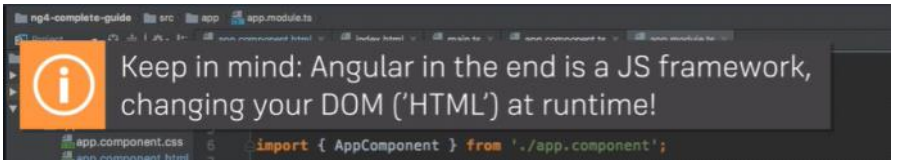
Angular

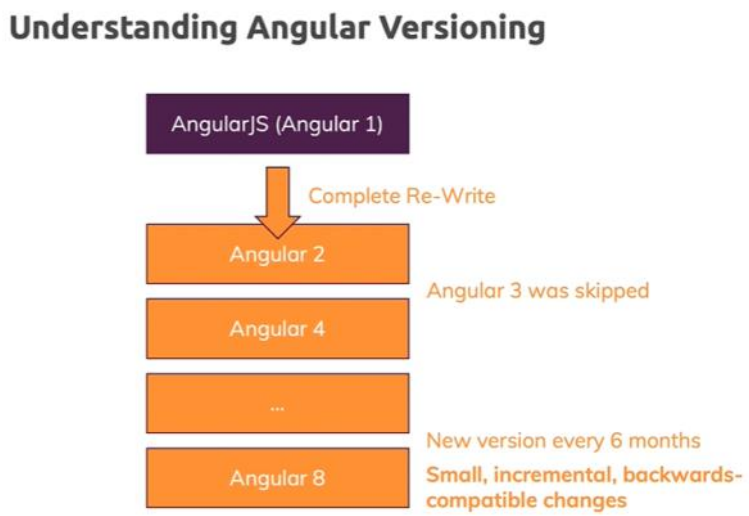
Monday, January 13, 2020 10:20 AM

Introduction

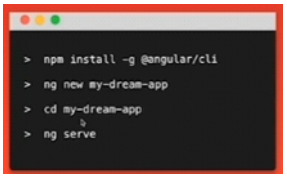
- Angular is a **JavaScript Framework** which allows you to create reactive **Single-Page-Applications** (SPAs).
- Changes the html code during run-time
- 

- Angular uses components to build web pages and modules to bundle up different pieces of app (Ex. components) to build packages

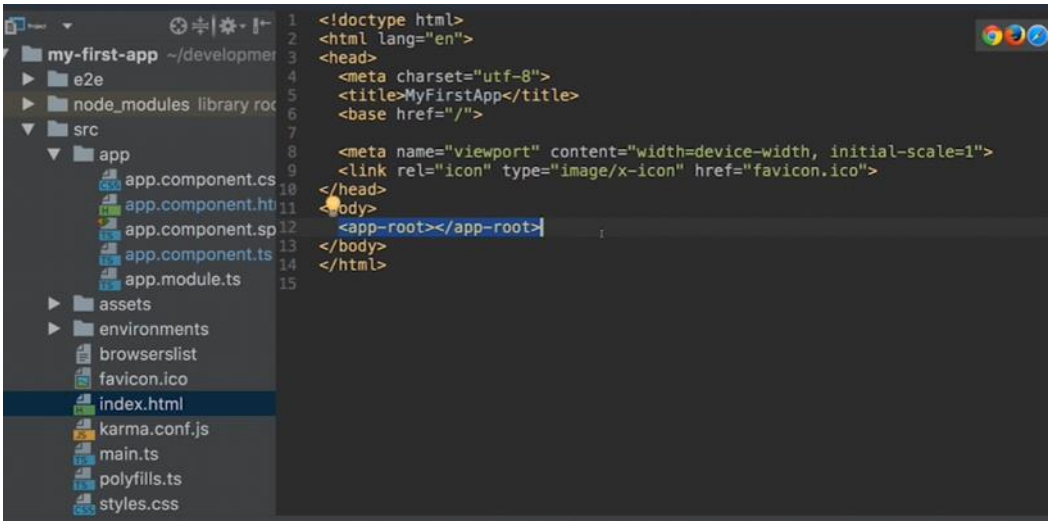
Angular Versioning

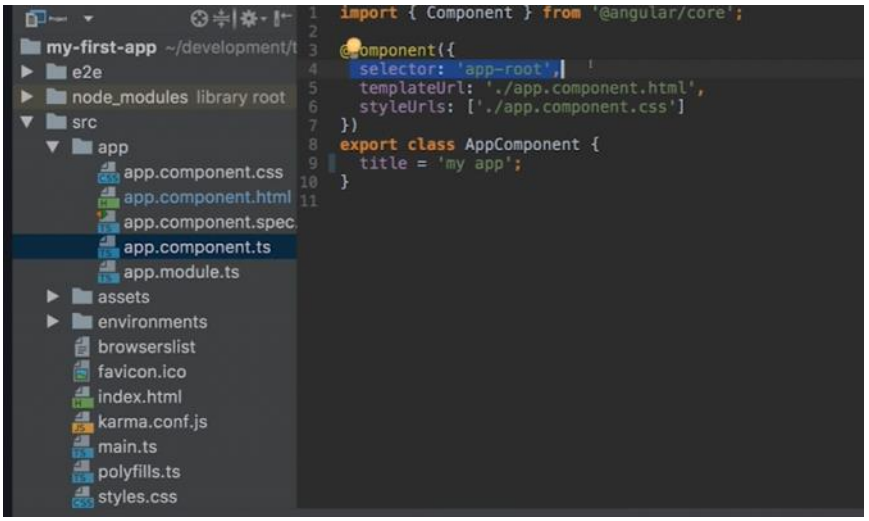
- 

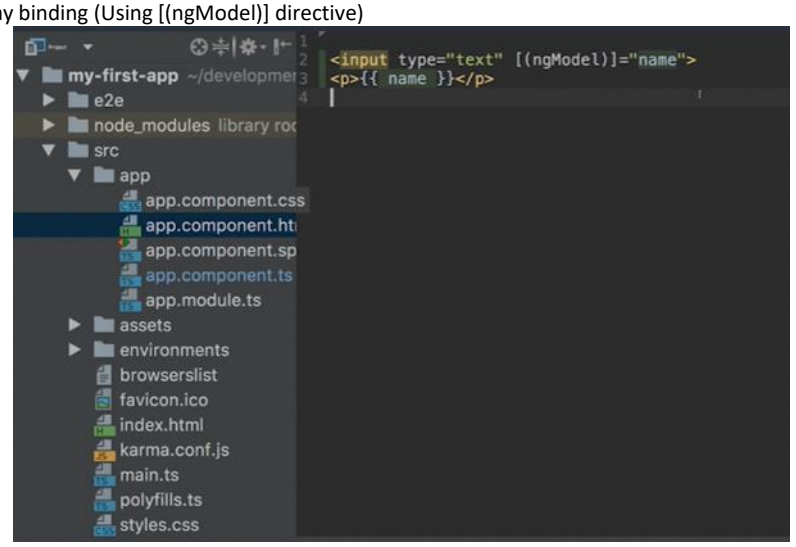
Angular Project Setup

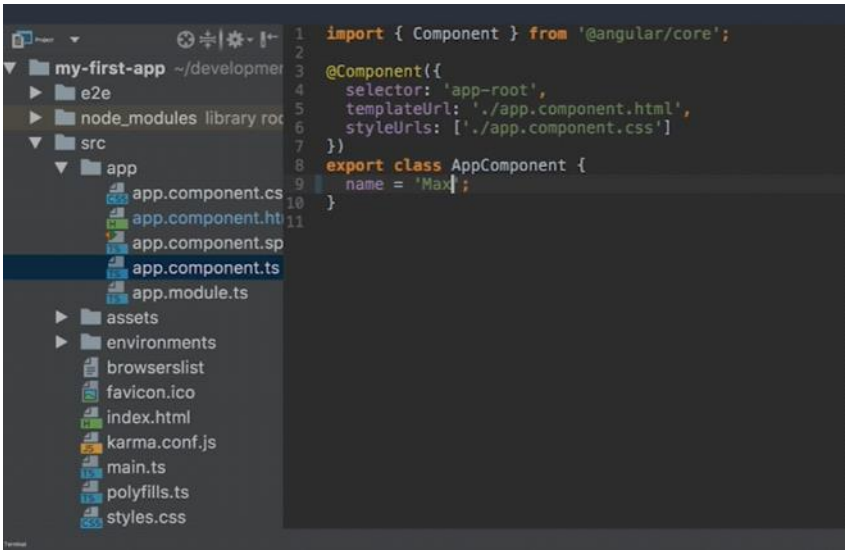
- 

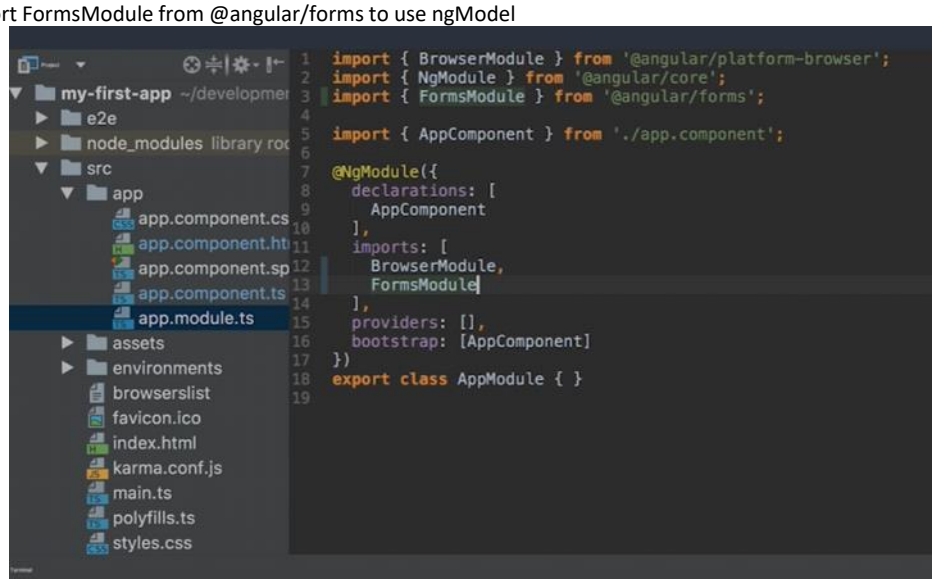
- App Root

- 

- 

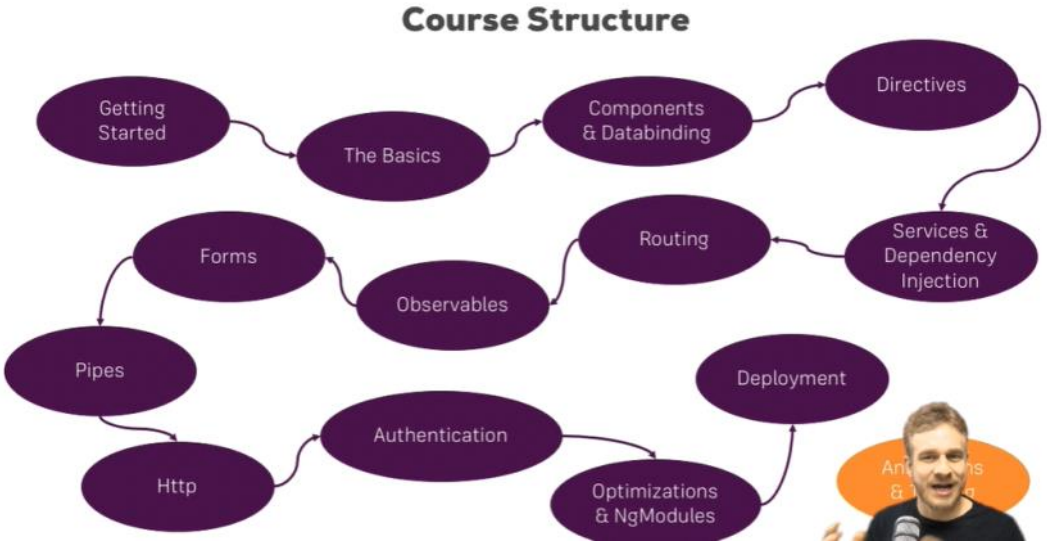
- 2-way binding (Using [(ngModel)] directive)
- 

- 

- Import FormsModule from @angular/forms to use ngModel
  - 

  - sub

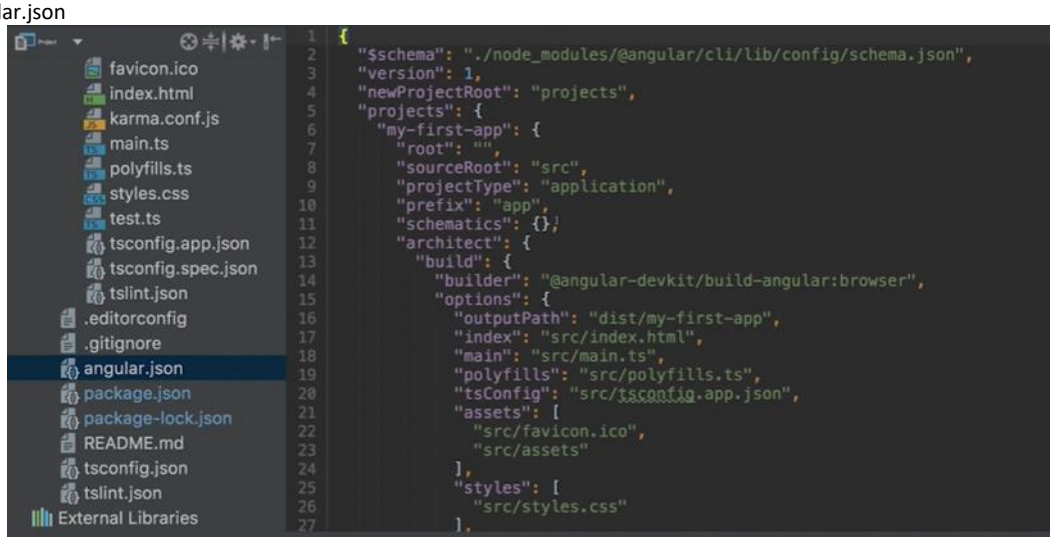
Course structure

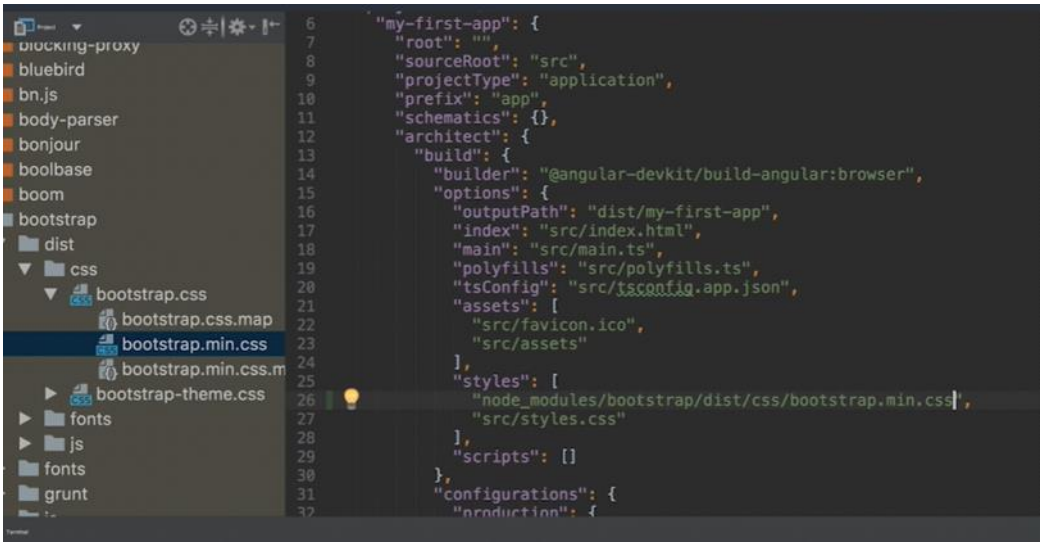
- 

Typescript

- Superset of js

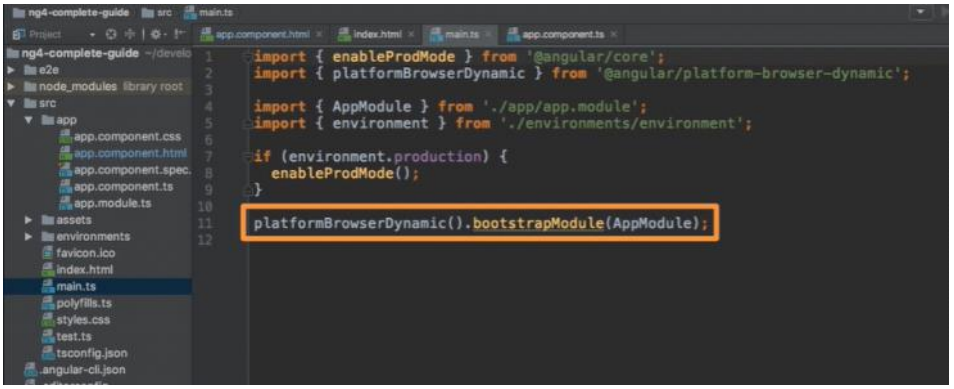
Angular project config

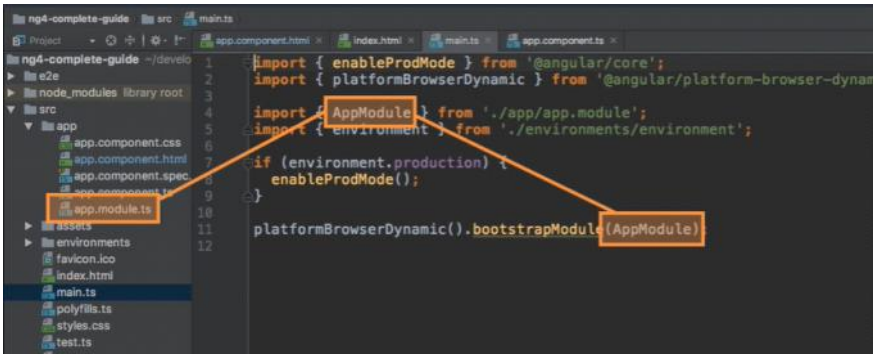
- angular.json
  - 

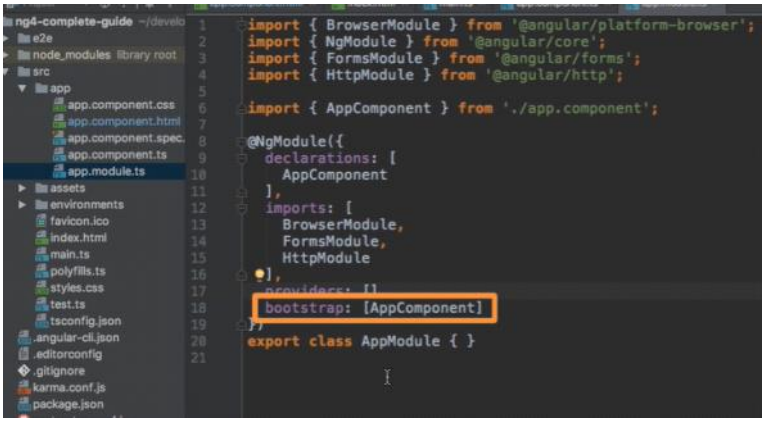
- To install bootstrap
  - npm install --save-dev bootstrap@3
  - 

