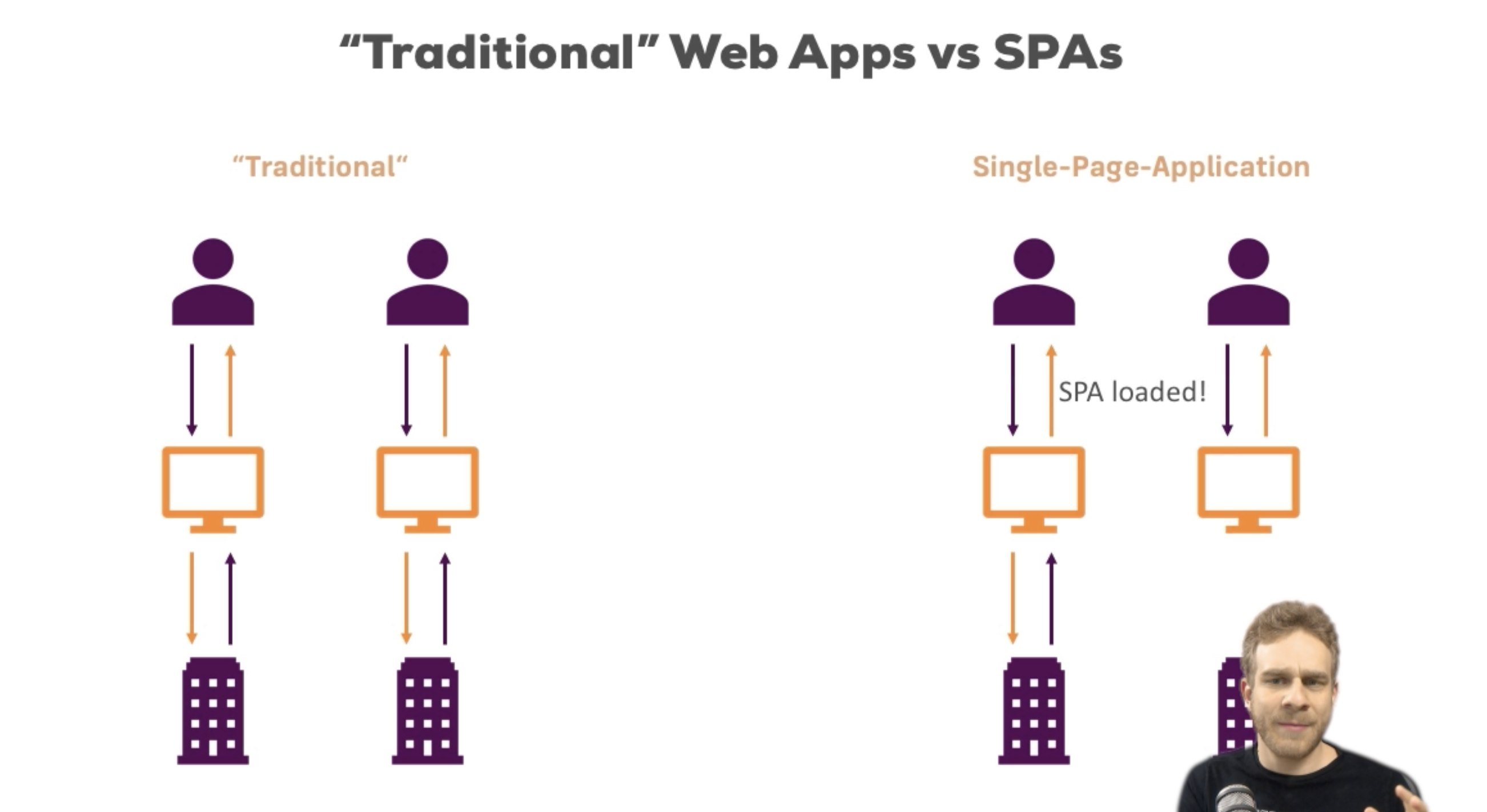
It means that the method **still works**, still is secure - you can use it! But there is a better Http module to use now: **HttpClient**.