Basics

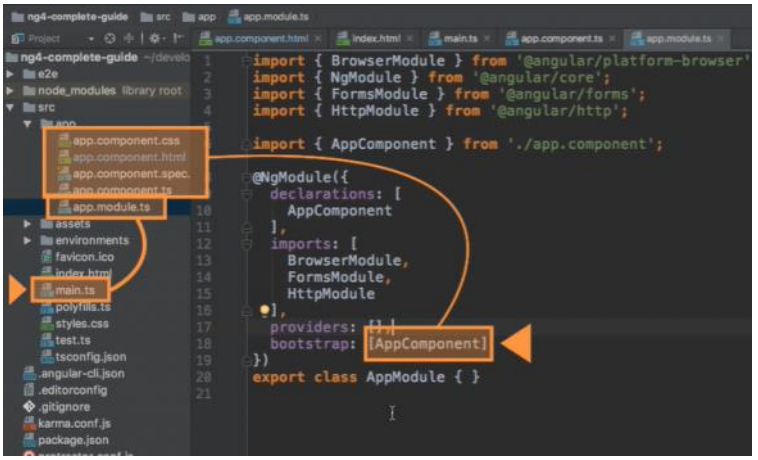
- Angular bootstrapModule

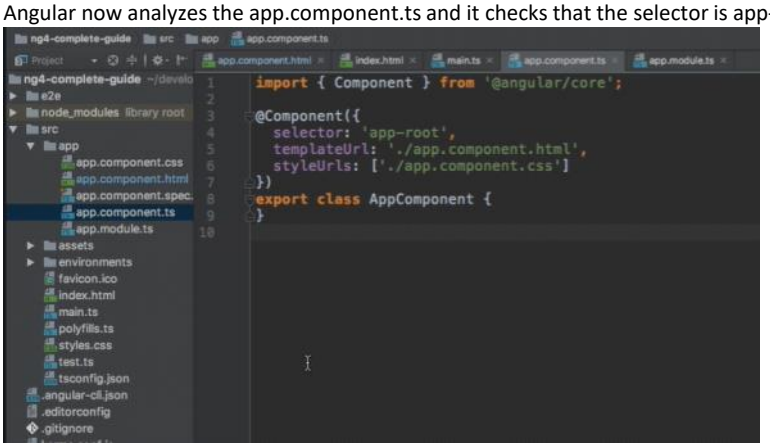
- 

- 

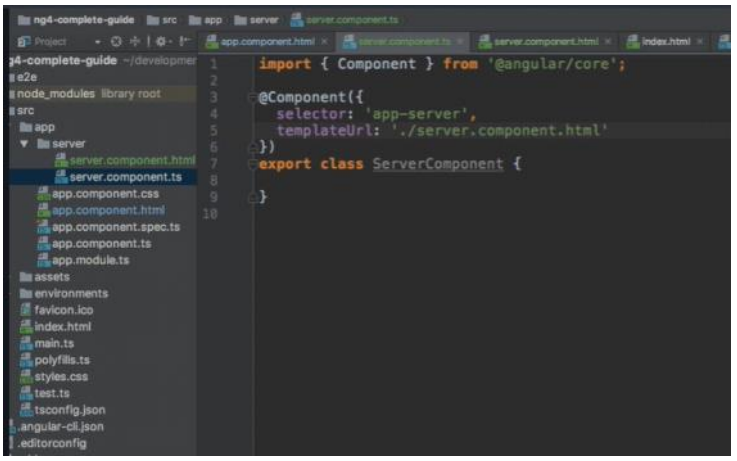
- 

- The above bootstrap array should know about all the components it is required to know when it loads the index.html
- So when Angular app gets started
    - It loads the main.ts file, where the bootstrapping of the angular app happens.
    - Then it sees the AppModule, loads the app.module.ts
    - In app.module.ts, it checks the component to be loaded which is AppComponent and renders it.

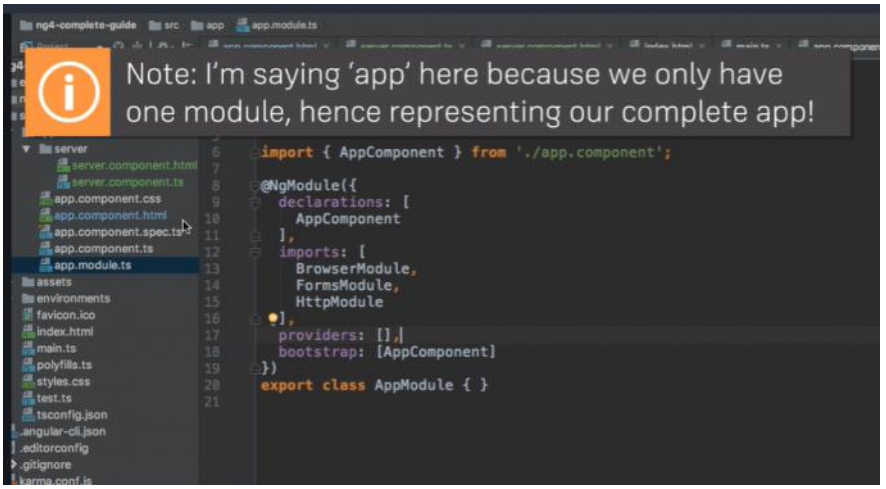
- 

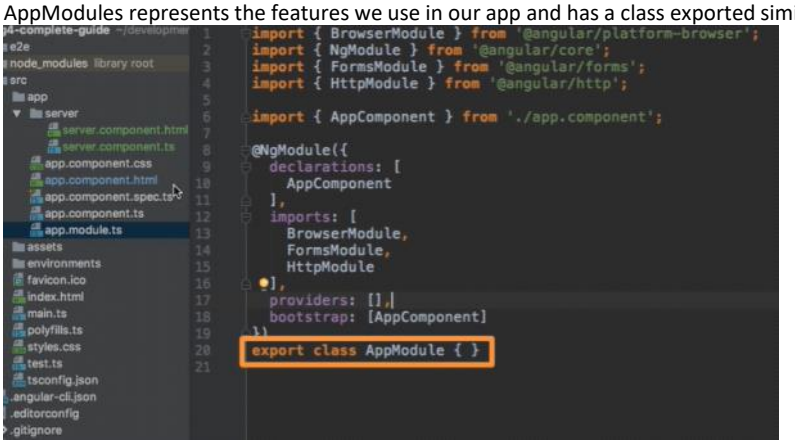
- Angular now analyzes the app.component.ts and it checks that the selector is app-root, recognizes it and loads in the index.html
- 

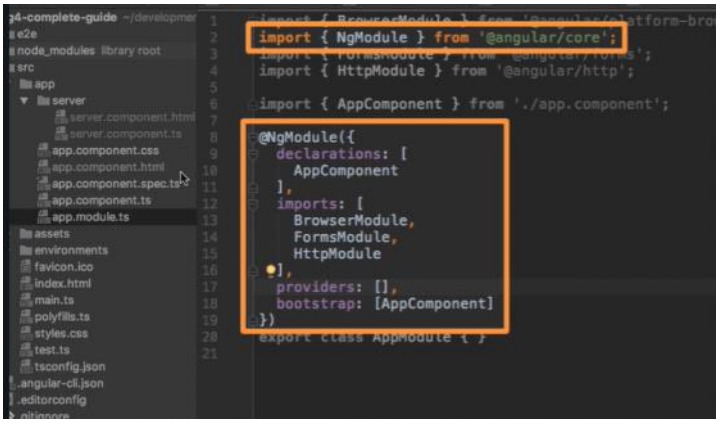
- Components
  - Components are important in angular
  - Creating a new component
    - Create a folder "<component-name>". Ex. server
    - Create a file "<component-name>.component.ts" Ex. server.component.ts
    - Component is just a class which Angular is able to instantiate it to create objects based on the blue print we set up here.
    -

    - Also we need to tell angular that it is not just a ts class, it is also a component, so we write component decorator to enhance the class
      - We need to import it from '@angular/core'
      - Add metadata to the object
      - selector: Should be a unique name prefixed with app-. Ex. "app-server". This will be used in other component html templates
      - templateUrl: relative path to the html file in the component folder Ex. server.component.html
      - 

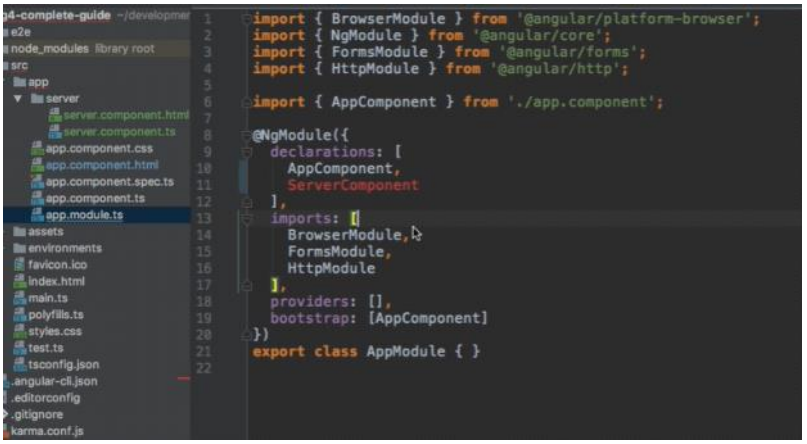
    - Registering in AppModule

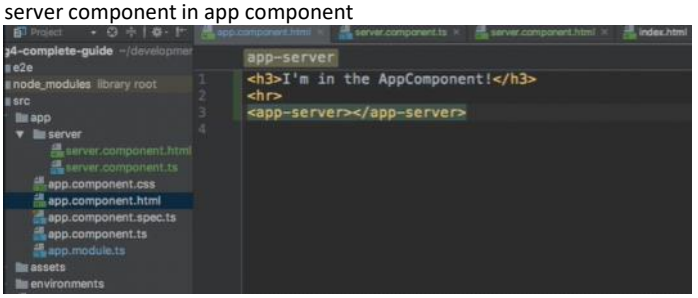
- 

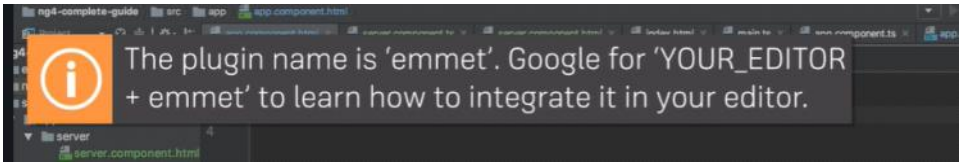
- AppModules represents the features we use in our app and has a class exported similar to our component.
- 

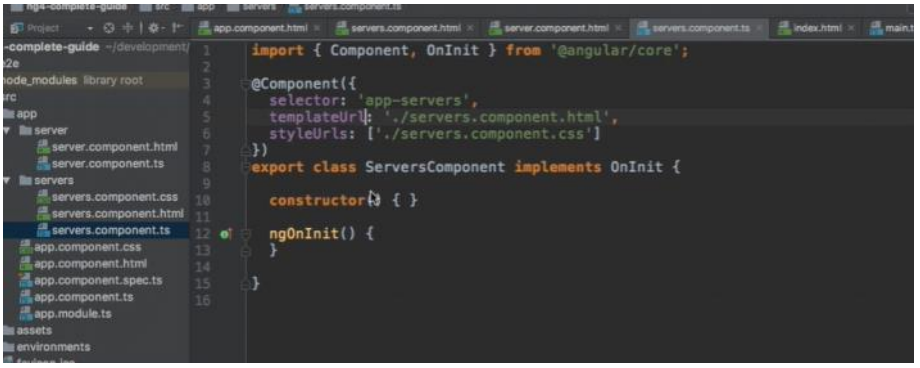
- 

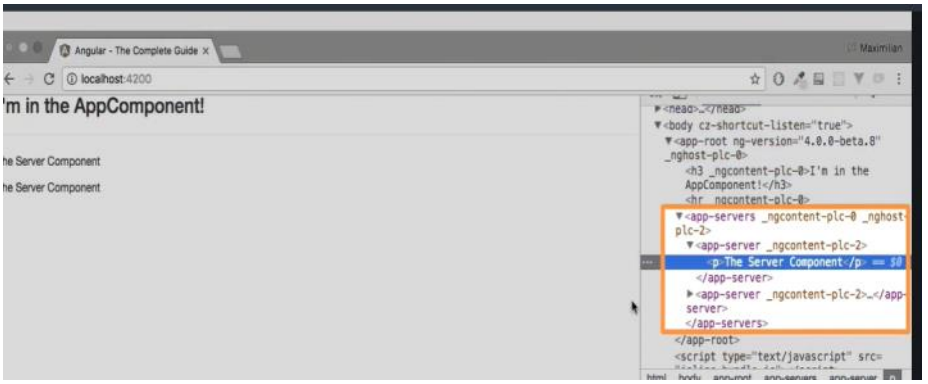
- And we have the NgModule decorator which is imported from '@angular/core' library.
- We have 4 properties
  - declarations: place where we register our newly created components by importing into the app.module.ts
  - imports:
  - providers
  - bootstrap: tells us which component to be held accountable for the app to load in the index.html

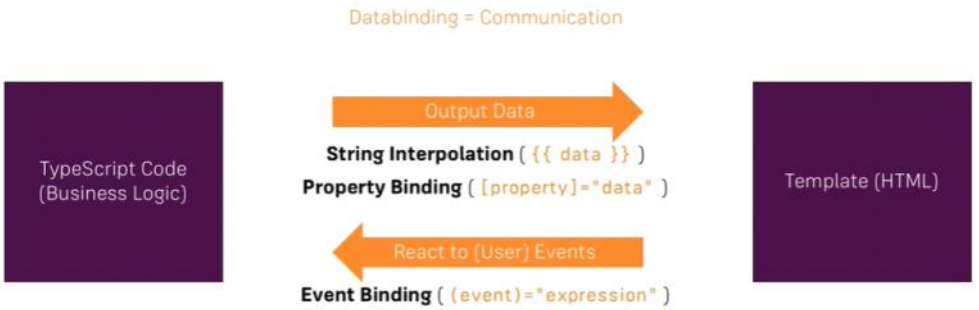
- 

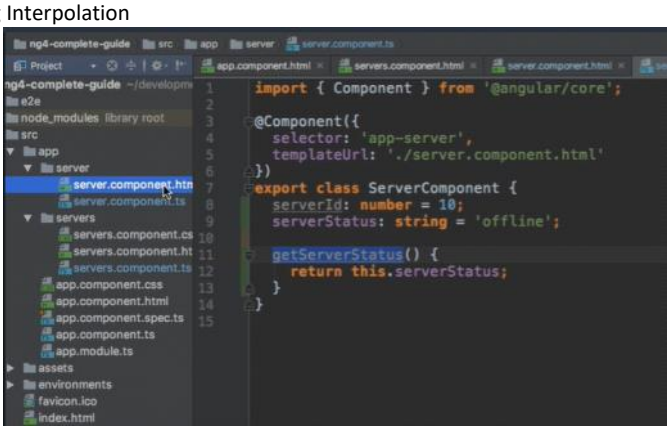
- Using server component in app component
  - 

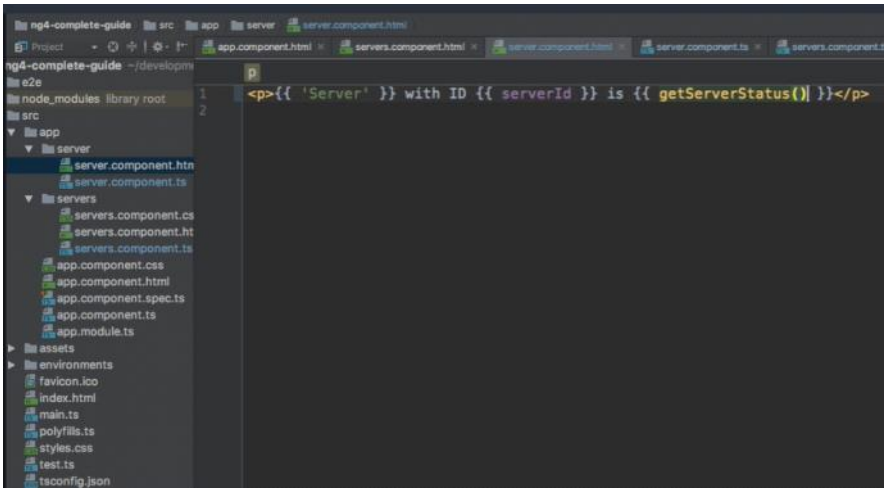
- 

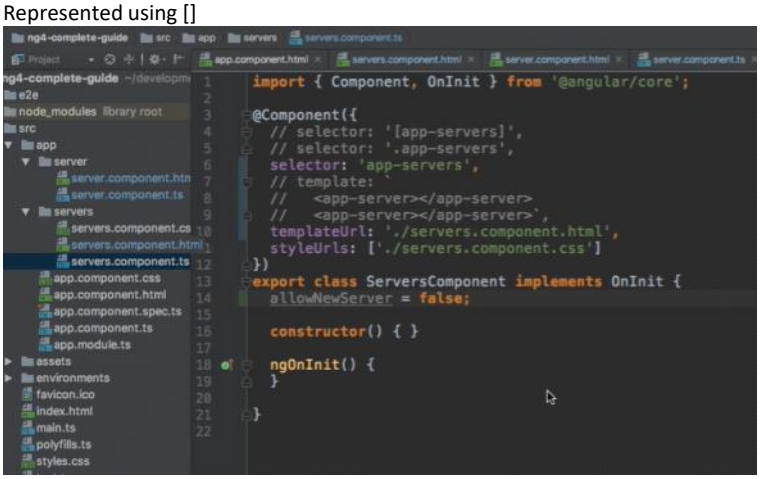
- We can generate a component using command line using the below command
  - ng g c servers --module=app
  - We have 3 component properties:
    - selector: component name to be used in the html as in the following ways
      - element -> Ex. selector: 'app-server' // Recommended
      - attribute -> Ex. selector: '[app-server]'
      - class -> Ex. selector: '.app-server'
    - template/templateUrl:
      - template: used to write html in the ts file as a string or template literals
      - templateUrl: relative path to the html to be loaded for the component
    - style/styleUrls:
      - style: used to write css in the ts file as an array of strings or template literals
      - styleUrls: array of the css files to be loaded for the component
  - 