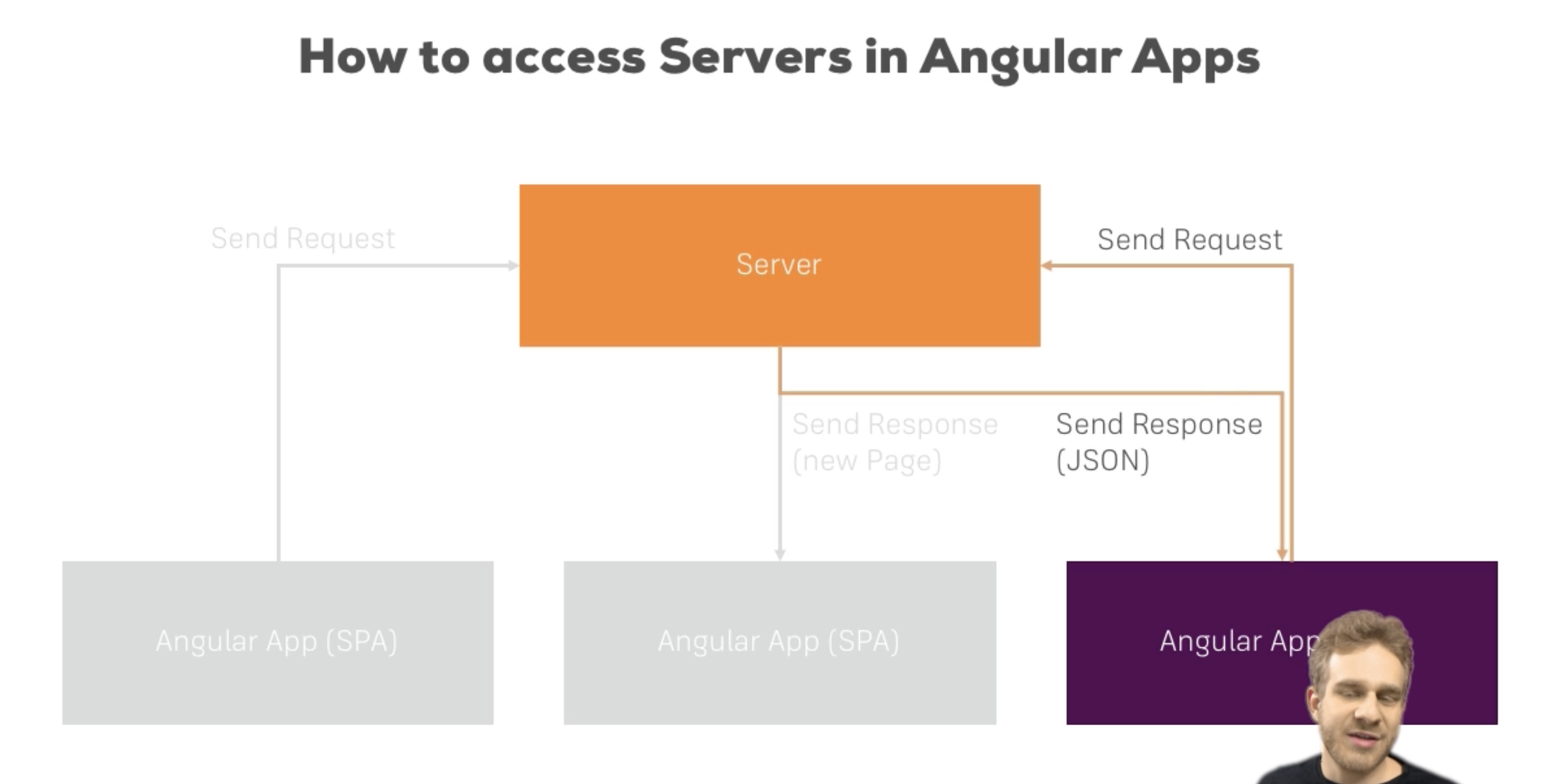
I added a **module (section 23)** on that new client months ago, even before Angular 5 was released. You'll meet it later in the course and we'll **easily update all our Http calls** with the new client there.

So for now, **follow along with this module here** - the core concepts taught here will still apply (i.e. how it works etc).

And later in the course, we'll revisit this solution and update it to HttpClient.