- 

- Data-binding in components
  - Communication between template and the ts file
  - 

  - String Interpolation
    - 

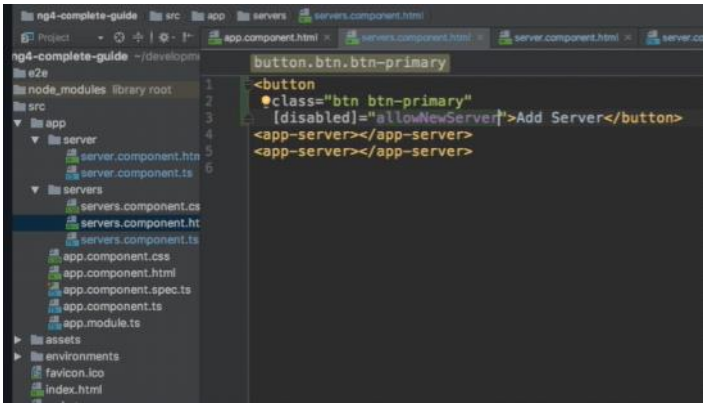
    - 

- Property Binding
  - Represented using []
  - 
```
import { Component, OnInit } from '@angular/core';

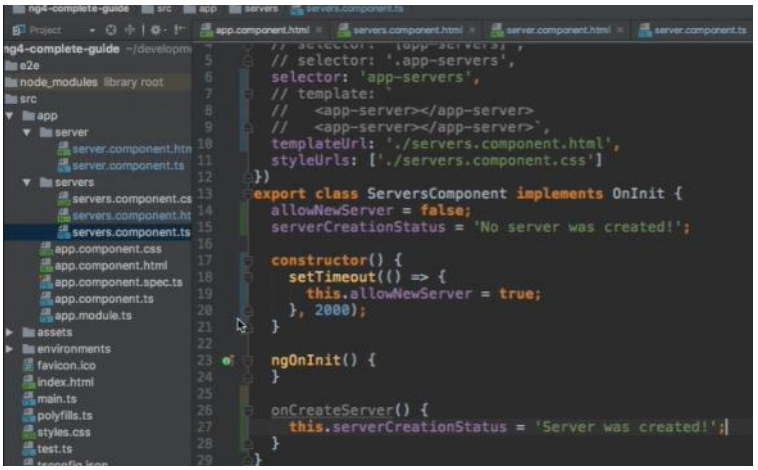
@Component({
  // selector: '[app-servers]',
  // selector: '.app-servers',
  selector: 'app-servers',
  // template: `
  //   <app-server></app-server>
  //   <app-server></app-server>`,
  templateUrl: './servers.component.html',
  styleUrls: ['./servers.component.css']
})
export class ServersComponent implements OnInit {
  allowNewServer = false;

  constructor() { }

  ngOnInit() {
  }

}
```
  - 
```
button.btn.btn-primary
<button
 class="btn btn-primary"
  [disabled]="allowNewServer">Add Server</button>
<app-server></app-server>
<app-server></app-server>
```

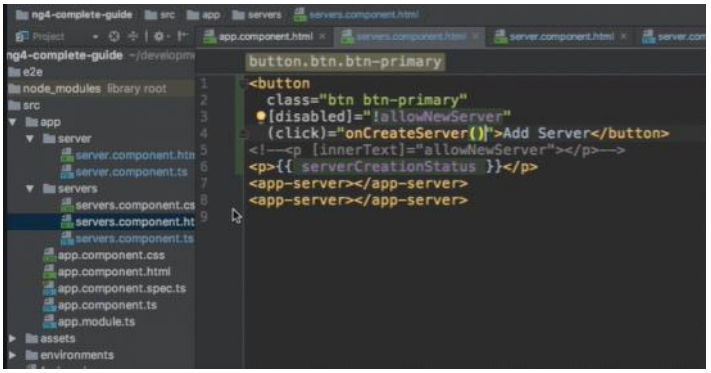
- String Interpolation vs Property Binding
  - When we want to display some text we use string interpolation.
  - When we want to change some property of an html element or for a directive or a component we use Property binding.

- Event Binding
  - Represented by ()
  - 
```
  // selector: '.app-servers',
  selector: 'app-servers',
  // template: `
  //   <app-server></app-server>
  //   <app-server></app-server>`,
  templateUrl: './servers.component.html',
  styleUrls: ['./servers.component.css']
})
export class ServersComponent implements OnInit {
  allowNewServer = false;
  serverCreationStatus = 'No server was created!';

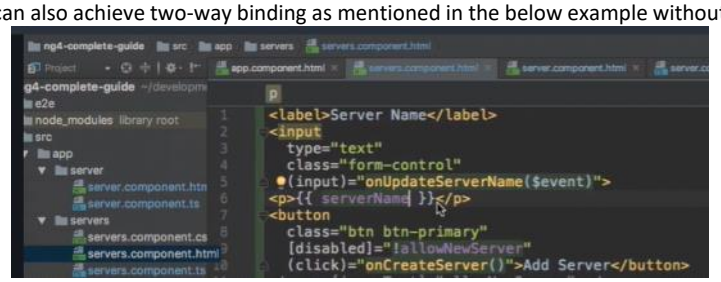
  constructor() {
    setTimeout(() => {
      this.allowNewServer = true;
    }, 2000);
  }

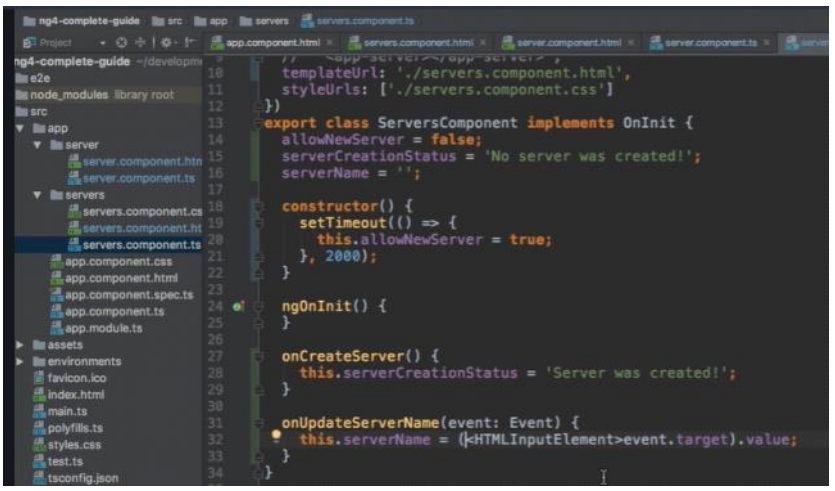
  ngOnInit() {
  }

  onCreateServer() {
    this.serverCreationStatus = 'Server was created!';
  }
```

- 

- Another Ex.
  - ***Js***
    - *onBtnClick($event):void {*
      alert('btn clicked:'+$event);
      }
  - ***Html***
    - *(click) = "onBtnClick($event)"*

- Passing and using data with event binding
  - We can also achieve two-way binding as mentioned in the below example without using ngModel
    - 

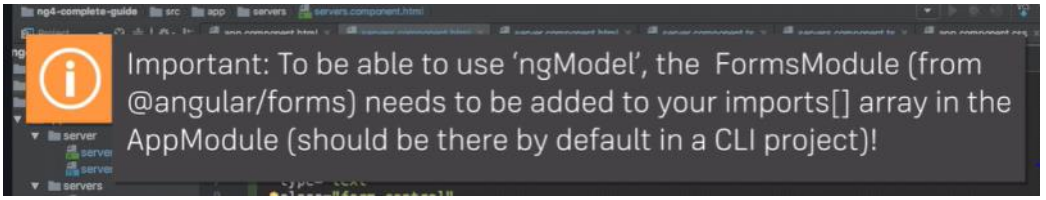
    - 

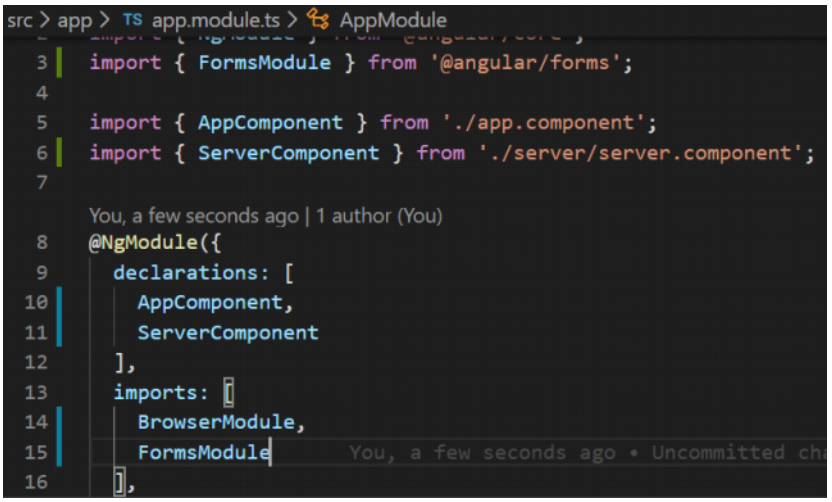
- Two-way binding

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:

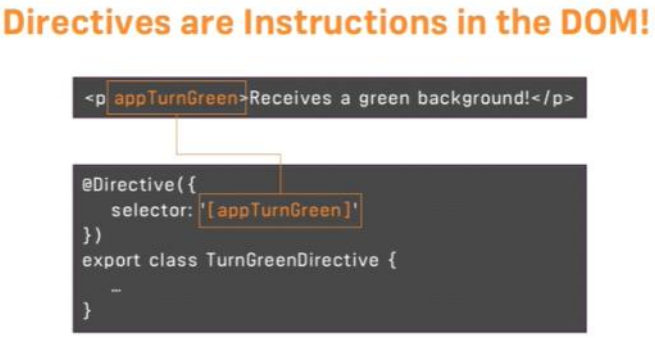
```
import { FormsModule } from '@angular/forms';
```

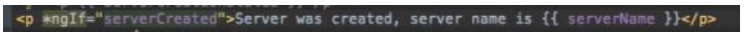
- 

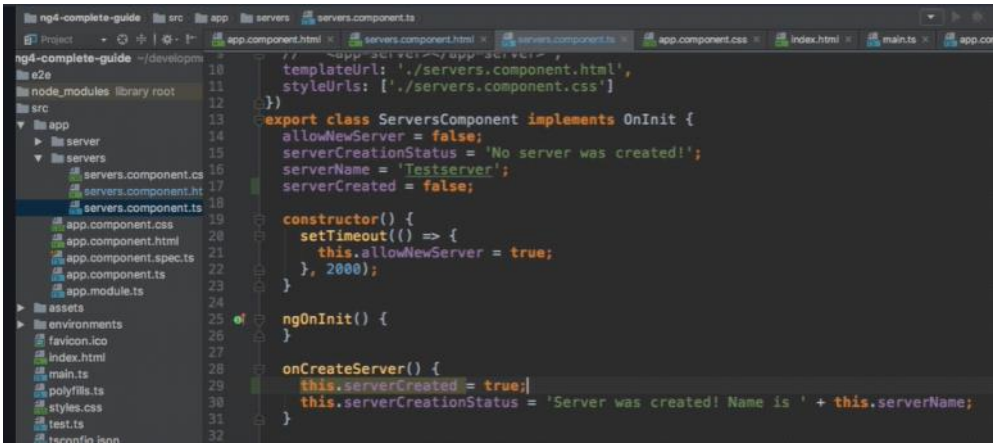
- 

-

- Directives
  - These are instructions in the DOM



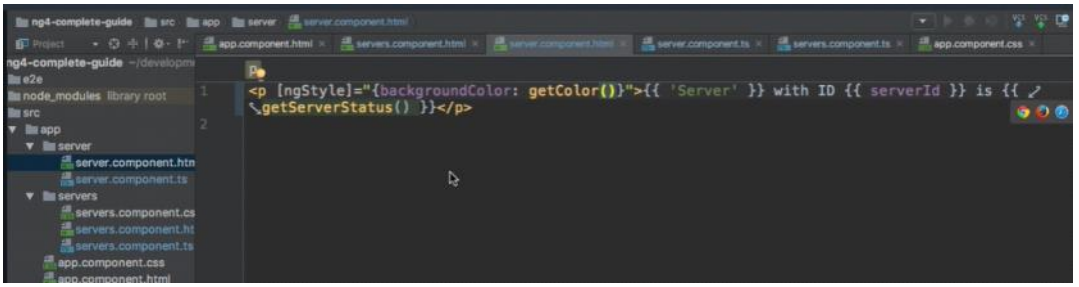
  - Two types of built-in directives are available in Angular
    - Attribute Directives (1-way)
      - *[ngClass]="{'color-text: true', 'font-size': true}"* // Pass as an object to pass multiple classes
    - Structural Directives (2-way) - Changes the DOM
      - *ngFor, *ngIf, [ngSwitch] .. * ngSwitchcase
      - Use trackBy in *ngFor for increasing the performance so that the entire DOM of the component does not get re-rendered
  - ngIf directive (Structural directive)
    - 

- 

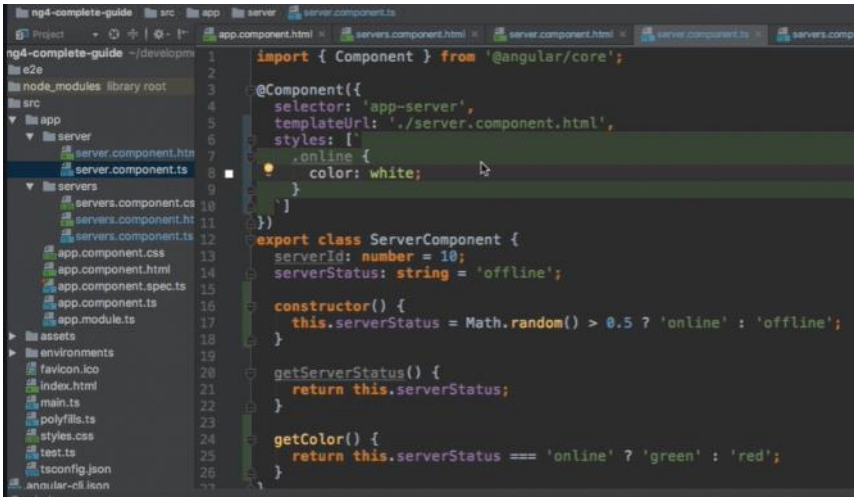
-

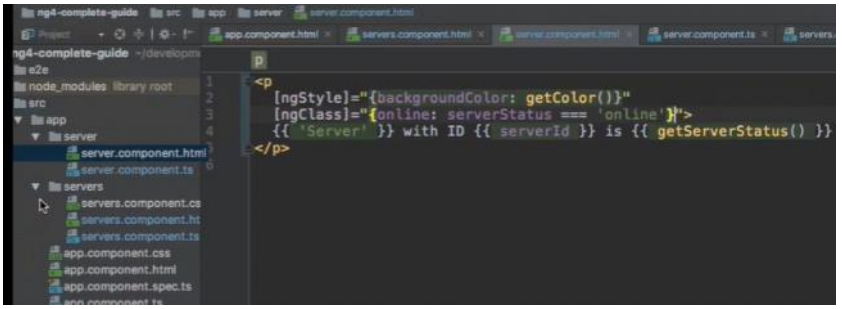
- ngIf directive with else condition (using template/reference variable)
  -

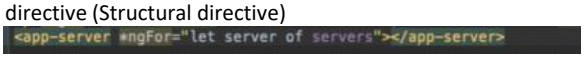
- ngStyle directive (Attribute directive)
  - [] indicate that we need to bind this to the property of the html element
    -

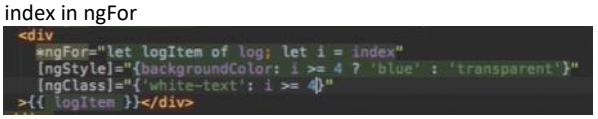
    - 

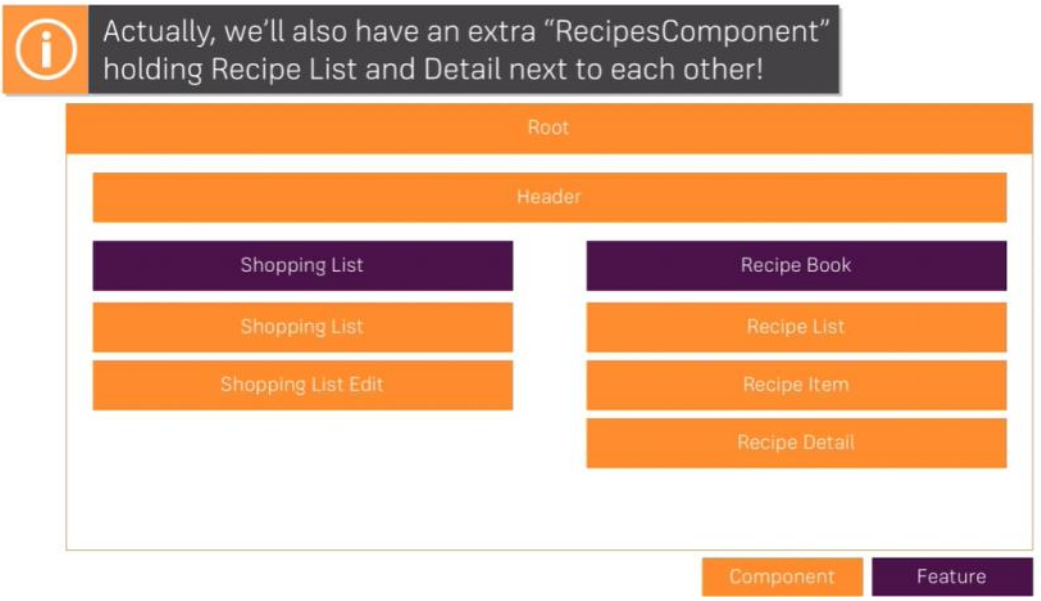
- ngClass directive (Attribute directive)

- 

- 

  - ngFor directive (Structural directive)
    - 

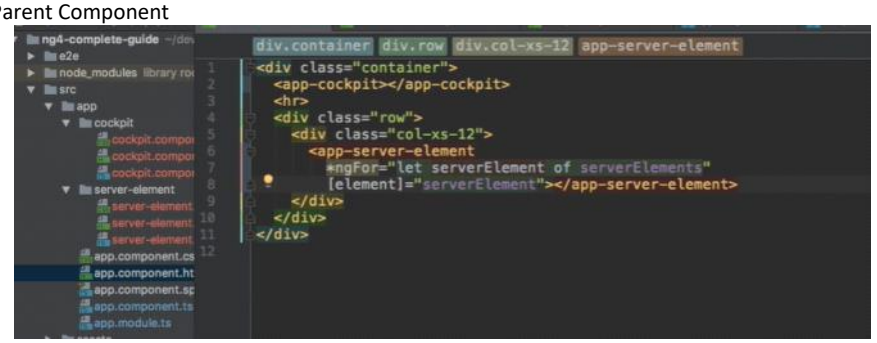
    - Using index in ngFor
      - 

- Sample App Structure
  - 

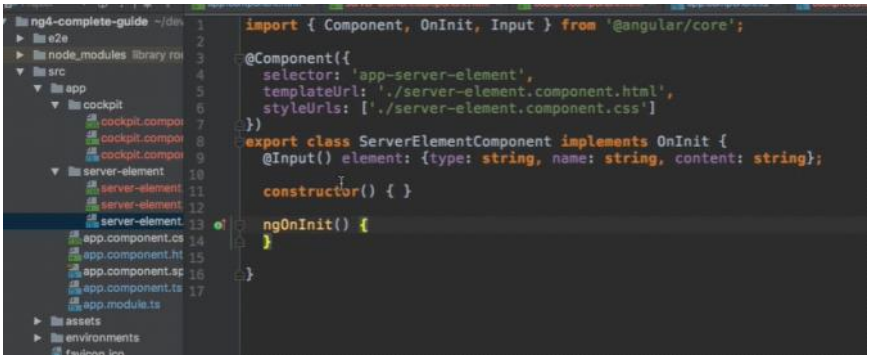
-

- **Components and Data binding**
  - Passing data between components using Property and Event binding

    - All component properties can be accessed only from inside the component and not from any other component.