How to make Http requests in an angular app. (Section 18-236)

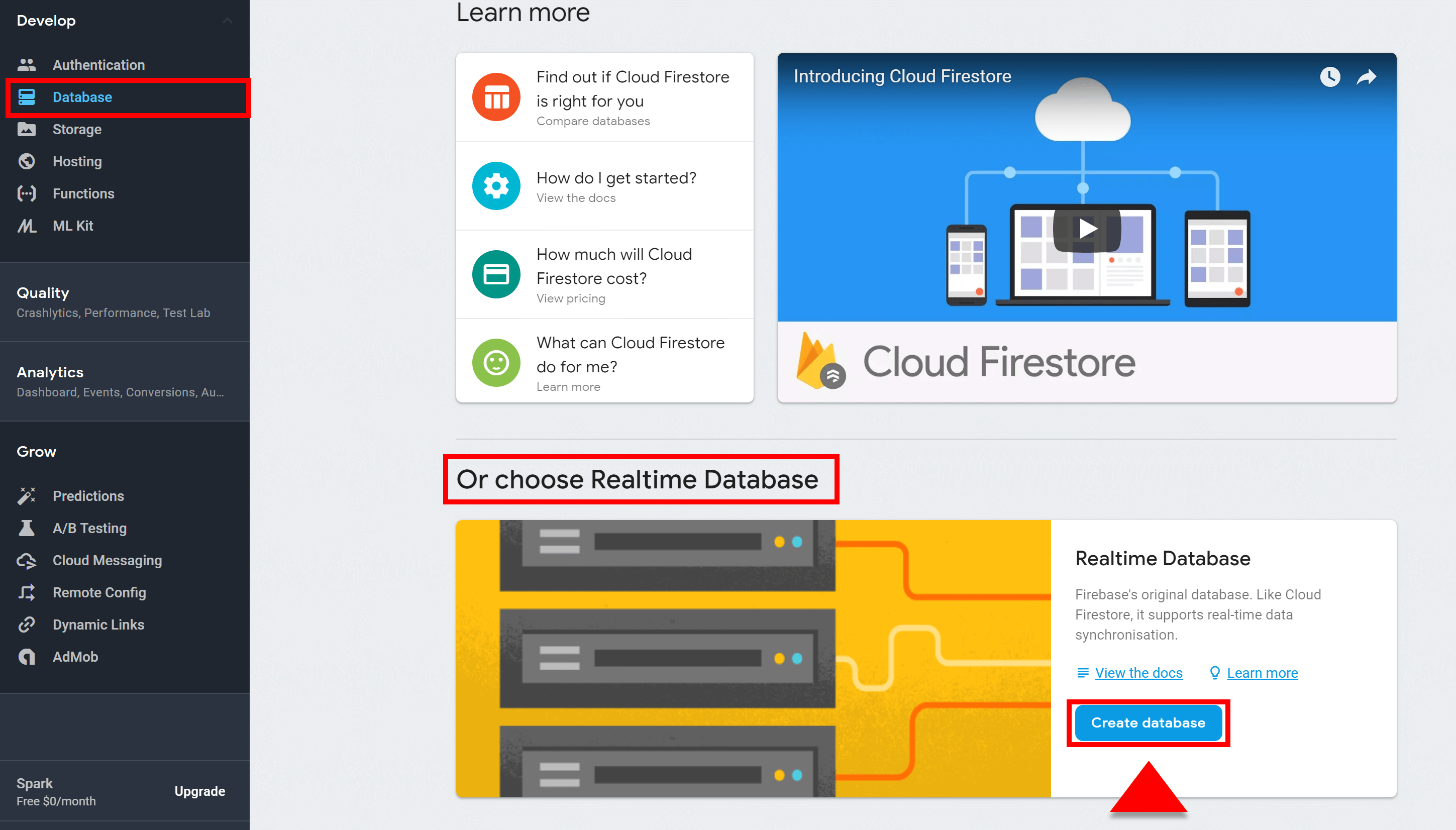




Firebase & The Right Database

(Section 18-237)

The Firebase Database console - which we'll see in the next lecture - changed visually.

**Important:** Make sure you pick the **Realtime Database**, NOT Firestore!

- **Note:** How to use Math.random()

ex in (Section 18-239, 0:19)

You can find this ex in better detail in “http-start” file app.component.ts.

- You can send Http requests from any place in your app, but typically you create a service to centralize this task.

- An Injectable is required if you plan to inject a service into a service.

Http requests:

* post() ~ appends to any existing elements (often used; stores data)
* put() ~ overrides existing elements

- As long as you don’t subscribe, no request gets sent. (Section 18-239)

- Adding data.json at the end of the url tells firebase that we’re about to work with it’s database. Otherwise you will get a course error. This is only firebase specific. (Section 18-239, 7:30)

- Make sure you use import HttpModule in app.module.ts, otherwise using the Http Service will not be possible. (Section 18-239, 8:10)

- .json() method is used to extract data

RxJS 6 without rxjs-compat

(Section 18-243)

Don't forget - if you're using Angular (and therefore also RxJS 6+) and you're NOT using rxjs-compat  (npm install --save rxjs-compat  - you may ignore this lecture then, use the code as shown in the videos!), you have to use operators like map()  differently:

Instead of

....map(...)

use

....pipe(map(...))

map also needs to be imported:

Instead of

import 'rxjs/Rx';

use

import { map } from 'rxjs/operators';

- The .map operator takes the old observable and wraps/turns the data we get back into some transformed data, and then wraps that transformed data in another observable. (Section 18-244, 0:55)

- import 'rxjs/Rx'; (in app.module.ts)

This package is reponsible for observables and labeling. By adding this import we unlock all the operators.

Catching Errors without rxjs-compat

(Section 18-246)

Are you using Angular 6 (and therefore RxJS 6+) and you're NOT using rxjs-compat  (npm install --save rxjs-compat  - you may ignore this lecture then, use the code as shown in the videos!)?

You then have to use the catch()  operator you'll see in the next lecture a bit differently.

Instead of

1. ....catch(error => {
2. return Observable.throw(...)
3. })

use

1. ....pipe(catchError(error => {
2. return throwError(...)
3. }))

And make sure to import it:

Instead of

import 'rxjs/Rx';

and

import { Observable } from 'rxjs/Observable';

use

import { catchError } from 'rxjs/operators';

and

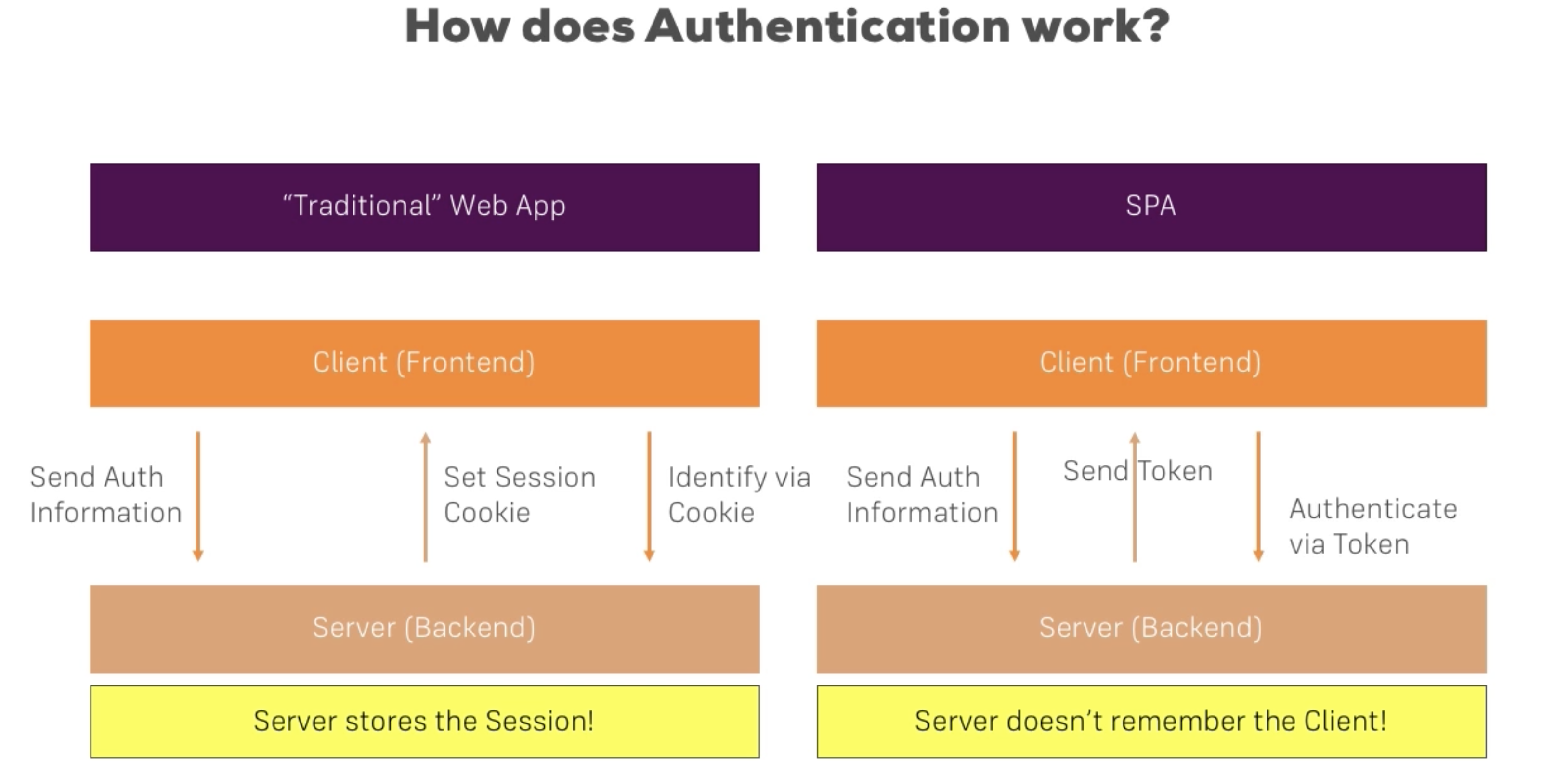
import { throwError } from 'rxjs';