  - Custom Property Binding (Parent -> Child)
    - Accessing data passed as an attribute from parent can be accessed in the child component using @Input decorator
    - Ex. Parent Component
      - 

    - Ex. Child Component
      - We need to import Input from '@angular/core' library

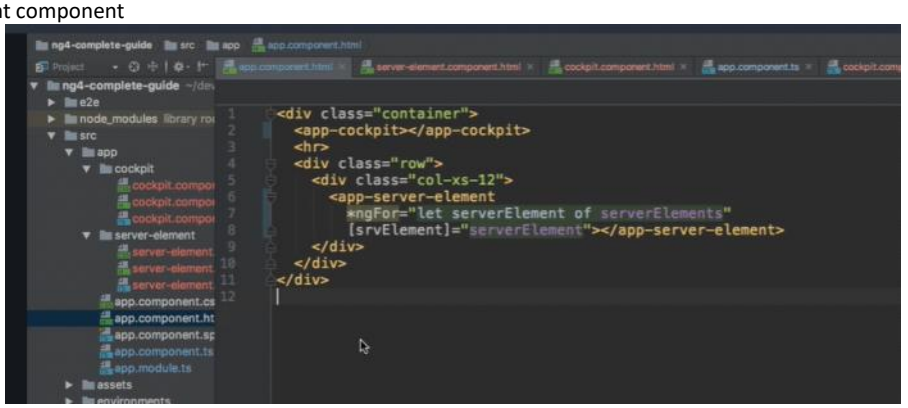
- 
```
import { Component, OnInit, Input } from '@angular/core';

@Component({
  selector: 'app-server-element',
  templateUrl: './server-element.component.html',
  styleUrls: ['./server-element.component.css']
})
export class ServerElementComponent implements OnInit {
  @Input() element: {type: string, name: string, content: string};

  constructor() { }

  ngOnInit() {
  }

}
```

- We can also assign an alias to the Input property as shown below
  - Parent component
    - 
```
<div class="container">
  <app-cockpit></app-cockpit>
  <hr>
  <div class="row">
    <div class="col-xs-12">
      <app-server-element
        *ngFor="let serverElement of serverElements"
        [srvElement]="serverElement"></app-server-element>
    </div>
  </div>
</div>
```
  - Child Component
    -
```
import { Component, OnInit, Input } from '@angular/core';

@Component({
  selector: 'app-server-element',
  templateUrl: './server-element.component.html',
  styleUrls: ['./server-element.component.css']
})
export class ServerElementComponent implements OnInit {
  @Input('srvElement') element: {type: string, name: string, content: string};

  constructor() { }

  ngOnInit() {
  }

}
```

- Event Property Binding (Child -> Parent)
  - Parent Component
    - Add the event name that was emitted from child which calls the function handler on receiving the event.
    -
```
div.container app-cockpit
<div class="container">
  <app-cockpit
    (serverCreated)="onServerAdded($event)"
    (blueprintCreated)="onBlueprintAdded($event)"
  ></app-cockpit>
  <hr>
  <div class="row">
    <div class="col-xs-12">
      <app-server-element
        *ngFor="let serverElement of serverElements"
        [srvElement]="serverElement"></app-server-element>
    </div>
  </div>
</div>
```

- ng4-complete-guide ~/dev
  - e2e
  - node_modules library root
  - src
    - app
      - cockpit
        - cockpit.component.ts
        - cockpit.component.html
        - cockpit.component.spec.ts
      - server-element
        - server-element.component.ts
        - server-element.component.html
        - server-element.component.spec.ts
      - app.component.ts
      - app.component.html
      - app.component.spec.ts
      - app.component.ts
      - app.module.ts
    - assets
    - environments
    - favicon.ico
    - index.html
    - main.ts
    - polyfills.ts
    - styles.css

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  serverElements = [{type: 'server', name: 'Testserver', content: 'Just a test!'}];

  onServerAdded(serverData: {serverName: string, serverContent: string}) {
    this.serverElements.push({
      type: 'server',
      name: serverData.serverName,
      content: serverData.serverContent
    });
  }

  onBlueprintAdded(blueprintData: {serverName: string, serverContent: string}) {
    this.serverElements.push({
      type: 'blueprint',
      name: blueprintData.serverName,
      content: blueprintData.serverContent
    });
  }
```

- Child Component
  - We need to create the output event emitters to emit the events

```
import { Component, OnInit, EventEmitter, Output } from '@angular/core';

@Component({
  selector: 'app-cockpit',
  templateUrl: './cockpit.component.html',
  styleUrls: ['./cockpit.component.css']
})
export class CockpitComponent implements OnInit {
  @Output() serverCreated = new EventEmitter<{serverName: string, serverContent: string}>();
  @Output() blueprintCreated = new EventEmitter<{serverName: string, serverContent: string}>();
  newServerName = '';
  newServerContent = '';

  constructor() { }

  ngOnInit() {
  }

  onAddServer() {
    this.serverCreated.emit({
      serverName: this.newServerName,
      serverContent: this.newServerContent
    });
  }
```

  - Adding alias to the output events

```
import { Component, OnInit, EventEmitter, Output } from '@angular/core';

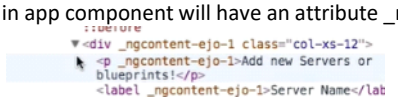
@Component({
  selector: 'app-cockpit',
  templateUrl: './cockpit.component.html',
  styleUrls: ['./cockpit.component.css']
})
export class CockpitComponent implements OnInit {
  @Output() serverCreated = new EventEmitter<{serverName: string, serverContent: string}>();
  @Output('bpCreated') blueprintCreated = new EventEmitter<{serverName: string, serverContent:
string}>();
  newServerName = '';
  newServerContent = '';

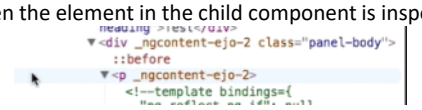
  constructor() { }

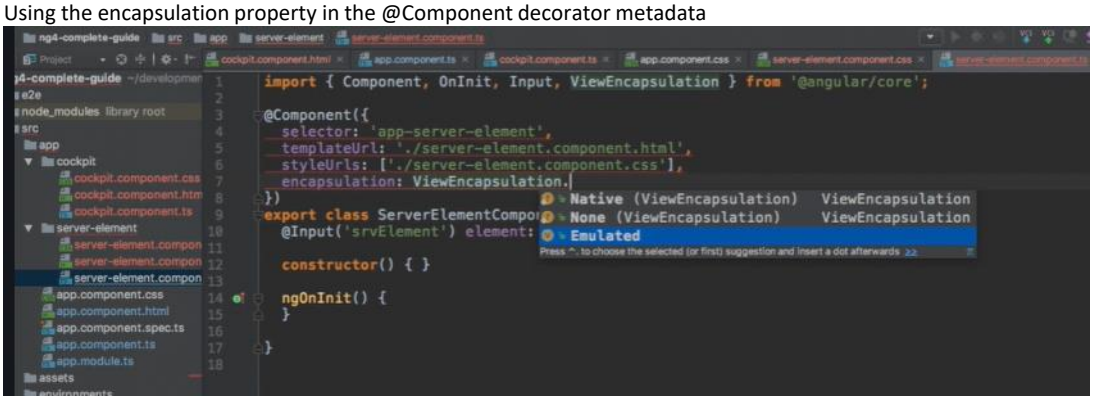
  ngOnInit() {
  }

  onAddServer() {
    this.serverCreated.emit({
```

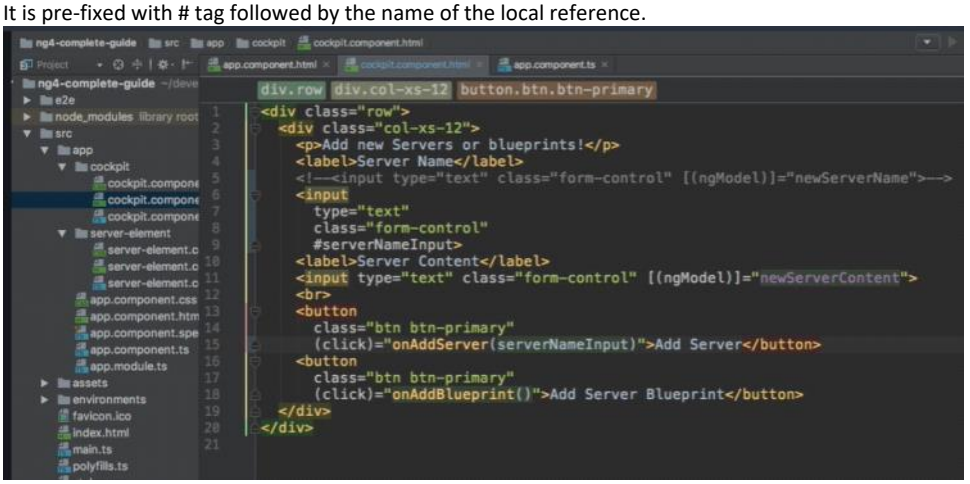
```
<div class="container">
  <app-cockpit
    (serverCreated)="onServerAdded($event)"
    (bpCreated)="onBlueprintAdded($event)"
  ></app-cockpit>
  <hr>
  <div class="row">
    <div class="col-xs-12">
      <app-server-element
        *ngFor="let serverElement of serverElements"
        [srvElement]="serverElement"></app-server-element>
    </div>
  </div>
</div>
```

- Understanding View Encapsulation
  - Styles applied to a component will be applied only to the component but not the child component because of the Encapsulation that Angular provides.
  - Angular enforces this style encapsulation.
  - Angular emulates the Shadow DOM. even though Shadow DOM is a technology which is not supported by all the browsers where each element has its kind of its own Shadow DOM assigned and its styles will be applied.
  - As all browsers do not support this Shadow DOM, as explained below Angular emulates this behavior of View Encapsulation.
  - Ex.
    - If app component has a style that p { color: blue}
    - p tags in app component will inherit this style
    - p tag in app component will have an attribute _ngcontent-ejo-1 added as shown in the below screenshot
      - 

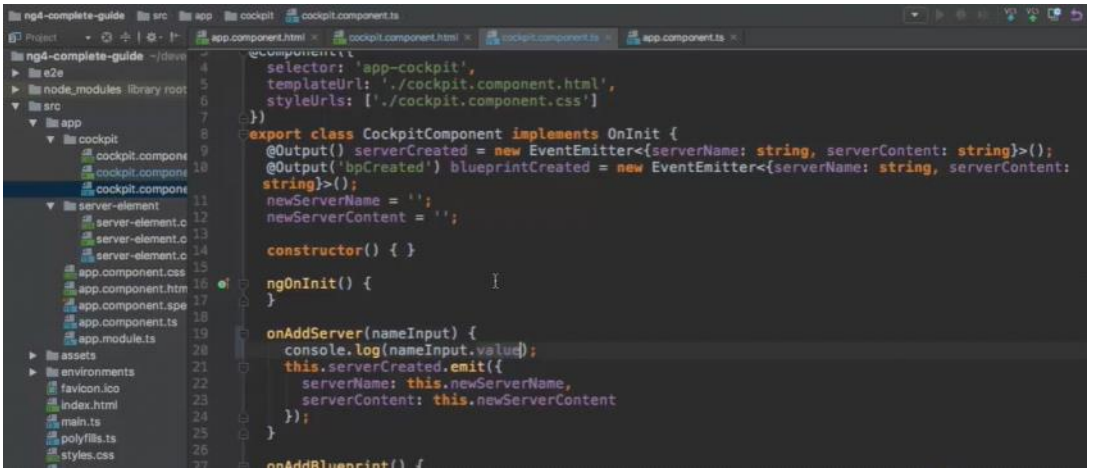
    - The child component of app-component with a p tag will not inherit the parent p tag style because of the below explanation
    - When the element in the child component is inspected, we see that it has an attribute appended to it _ngcontent-ejo-2
      - 

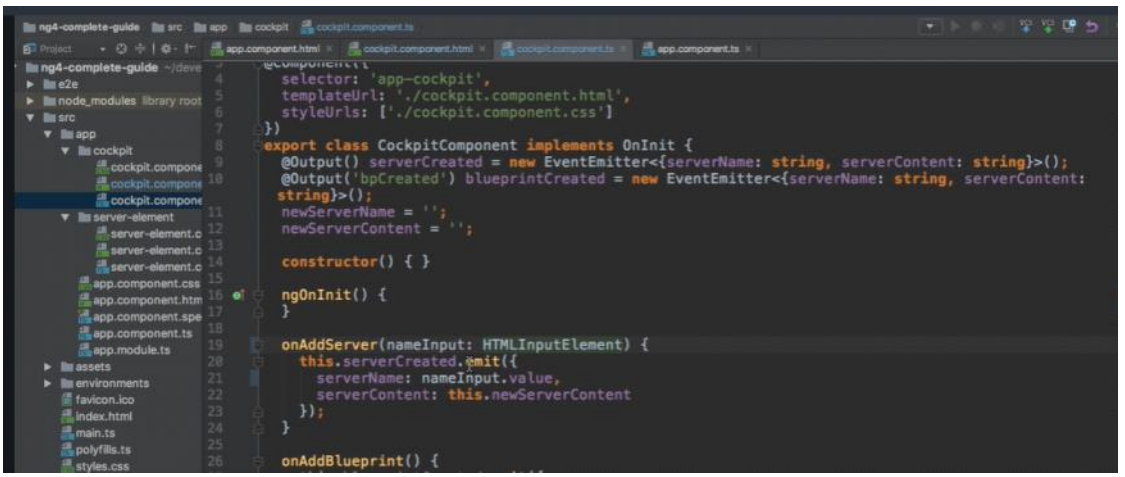
  - We can override this Encapsulation as shown below
    - Using the encapsulation property in the @Component decorator metadata
    - 

    - ViewEncapsulation needs to be imported from '@angular/core'
    - ViewEncapsulation has three modes
      - Emulated (Default) // Applies View Encapsulation
      - None // Removes all the dynamic attributes added to the elements in the component. View Encapsulation will be removed
      - Native // Uses Shadow DOM technology, but gives us the same result as Emulated option but it is not supported by all the browsers

- Using local references in templates
  - Local reference can be placed on any html element.
  - It is pre-fixed with # tag followed by the name of the local reference.
  - 

  - The local reference in the above code #serverNameInput will hold a reference to the element not the element's value
  - The local reference can be used in button click to get the element's value

- 

- 

- Getting access to the template and DOM with @ViewChild
  - @ViewChild needs to be imported from '@angular/core'

In **Angular 8**, the `@ViewChild()` syntax which you'll see in the next lecture needs to be changed slightly:

Instead of:

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

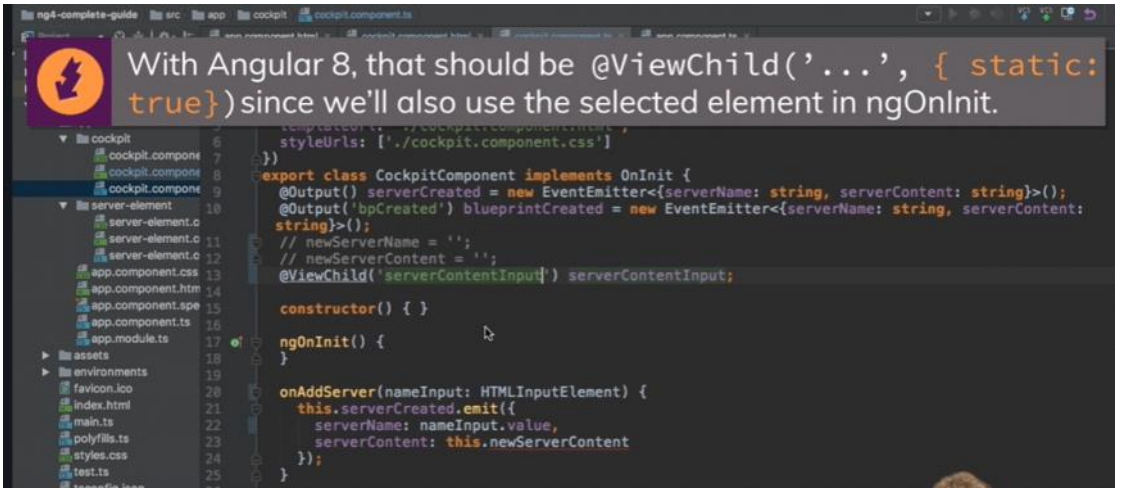
use

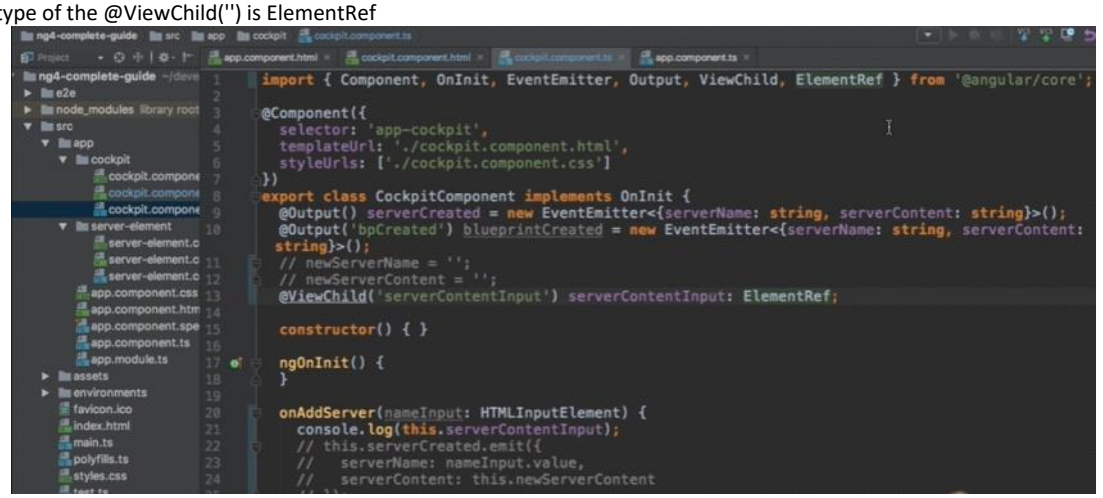
```
@ViewChild('serverContentInput', {static: true}) serverContentInput: Elemen
```

The same change (add `{ static: true }` as a second argument) needs to be applied to ALL usages of `@ViewChild()` (and also `@ContentChild()` which you'll learn about later) IF you plan on accessing the selected element inside of `ngOnInit()`.

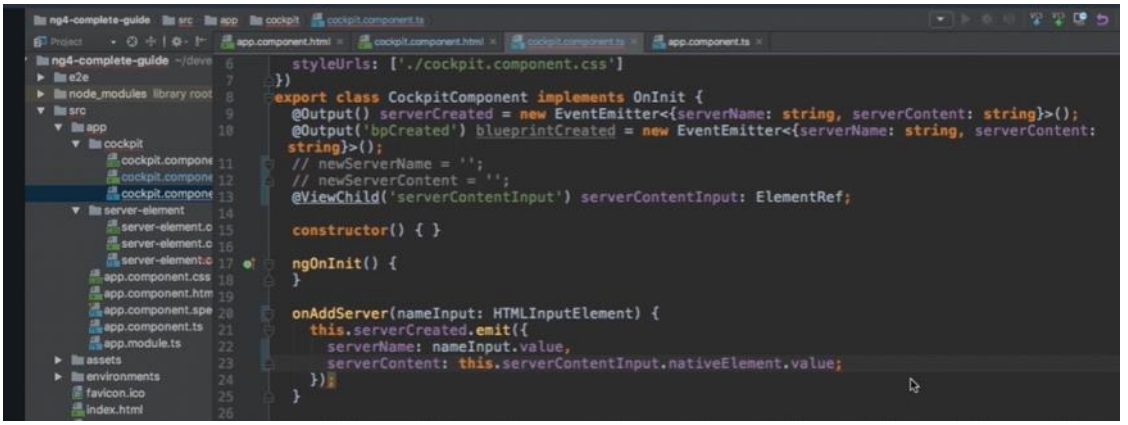
- If you DON'T access the selected element in `ngOnInit` (but anywhere else in your component), set `static: false` instead!

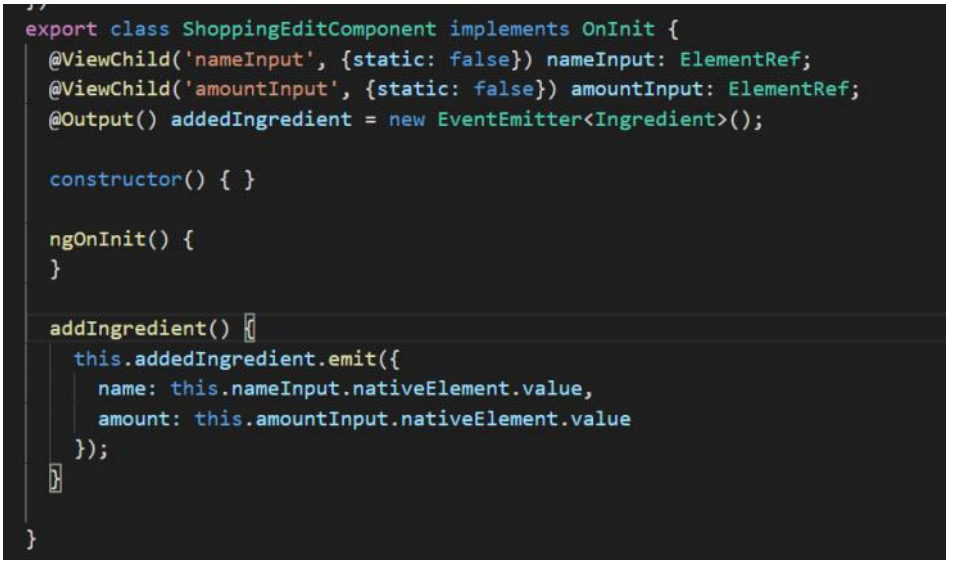
  This is a temporary adjustment which will NOT be required anymore once Angular 9 is released!

- 

- In the above code @ViewChild('serverContentInput'), serverContentInput is the local reference of an element (Ex. #serverContentInput on an input element)
- We can also pass a component in the parenthesis of @ViewChild. Ex. @ViewChild(AppComponent)
- The type of the @ViewChild('') is ElementRef
  - 

    - ElementRef has an nativeElement property to get access to the underlying element

- 

- 

- Projecting content into components using ng-content
  - ng-content is used to dynamically pass an html into an component
  - To dynamically place the html markup in a particular template
  - Ex.
    - Panel component

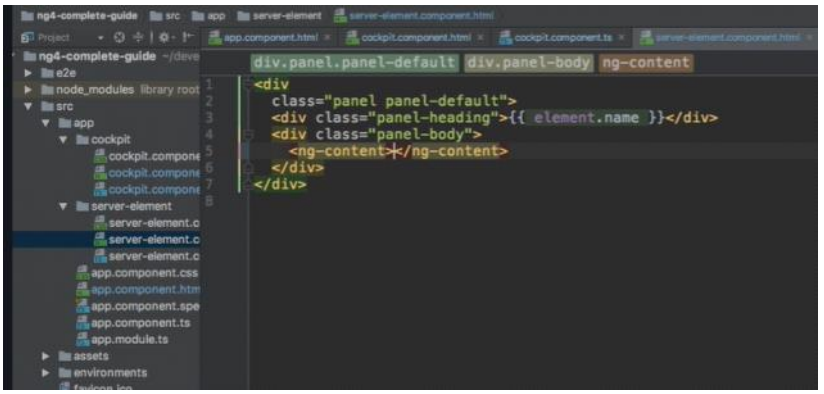
```
<div>
  <div class="card-header">
    <ng-content select=".header"></ng-content>
  </div>
  <div class="card-body">
    <ng-content select=".body"></ng-content>
  </div>
</div>

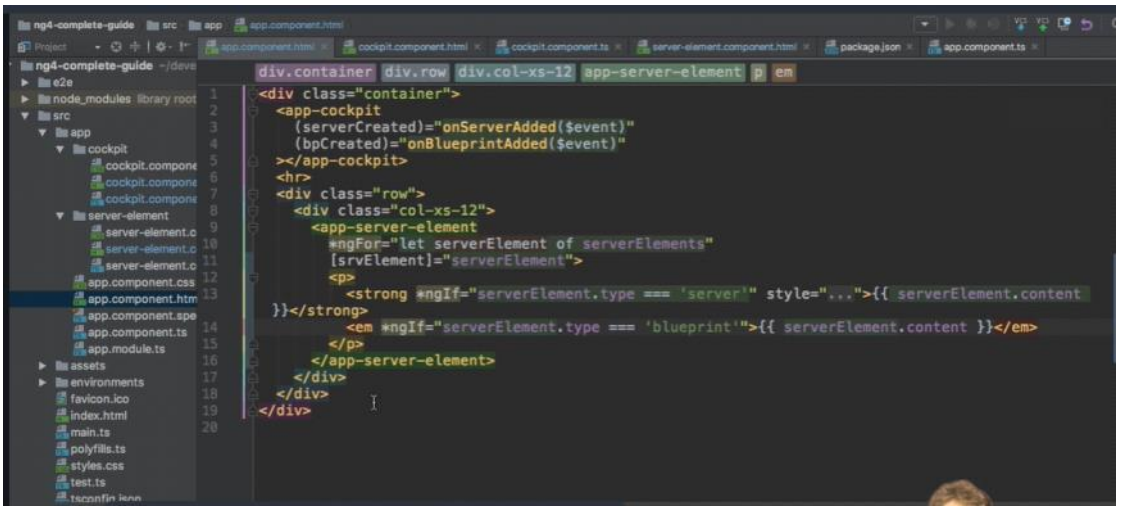
<panel>
 <div class="header">
   <h1> My Passed Header<h1>
 </div>
 <div class="body">
   <p> My Passed Body</p>
 </div>
</panel>
```

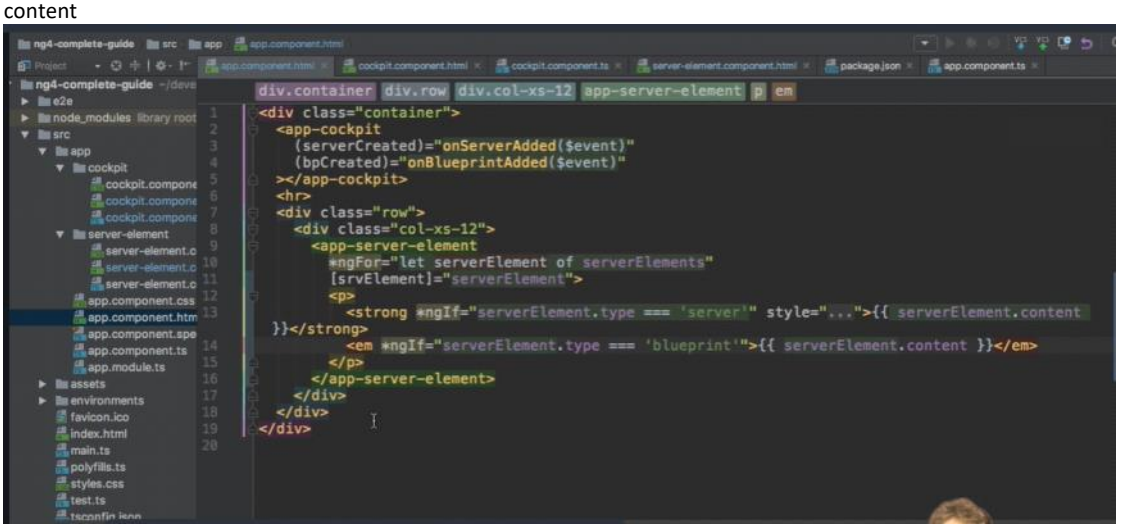
  - Use ng-container if we want to replace the whole markup directly
  - Ex.

```
<panel>
 <ng-container class="header">
   <h1> My Passed Header<h1>
 </ng-container>
 <ng-container class="body">
```

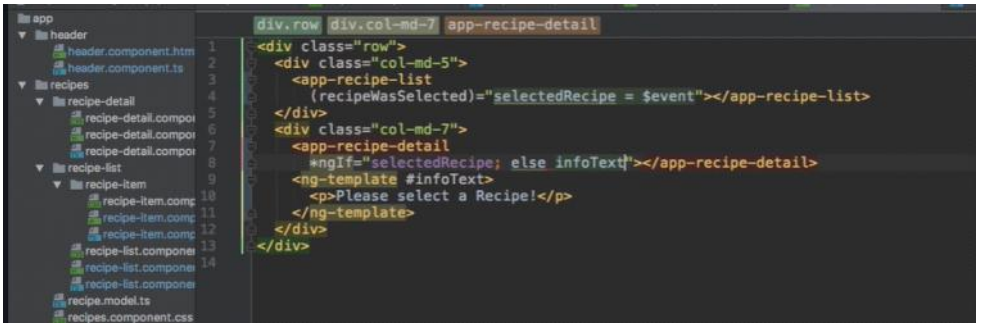
```
  <p> My Passed Body</p>
 </ng-container>
</panel>
```

- Ex
  - 

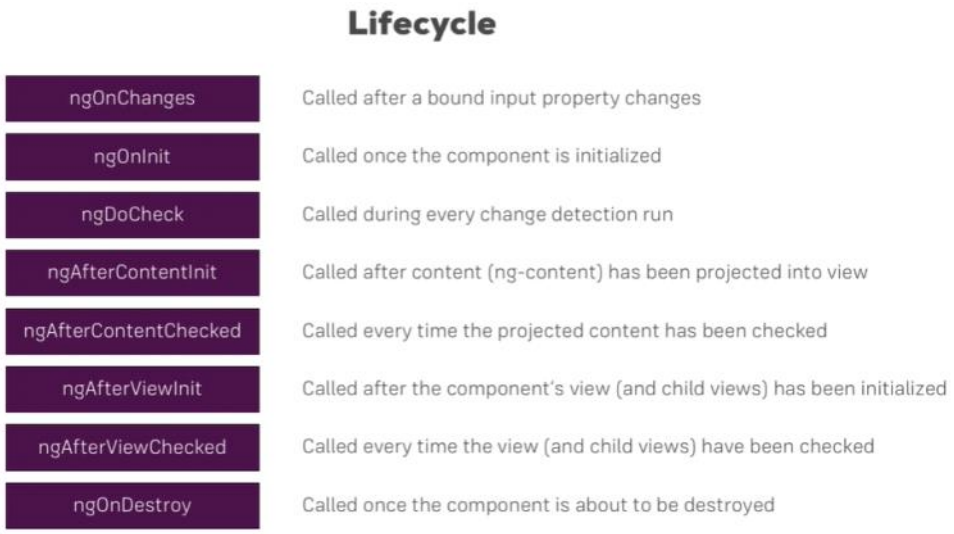
  - 

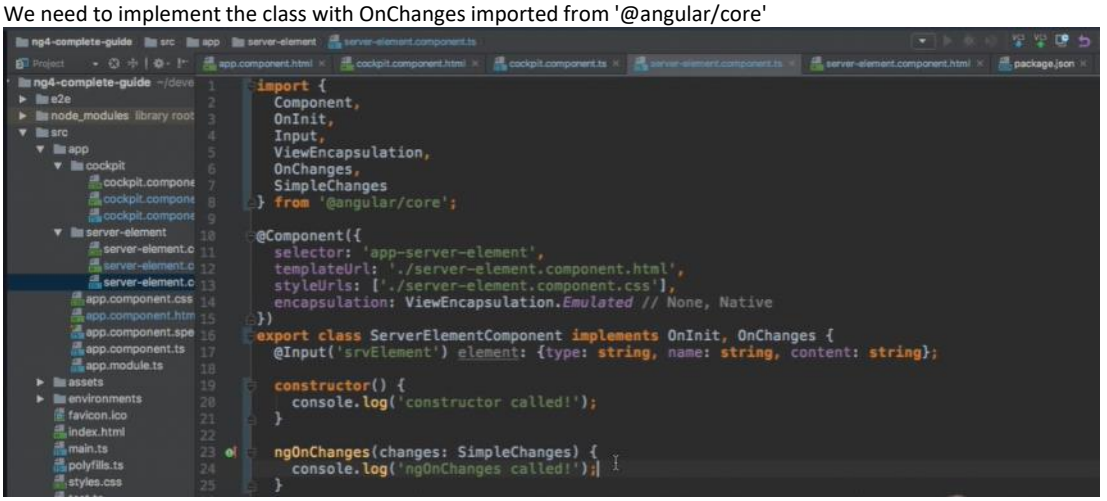
  - In the above code, the code between the opening and closing tags of app-server-element will be dynamically projected into ng-content
  - 

- Using ng-template

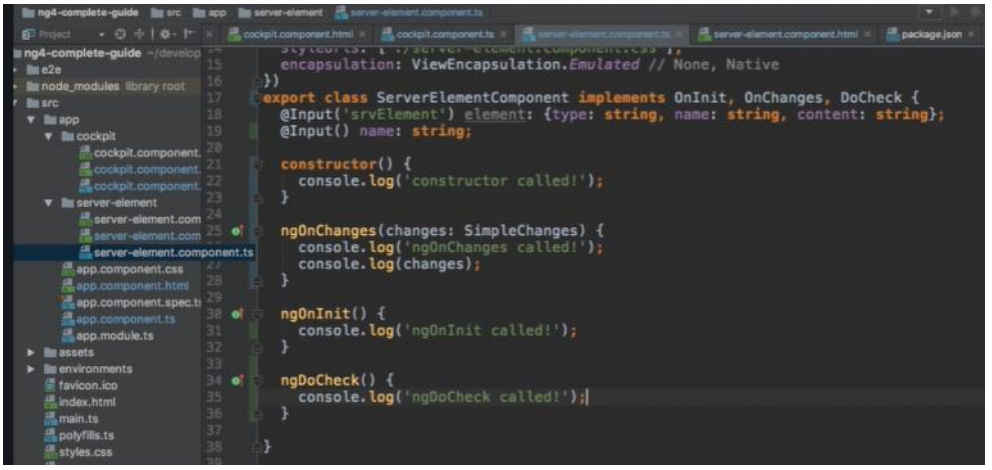
- 

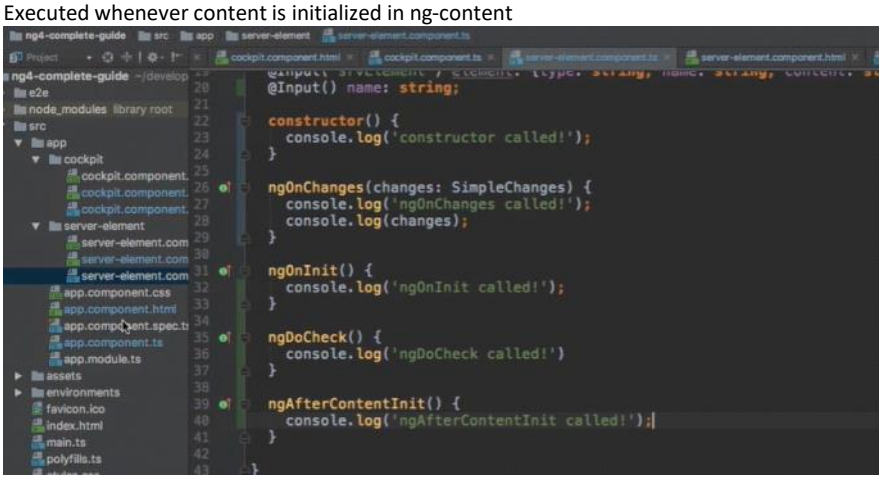
- Component LifeCycle

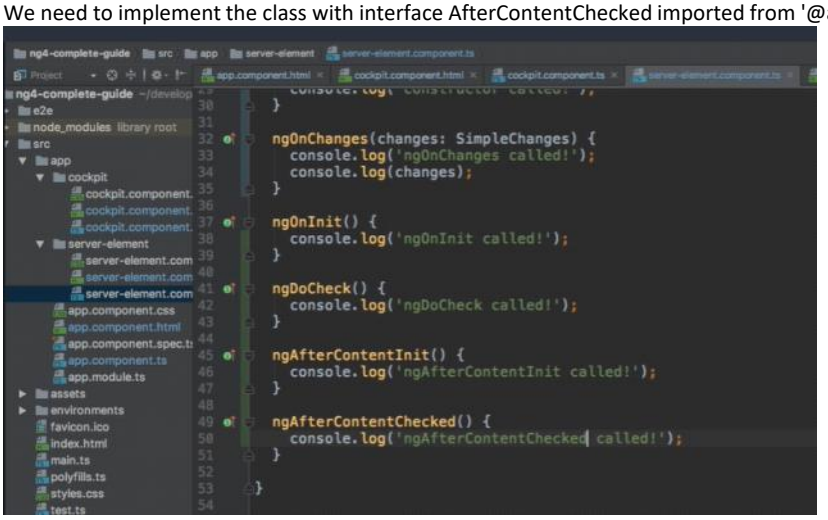


-
- Order of life cycle in which they will be called
  - Constructor
  - ngOnChanges
    - We need to implement the class with OnChanges imported from '@angular/core'
    - 

    - Also we need to pass an argument of type SimpleChanges imported from '@angular/core' to the onChanges method
  - ngOnInit
  - ngDoCheck
    - We need to implement the class with DoCheck imported from '@angular/core'
    - Executed on every change detection run i.e whenever angular checks for any changes.
    - Changes can be triggered by a click or a promise or any other event
    - Costs a lot of performance
    - Can run this if angular did not change something

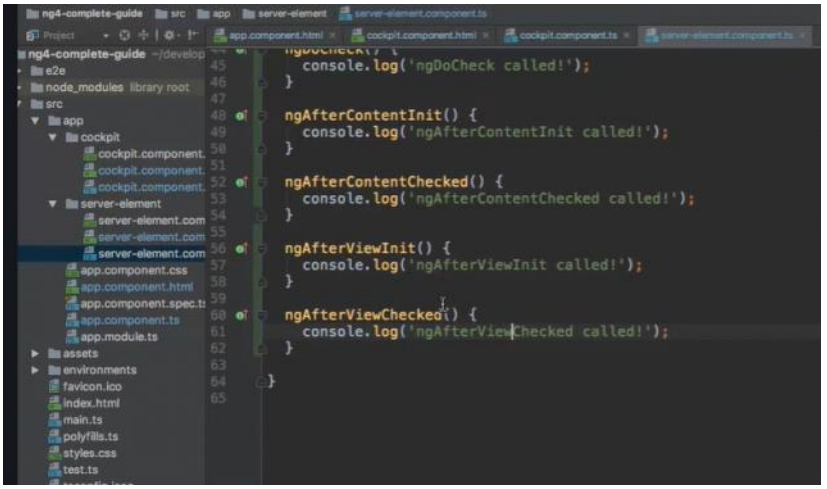
- 

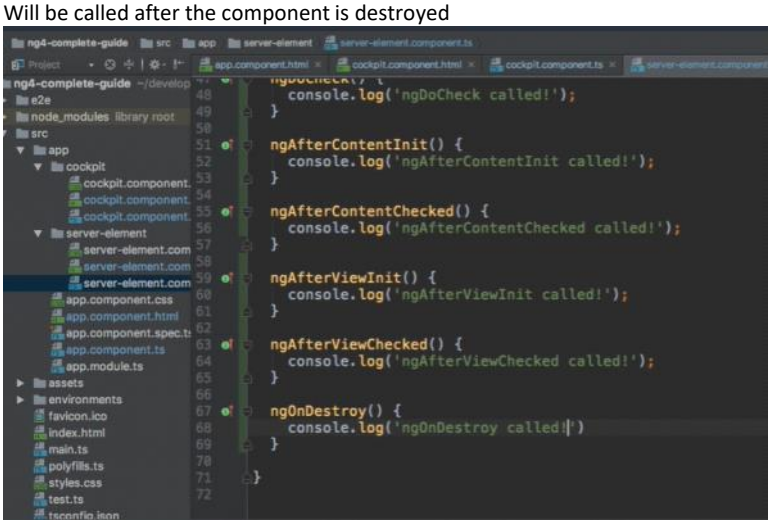
- ngAfterContentInit
  - We need to implement the class with AfterContentInit imported from '@angular/core'
  - Executed whenever content is initialized in ng-content
  - 

- ngAfterContentChecked
  - We need to implement the class with interface AfterContentChecked imported from '@angular/core'
  - 

  - This is also called after every ngDoCheck

- ngAfterViewInit, ngAfterViewChecked
  - We need to implement the class with interfaces AfterViewInit, AfterViewChecked imported from '@angular/core'
  - AfterViewInit gives access to the elements references as the view will be rendered

- 

  - ngOnDestroy
    - We need to implement the class with interface OnDestroy imported from '@angular/core'
    - Will be called after the component is destroyed
    - 

- @ContentChild()

  - In **Angular 8**, the `@ContentChild()` syntax which you'll see in the next lecture needs to be changed slightly:

    Instead of:

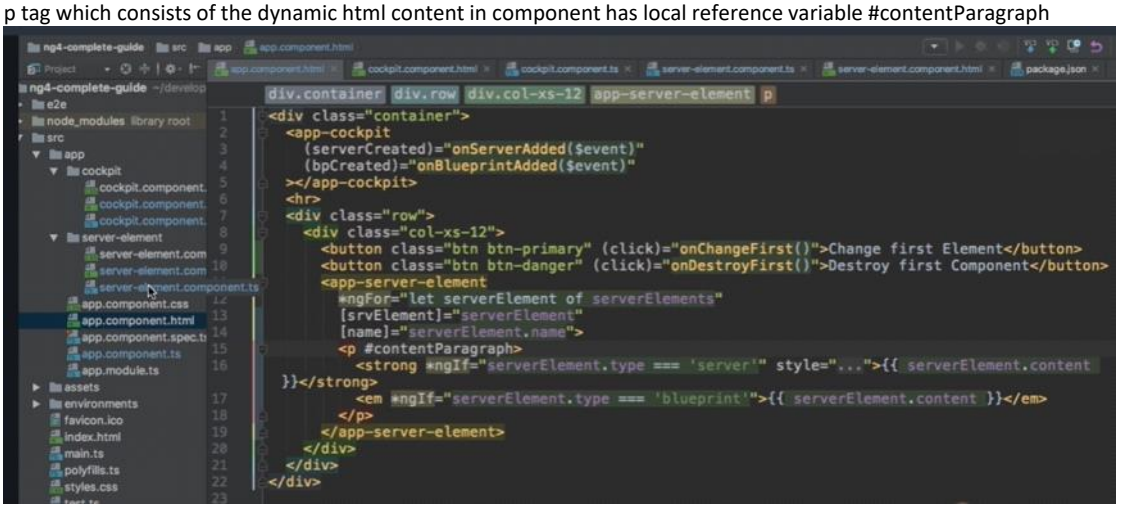
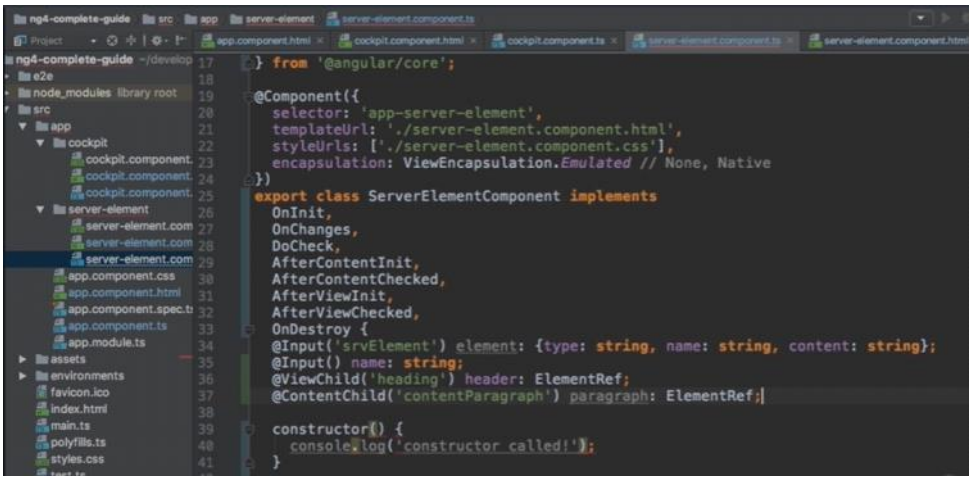
    ```
    @ContentChild('contentParagraph') paragraph: ElementRef;
    ```

    use

    ```
    @ContentChild('contentParagraph', {static: true}) paragraph: ElementRef;
    ```

- The same change (add `{ static: true }` as a second argument) needs to be applied to ALL usages of `@ContentChild()` if you use the selected element inside of `ngOnInit` (as we do it in the lectures).

  If you DON'T use the selected element in `ngOnInit`, set `static: false` instead.

- The dynamically added html using ng-content can be referenced through a local reference variable using @ContentChild in the component where <ng-content> tag has been added
- Ex. Dynamic html through local reference variable in app-component can be referenced in app-server-element where ng-content was added using @ContentChild
  - p tag which consists of the dynamic html content in component has local reference variable #contentParagraph
  - 
```
<div class="container">
  <app-cockpit
    (serverCreated)="onServerAdded($event)"
    (bpCreated)="onBlueprintAdded($event)"
  ></app-cockpit>
  <hr>
  <div class="row">
    <div class="col-xs-12">
      <button class="btn btn-primary" (click)="onChangeFirst()">Change first Element</button>
      <button class="btn btn-danger" (click)="onDestroyFirst()">Destroy first Component</button>
      <app-server-element
        *ngFor="let serverElement of serverElements"
        [srvElement]="serverElement"
        [name]="serverElement.name">
        <p #contentParagraph>
          <strong *ngIf="serverElement.type === 'server'" style="...">{{ serverElement.content }}</strong>
          <em *ngIf="serverElement.type === 'blueprint'">{{ serverElement.content }}</em>
        </p>
      </app-server-element>
    </div>
  </div>
</div>
```
  - 
```
} from '@angular/core';

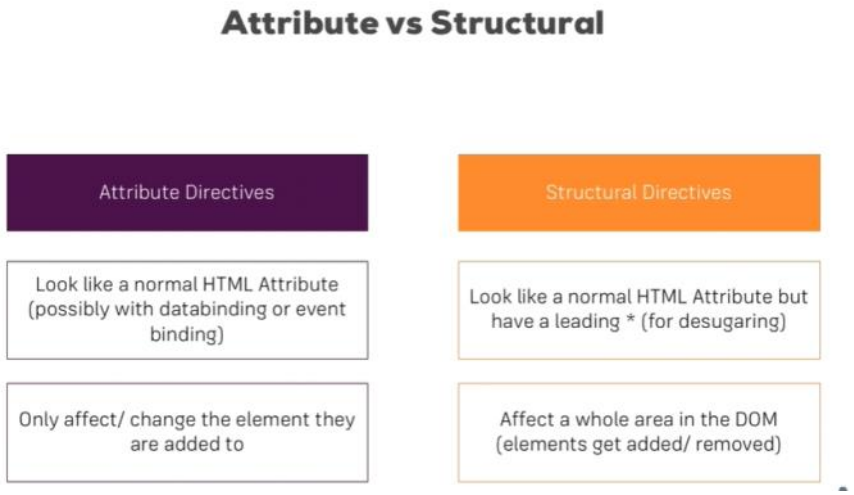
@Component({
  selector: 'app-server-element',
  templateUrl: './server-element.component.html',
  styleUrls: ['./server-element.component.css'],
  encapsulation: ViewEncapsulation.Emulated // None, Native
})
export class ServerElementComponent implements
  OnInit,
  OnChanges,
  DoCheck,
  AfterContentInit,
  AfterContentChecked,
  AfterViewInit,
  AfterViewChecked,
  OnDestroy {
  @Input('srvElement') element: {type: string, name: string, content: string};
  @Input() name: string;
  @ViewChild('heading') header: ElementRef;
  @ContentChild('contentParagraph') paragraph: ElementRef;

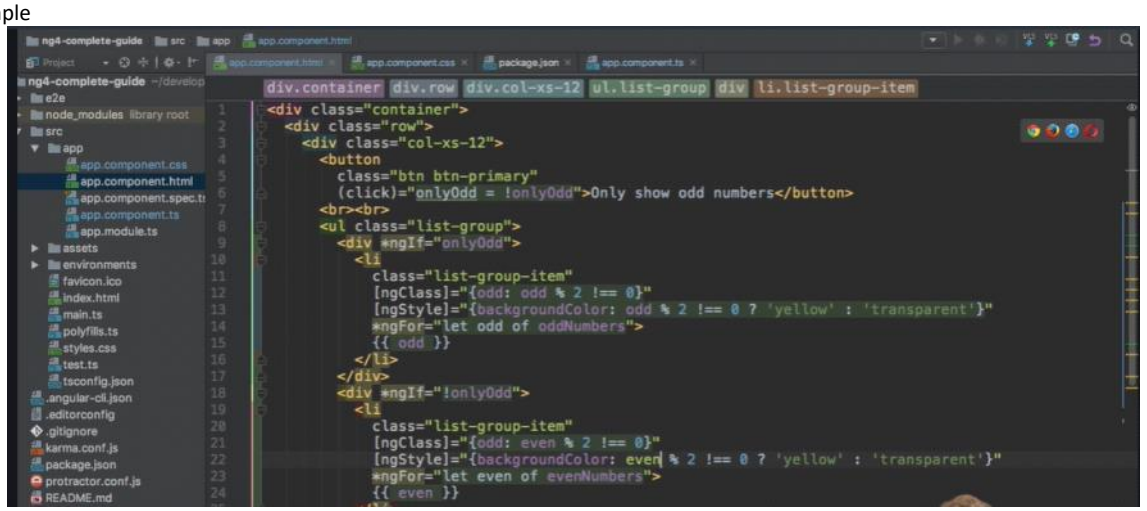
  constructor() {
    console.log('constructor called!');
  }
```
  - The element referenced can be accessed in ngAfterContentInit as the view will be rendererd by the time this hook runs.
  - In ngOnInit, as the view will not be rendered by the time this hook runs it returns empty when we reference the element

- Exercise
  - **Bind it!**

    1. Create three new components: GameControl, Odd and Even
    2. The GameControl Component should have buttons to start and stop the game
    3. When starting the game, an event (holding a incrementing number) should get emitted each second (ref = setInterval())
    4. The event should be listenable from outside the component
    5. When stopping the game, no more events should get emitted (clearInterval(ref))
    6. A new Odd component should get created for every odd number emitted, the same should happen for the Even Component (on even numbers)
    7. Simply output Odd - NUMBER or Even - NUMBER in the two components
    8. Style the element (e.g. paragraph) holding your output text differently in both components

- Directives



- Example
  - 
```
<div class="container">
  <div class="row">
    <div class="col-xs-12">
      <button
        class="btn btn-primary"
        (click)="onlyOdd = !onlyOdd">Only show odd numbers</button>
      <br><br>
      <ul class="list-group">
        <div *ngIf="onlyOdd">
          <li
            class="list-group-item"
            [ngClass]="{odd: odd % 2 !== 0}"
            [ngStyle]="{backgroundColor: odd % 2 !== 0 ? 'yellow' : 'transparent'}"
            *ngFor="let odd of oddNumbers">
            {{ odd }}
          </li>
        </div>
        <div *ngIf="!onlyOdd">
          <li
            class="list-group-item"
            [ngClass]="{odd: even % 2 !== 0}"
            [ngStyle]="{backgroundColor: even % 2 !== 0 ? 'yellow' : 'transparent'}"
            *ngFor="let even of evenNumbers">
            {{ even }}
```

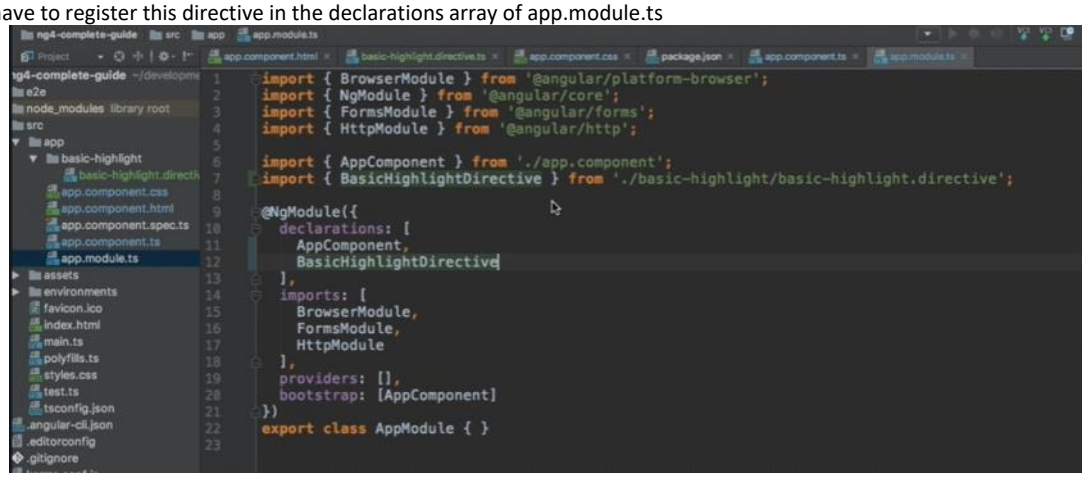
- Structural Directives
  - Examples: *nglf, *ngFor etc..
  - We can't have more than 1 structural directive on the same element.
- Attribute Directives
  - Examples: [ngClass], [ngStyle] etc..

- Creating a basic directive
  - We use Directive decorator to create a directive, which is imported from '@angular/core'
  - It needs the below metadata
    - selector
      - Ex. selector: appBasicHighLight // selects by element
      - Ex. selector: [appBasicHighlight] // selects by attribute

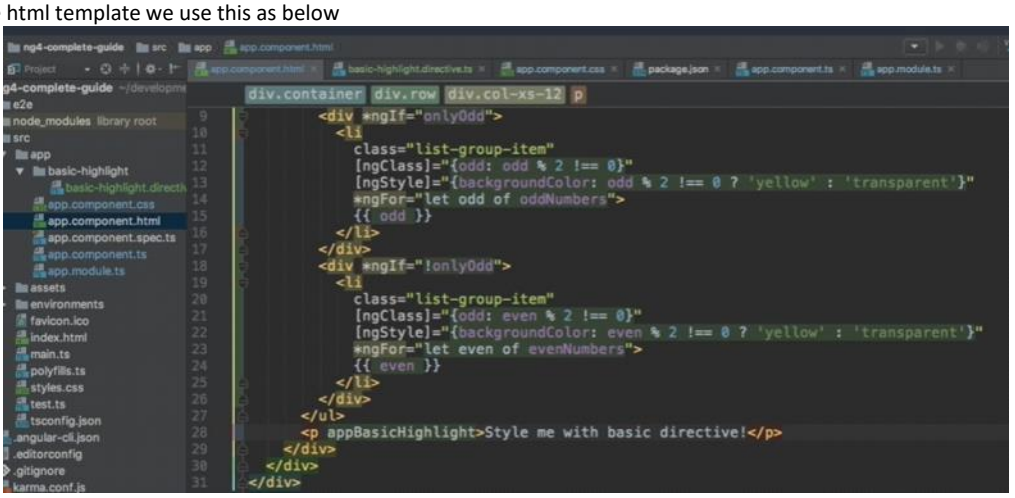
  - In order to get access to the element, we need to import ElementRef from '@angular/core'
  -
```
import { Directive, ElementRef, OnInit } from '@angular/core';

@Directive({
  selector: '[appBasicHighlight]'
})
export class BasicHighlightDirective implements OnInit {
  constructor(private elementRef: ElementRef) {
  }

  ngOnInit() {
    this.elementRef.nativeElement.style.backgroundColor = 'green';
  }
}
```

- We have to register this directive in the declarations array of app.module.ts
  - 
```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';

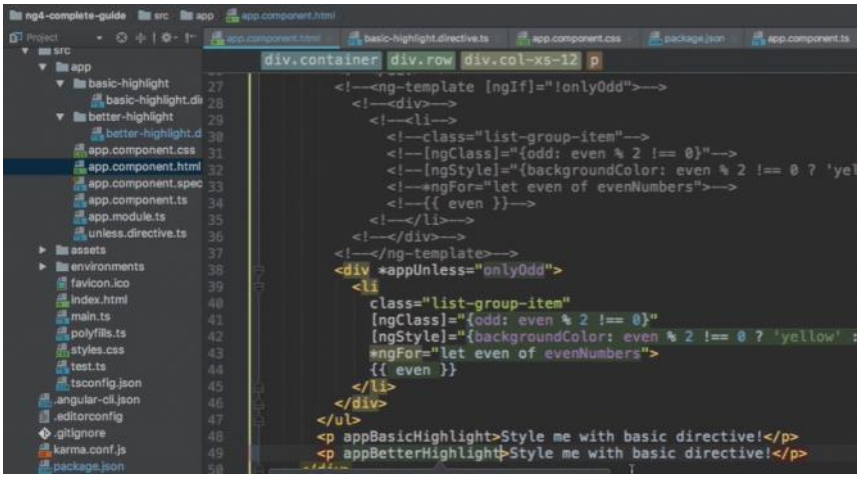
import { AppComponent } from './app.component';
import { BasicHighlightDirective } from './basic-highlight/basic-highlight.directive';

@NgModule({
  declarations: [
    AppComponent,
    BasicHighlightDirective
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```
- In the html template we use this as below
  - 
```
div.container div.row div.col-xs-12 p
        <div *ngIf="onlyOdd">
          <li
            class="list-group-item"
            [ngClass]="{odd: odd % 2 !== 0}"
            [ngStyle]="{backgroundColor: odd % 2 !== 0 ? 'yellow' : 'transparent'}"
            *ngFor="let odd of oddNumbers">
            {{ odd }}
          </li>
        </div>
        <div *ngIf="!onlyOdd">
          <li
            class="list-group-item"
            [ngClass]="{odd: even % 2 !== 0}"
            [ngStyle]="{backgroundColor: even % 2 !== 0 ? 'yellow' : 'transparent'}"
            *ngFor="let even of evenNumbers">
            {{ even }}
          </li>
        </div>
      </ul>
      <p appBasicHighlight>Style me with basic directive!</p>
    </div>
  </div>
</div>
```
- But as used in the above example it is not recommended to directly access the element.
- Creating directive via cli
  - ng g directive <directive-name>
- Using Renderer to build a directive
  - We need to import Renderer2 from '@angular/core'
  - Rendererv2 is changes to Renderer2
  - This is a better way to access the DOM elements and set style via renderer methods
  - Renderer has an in-built setStyle method to set the style we want to the element by accessing the element via ElementRef
  -
```
import { Directive, OnInit, ElementRef, Renderer2 } from '@angular/core';

@Directive({
  selector: '[appBetterHighlight]'
})
export class BetterHighlightDirective implements OnInit {

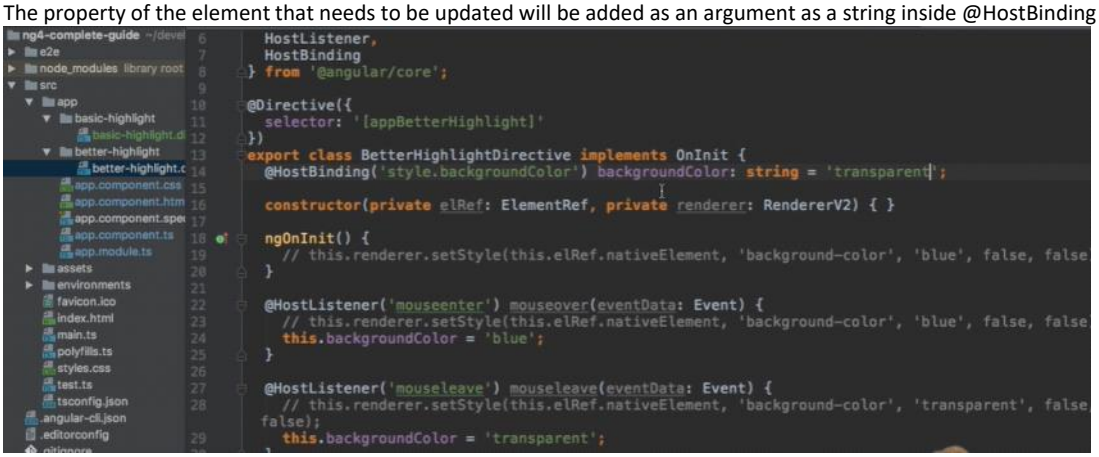
  constructor(private elRef: ElementRef, private renderer: Renderer2) { }

  ngOnInit() {
    this.renderer.setStyle(this.elRef.nativeElement, 'background-color', 'blue')
  }
}
```

- 

  - Listed below are the methods of Renderer
  -
```
abstract class Renderer2 {
  abstract data: {...}
  destroyNode: ((node: any) => void) | null
  abstract destroy(): void
  abstract createElement(name: string, namespace?: string): any
  abstract createComment(value: string): any
  abstract createText(value: string): any
  abstract appendChild(parent: any, newChild: any): void
  abstract insertBefore(parent: any, newChild: any, refChild: any): void
  abstract removeChild(parent: any, oldChild: any, isHostElement?: boolean): void
  abstract selectRootElement(selectorOrNode: any, preserveContent?: boolean): any
  abstract parentNode(node: any): any
  abstract nextSibling(node: any): any
  abstract setAttribute(el: any, name: string, value: string, namespace?: string):
void
  abstract removeAttribute(el: any, name: string, namespace?: string): void
  abstract addClass(el: any, name: string): void
  abstract removeClass(el: any, name: string): void
  abstract setStyle(el: any, style: string, value: any, flags?:
RendererStyleFlags2): void
  abstract removeStyle(el: any, style: string, flags?: RendererStyleFlags2): void
```

- But the above approach is not that interactive, so to listen to all the events to the element to which the directive is attached to, we need to use @HostListener
  - @HostListener decorator is imported from '@angular/core'
  - The event that needs to be triggered is added as an argument as a string inside @HostListener
  -

- In order to replace the renderer, we can use @HostBinding
  - @HostBinding decorator need to be imported from '@angular/core'
  - The property of the element that needs to be updated will be added as an argument as a string inside @HostBinding
  - 
```
  HostListener,
  HostBinding
} from '@angular/core';

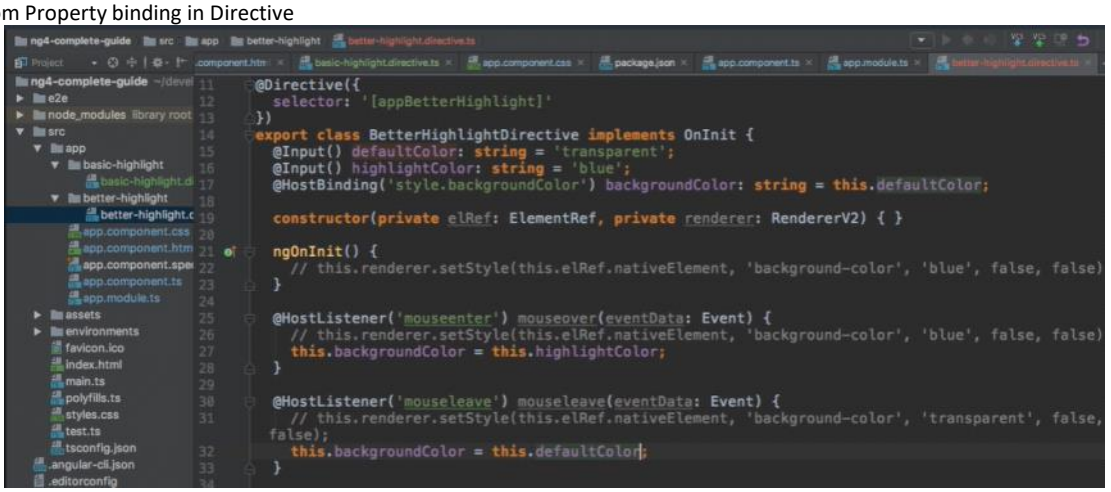
@Directive({
  selector: '[appBetterHighlight]'
})
export class BetterHighlightDirective implements OnInit {
  @HostBinding('style.backgroundColor') backgroundColor: string = 'transparent';

  constructor(private elRef: ElementRef, private renderer: RendererV2) { }

  ngOnInit() {
    // this.renderer.setStyle(this.elRef.nativeElement, 'background-color', 'blue', false, false
  }

  @HostListener('mouseenter') mouseover(eventData: Event) {
    // this.renderer.setStyle(this.elRef.nativeElement, 'background-color', 'blue', false, false
    this.backgroundColor = 'blue';
  }

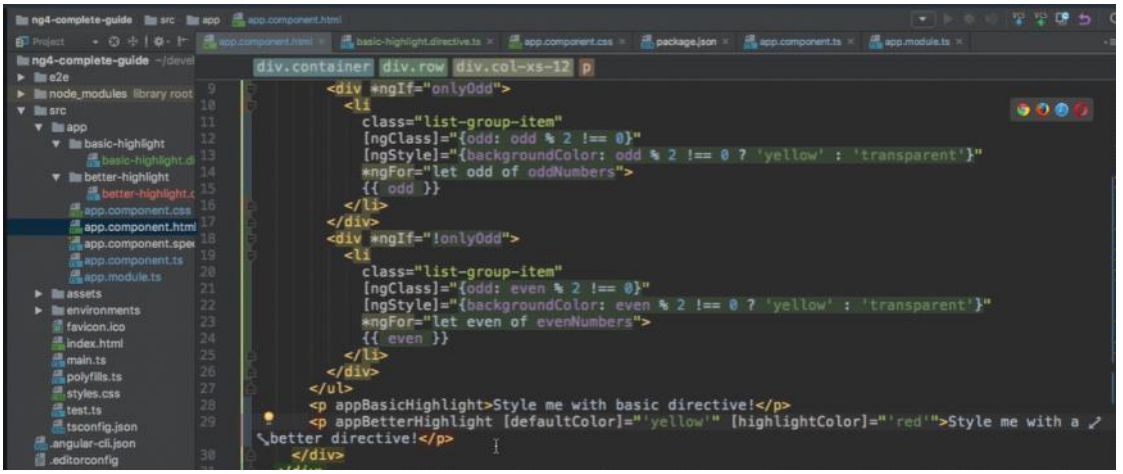
  @HostListener('mouseleave') mouseleave(eventData: Event) {
    // this.renderer.setStyle(this.elRef.nativeElement, 'background-color', 'transparent', false
  false);
    this.backgroundColor = 'transparent';
```

- Custom Property binding in Directive
  - 
```
@Directive({
  selector: '[appBetterHighlight]'
})
export class BetterHighlightDirective implements OnInit {
  @Input() defaultColor: string = 'transparent';
  @Input() highlightColor: string = 'blue';
  @HostBinding('style.backgroundColor') backgroundColor: string = this.defaultColor;

  constructor(private elRef: ElementRef, private renderer: RendererV2) { }

  ngOnInit() {
    // this.renderer.setStyle(this.elRef.nativeElement, 'background-color', 'blue', false, false)
  }

  @HostListener('mouseenter') mouseover(eventData: Event) {
    // this.renderer.setStyle(this.elRef.nativeElement, 'background-color', 'blue', false, false)
    this.backgroundColor = this.highlightColor;
  }

  @HostListener('mouseleave') mouseleave(eventData: Event) {
    // this.renderer.setStyle(this.elRef.nativeElement, 'background-color', 'transparent', false,
  false);
    this.backgroundColor = this.defaultColor;
  }
```
  - 
```
      <div *ngIf="onlyOdd">
        <li
          class="list-group-item"
          [ngClass]="{odd: odd % 2 !== 0}"
          [ngStyle]="{backgroundColor: odd % 2 !== 0 ? 'yellow' : 'transparent'}"
          *ngFor="let odd of oddNumbers">
          {{ odd }}
        </li>
      </div>
      <div *ngIf="!onlyOdd">
        <li
          class="list-group-item"
          [ngClass]="{odd: even % 2 !== 0}"
          [ngStyle]="{backgroundColor: even % 2 !== 0 ? 'yellow' : 'transparent'}"
          *ngFor="let even of evenNumbers">
          {{ even }}
        </li>
      </div>
    </ul>
    <p appBasicHighlight>Style me with basic directive!</p>
    <p appBetterHighlight [defaultColor]="'yellow'" [highlightColor]="'red'">Style me with a
better directive!</p>
  </div>
```

- Structural Directives
  - Behind the scenes, *ngIf which is a structural directive, * gets transformed into property binding.
  - It wraps the html containing *ngIf in ng-template with the property binding of ngIf

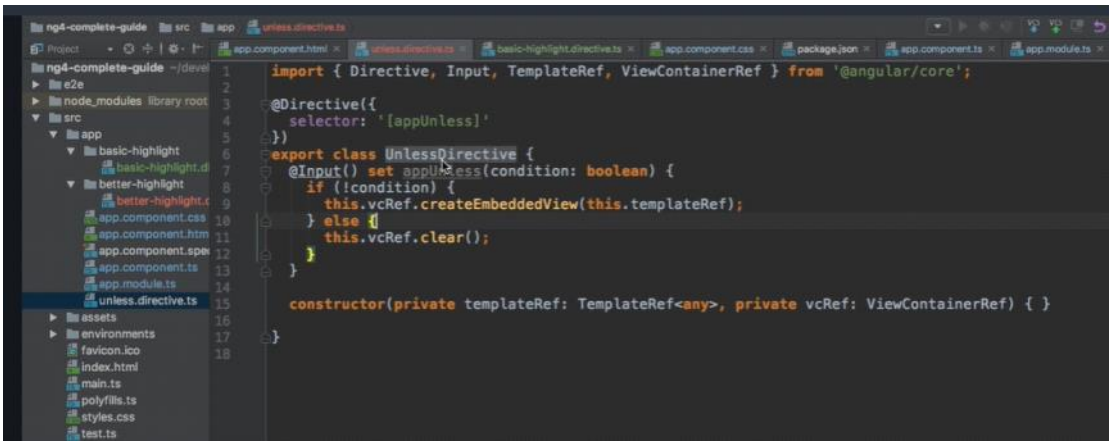
-
```
<div *ngIf="!onlyOdd">
  <li
    class="list-group-item"
    [ngClass]="{odd: even % 2 !== 0}"
    [ngStyle]="{backgroundColor: even % 2 !== 0 ? 'yellow' : 'transparent'}"
    *ngFor="let even of evenNumbers">
    {{ even }}
  </li>
</div>
<ng-template [ngIf]="!onlyOdd">
  <div>
    <li
      class="list-group-item"
      [ngClass]="{odd: even % 2 !== 0}"
      [ngStyle]="{backgroundColor: even % 2 !== 0 ? 'yellow' : 'transparent'}"
      *ngFor="let even of evenNumbers">
      {{ even }}
    </li>
  </div>
```

- Building Structural directives
  - ng g d unless
  - If we need to track the changes on an @Input property, we can use a set method along with it
    -
```
import { Directive, Input } from '@angular/core';

@Directive({
  selector: '[appUnless]'
})
export class UnlessDirective {
  @Input() set unless() {

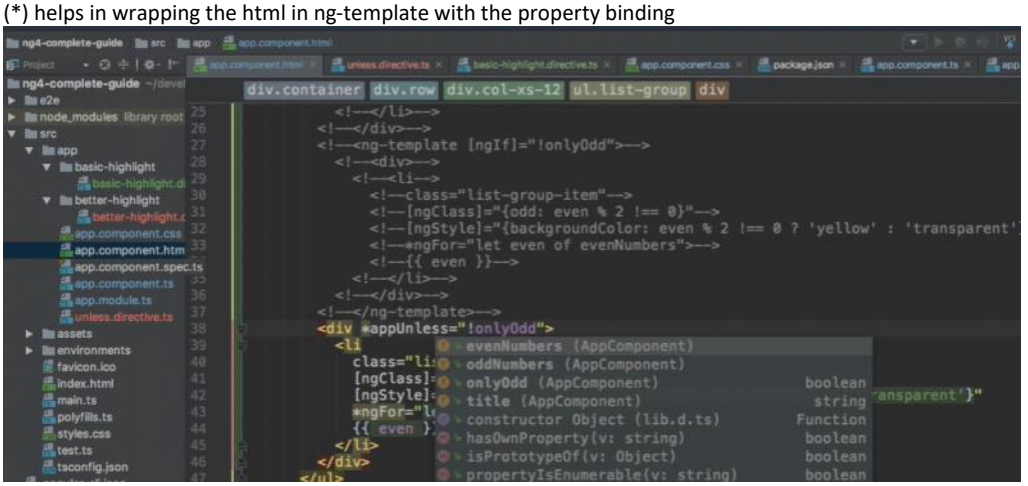
  }

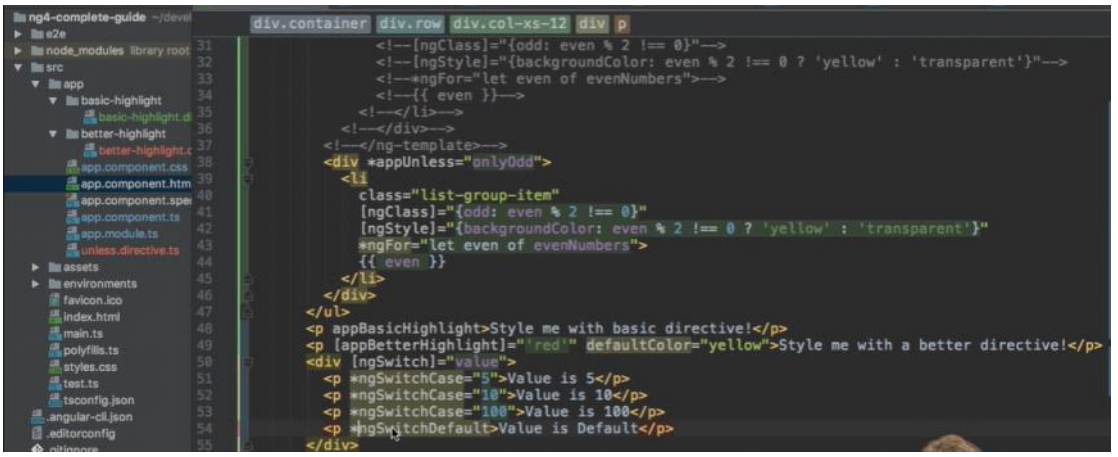
  constructor() { }

}
```
  - Template can be used in the directive through TemplateRef and it can be of type "any"
    - TemplateRef needs to be imported from '@angular/core'
  - We need to use get the ViewContainer reference to contain the template
    - ViewContainerRef needs to be imported from '@angular/core'
    - ViewContainerRef has createEmbeddedView method to create the view with the template
    - ViewContainerRef has clear method to clear the view
  - 
```
import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';

@Directive({
  selector: '[appUnless]'
})
export class UnlessDirective {
  @Input() set appUnless(condition: boolean) {
    if (!condition) {
      this.vcRef.createEmbeddedView(this.templateRef);
    } else {
      this.vcRef.clear();
    }
  }

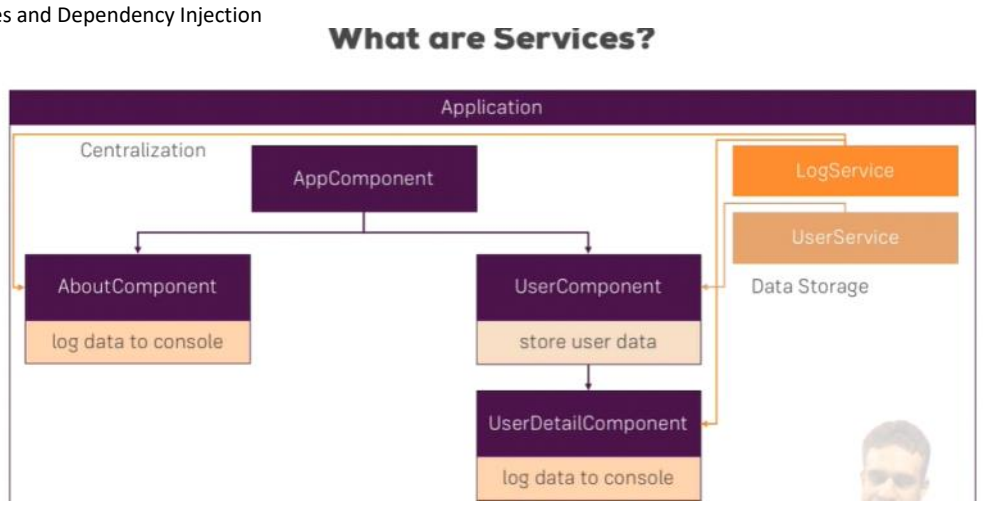
  constructor(private templateRef: TemplateRef<any>, private vcRef: ViewContainerRef) { }

}
```
  - Using the structural directive is as shown below
    - (*) helps in wrapping the html in ng-template with the property binding
    - 
```
<!--</li>-->
<!--</div>-->
<!--<ng-template [ngIf]="!onlyOdd">-->
  <!--<div>-->
    <!--<li-->
      <!--class="list-group-item"-->
      <!--[ngClass]="{odd: even % 2 !== 0}"-->
      <!--[ngStyle]="{backgroundColor: even % 2 !== 0 ? 'yellow' : 'transparent'}"-->
      <!--*ngFor="let even of evenNumbers">-->
      <!--{{ even }}-->
    <!--</li>-->
  <!--</div>-->
<!--</ng-template>-->
<div *appUnless="!onlyOdd">
  <li
    class="li
    [ngClass]
    [ngStyle]
    *ngFor="l
    {{ even }
  </li>
</div>
</ul>
```

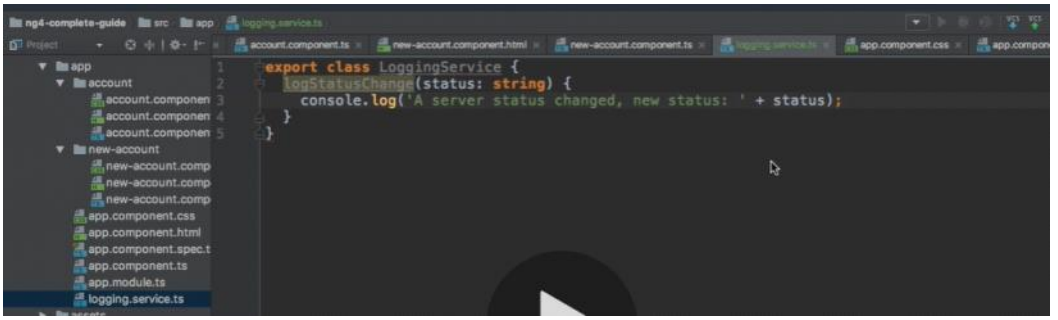
- ngSwitch Structural directive
  - 

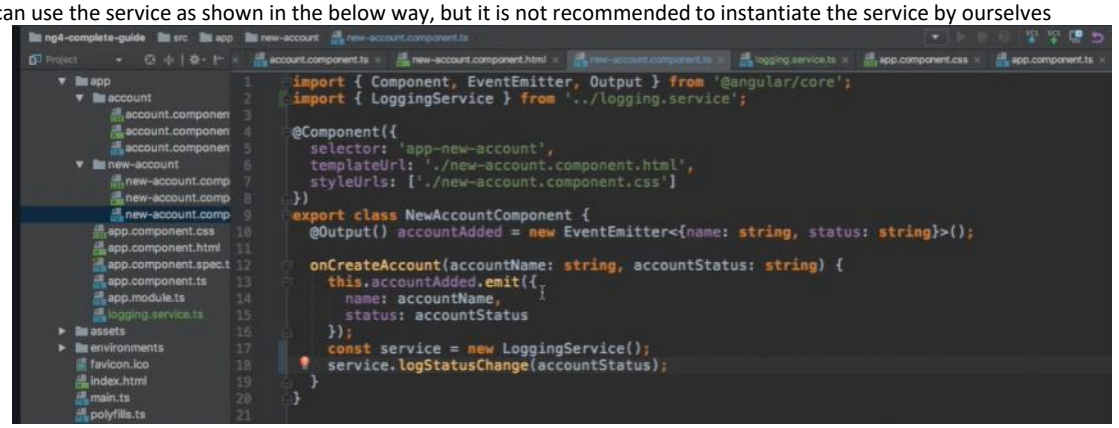
- Example of a dropdown directive
  -
```
import {Directive, ElementRef, HostBinding, HostListener} from '@angular/core';

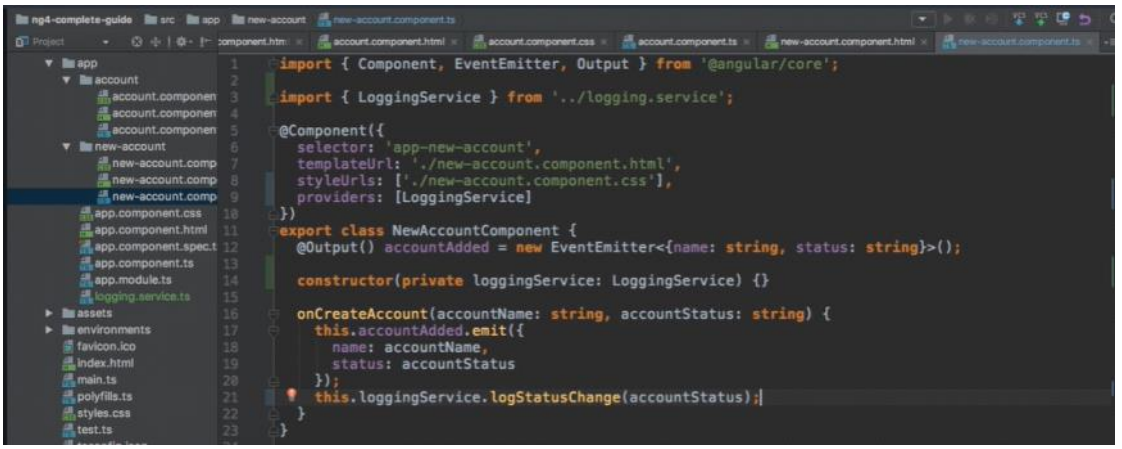
@Directive({
  selector: '[appDropdown]'
})
export class DropdownDirective {
  @HostBinding('class.open') isOpen = false;
  @HostListener('document:click', ['$event']) toggleOpen(event: Event) {
    this.isOpen = this.elRef.nativeElement.contains(event.target) ? !this.isOpen : false;
  }
  constructor(private elRef: ElementRef) {}
}
```

- Services and Dependency Injection
  - 

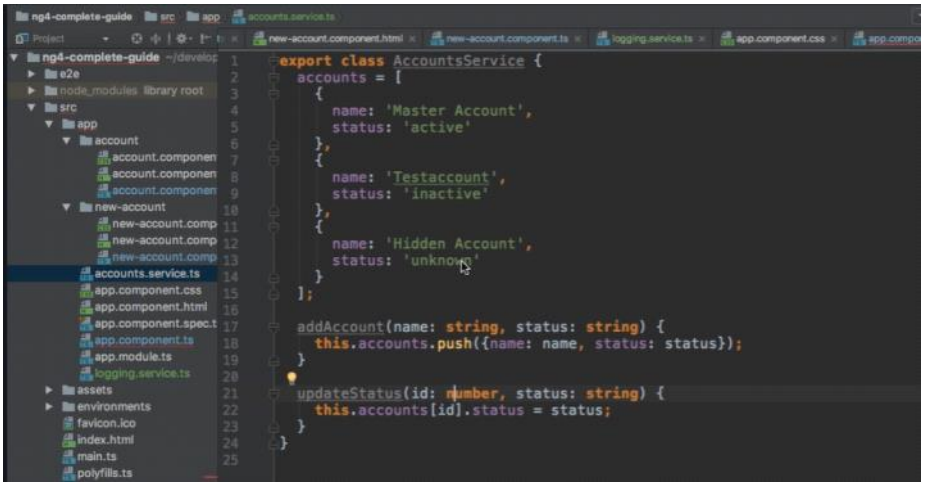
- Creating service component
  - Service is just a normal typescript which we export to be used in other places

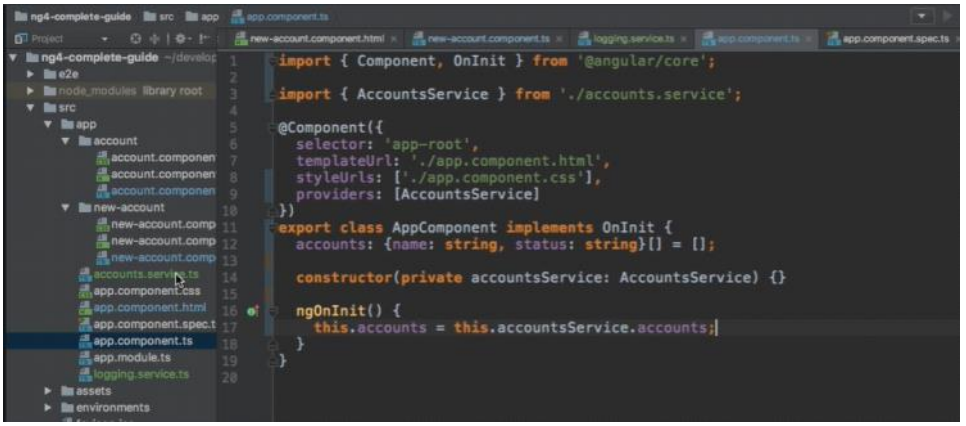
- 

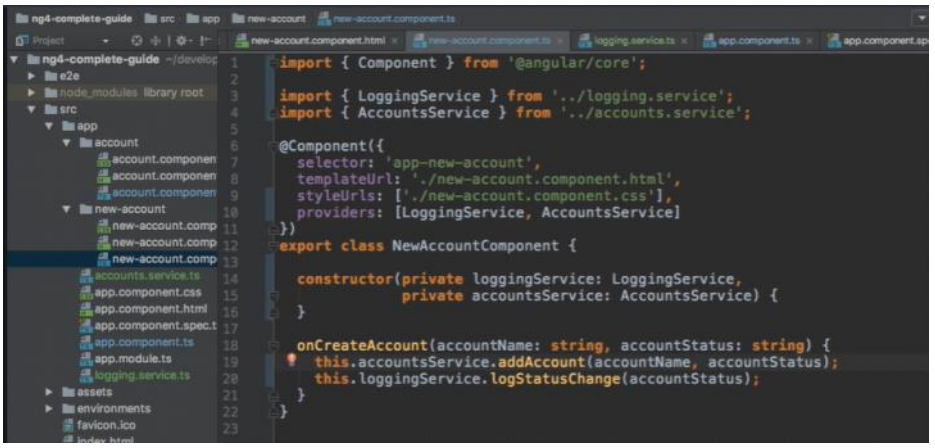
- We can use the service as shown in the below way, but it is not recommended to instantiate the service by ourselves
  - 

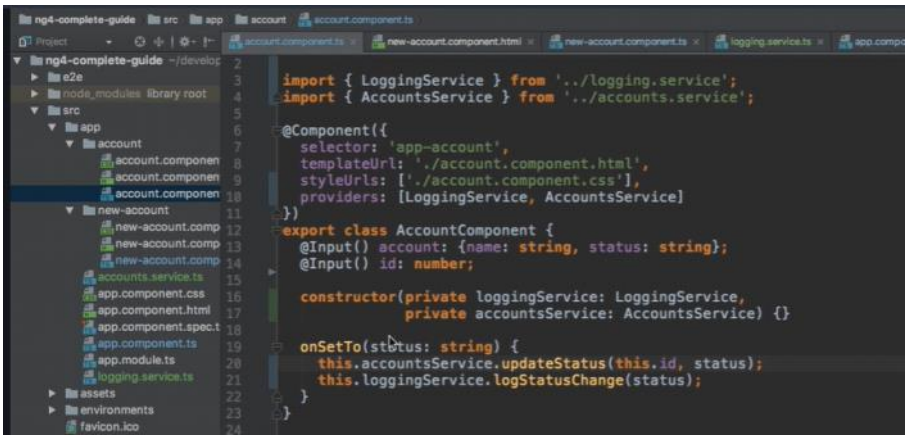
  - Angular has provided a way called Dependency Injection through which we can use the services .
  - So to use the service
    - We import the service component
    - We use the constructor method to instantiate
    - We need to add it in the providers in the Component decorator to recognize that we are using a service called LoggingService and also tell Angular how to create this LoggingService.
    - 

- Creating a data service

- 

- 

- 

- 

- Angular Dependency Injection is a Hierarchical Injector.

Hierarchical Injector

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 components**

- Now new account and accounts component have their own instances, so these are the other two instances of this service class.
- And therefore if we add a new service, we call add account, there we push it on this array of accounts and this actually works but it's a totally different instance of the service than the one we use here in app component, the one through which we loop.
- To fix this we simply, remove it from
  - the providers array.
- Don't remove it from the constructor, we of course do need it here because that tells Angular that we want an instance, the providers array basically tells which instance so there we need it to remove it and the same for account component.

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

import { AccountsService } from './accounts.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers: [AccountsService]
})
export class AppComponent implements OnInit {
  accounts: {name: string, status: string}[] = [];

  constructor(private accountsService: AccountsService) {}

  ngOnInit() {
    this.accounts = this.accountsService.accounts;
  }
}
```

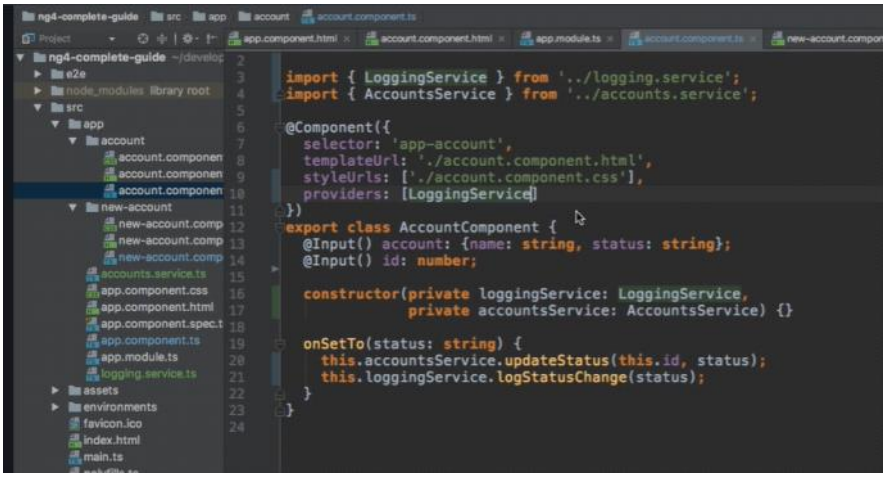
```
import { Component } from '@angular/core';

import { LoggingService } from '../logging.service';
import { AccountsService } from '../accounts.service';

@Component({
  selector: 'app-new-account',
  templateUrl: './new-account.component.html',
  styleUrls: ['./new-account.component.css'],
  providers: [LoggingService]
})
export class NewAccountComponent {