- Typically you subscribe to a componenent and not in a service, because you may want to show something in the component such as an error.

Application Database:

https://muse-http-start.firebaseio.com/

How does Authentication work? (Section 20-255)



More about JWT

Section 20, Lecture 256

**Want to learn about the Token which is exchanged?**

The following page should be helpful: <https://jwt.io/> - specifically, the introduction: <https://jwt.io/introduction/>

In authentication, when the user successfully logs in using their credentials, a JSON Web Token will be returned. Since tokens are credentials, great care must be taken to prevent security issues. In general, you should not keep tokens longer than required.

How to download firebase? (Section 20-259)

npm install --save firebase

Import firebase through app.component.ts (Section 20-259, 3:14)

import \* as firebase from ‘firebase’;

Use getIdToken() instead of getToken()

Section 20, Lecture 260

Important note: If you're using**Firebase 5.x** or higher (you can check the package.json  file to find out), you should use getIdToken()  for obtaining the token, NOT getToken()  as shown in the next lectures.

To authenticate ourselves to the backend. We need to add a query parameter to the route. (The query parameter firebase recognizes is the .json?auth=” + token) )

(Section 20-264, 4:40)

Use getIdToken() instead of getToken()

Section 20, Lecture 260

Important note: If you're using**Firebase 5.x** or higher (you can check the package.json  file to find out), you should use getIdToken()  for obtaining the token, NOT getToken()  as shown in the next lectures.

**Angular Commands/packages ~**

-emmet

-TODO

comment Todo, Fixme, or Bug

ex:

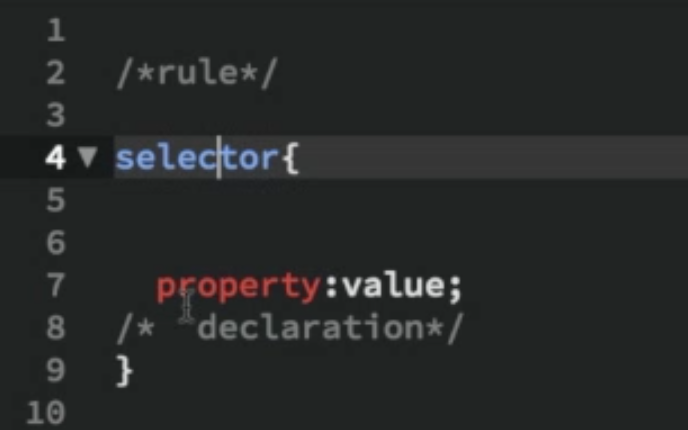
//Todo: You need to make a body here.

Command-shift-c: color picker

-atom-save-all

control+s

**CSS Tutorial for Beginners**

****

- The \* is used as a wild card. When using \*, you refer to every possibility.

Ex.

\* {

color: blue;

} /\* This means all text colors will be blue if not overwritten. Every class will implement this by default. This effect can also be used on routes \*/

- span is used to provide a different class for contents wrapped by an already existing class. This is used to make a unique look for the span contents.

- Example bellow shows you that you can target tags with specific classes.

Ex:

p.red {  
 color: red;

} /\* This only colors the text red for tags p with class red

Another Ex:

a[href] {

}

//Affects a tags with href. You can be more specific. Ex: a[href=“#”]

//Ex: a[href^=‘http’] affects href that starts with http

//Ex: a[href$=“html”] affects href that ends with html

Another Ex:

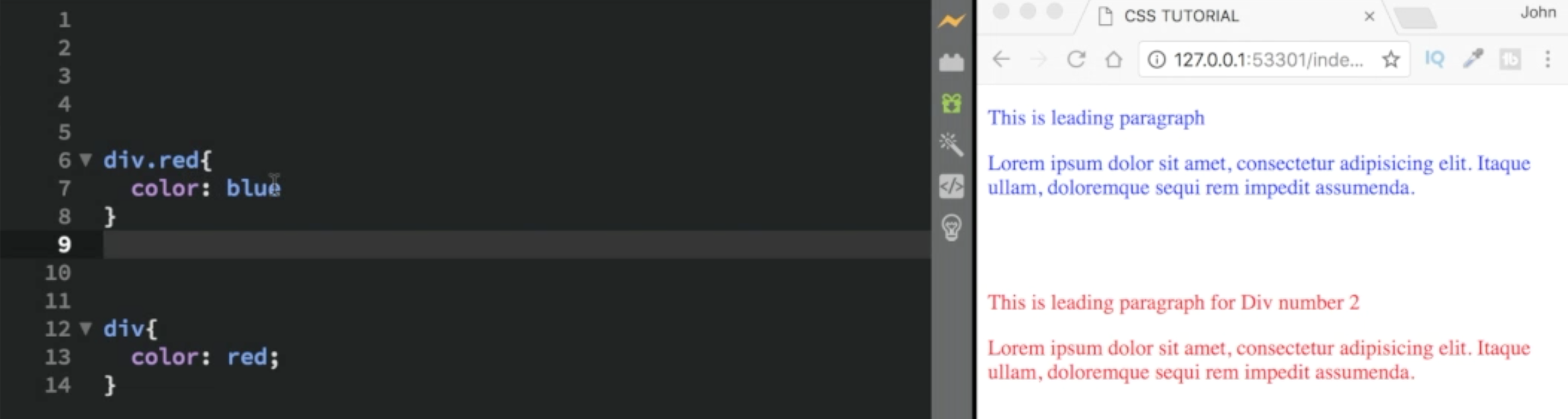
span[class] {

}

//Affects span tags with a class

\*/

- An id and class are similar, but an id should be unique, meaning they should be only affecting a particular tag. Also, you cannot use two ids in the same tag.

- The more specific you are when styling elements, the harder it is to overwrite it.

- rgb specifies color. rgba specifies color and opacity.

-Find suggestions of colors at colorhunt.co, materialui.co, colors.co

-px is an absolute value. em is a relative value. em is the default value. Normally the default value is 16px, but this can be changed in the body. 2em = 32px.

- If px is defined in a parent, the child containing em will be overwritten. If you want to keep the em property, use rem instead of em.

- vw and vh are tools you can use to size a tag relative to the window.

- To get custom fonts, visit **fonts.google.com**

- Font tools: font-style, font-weight, and font-size

- Text tools: text-indent, text-align, line-height, word-spacing, letter-spacing, text-transform, color, text-decoration

- Container tools: margin, border, padding (paddings of containers can overlap, but margins cannot)

-Display Tools: display, text-align, list-style-type, transition (hover), border-radius

- Tag input classes: type, placeholder

- Tag a classes: hover, active. Ex. a:hover{}

- Tag input classes: hover, focus.

- Target tag without class. Ex: p:not([class]){}

-Target tag without a particular tag. Ex: p:not(.special)

- Selectors: before, after, first-line, first-letter, target, first-child, last-child, nth-child, first-of-type, last-of-type, nth-of-type, only-child, only-of-type. Ex: p::before{}, Ex: nth-child(#n), Ex: nth-of-type(even or odd)

- background-repeat: repeat / no-repeat / repeat-x / repeat-y / space / round

- background-size: cover / size / auto (default)

- background-position: center / left / top/ right / bottom / x, y (coordinates by percent)

- background-attachment: fixed / inherit / local/ scroll (default)

- display: flex

- justify-content: center

- align-items: center

- Good UI details on (Section 1-43)

-shortcut:

original ~

background-size: cover;

background-position: center;

background-repeat: no-repeat;

background-attachment: fixed;

shortcut ~

background: url(small.jpg)center/cover fixed no-repeat;

- Gradients:

background: linear-gradient(to bottom (default) / to top / to top left/ #deg, color, color);

*DETAILS on how GRADIENTS can be HELPFUL~*(Section1-45)

- float: inherit / left / none (default) / right

- clear: both / inherit / left / none/ right

- position: absolute / fixed / inherit / relative (relative to parent container) / static / sticky

- Media Queries (Section 1-51)

- transform: translate() / scale() / rotate() / skew()

- transition-property: (Section 1-56 & 1-57)

- transition-duration: (Section 1-56 & 1-57)

- transition-delay: (Section 1-58)

- transition shortcut: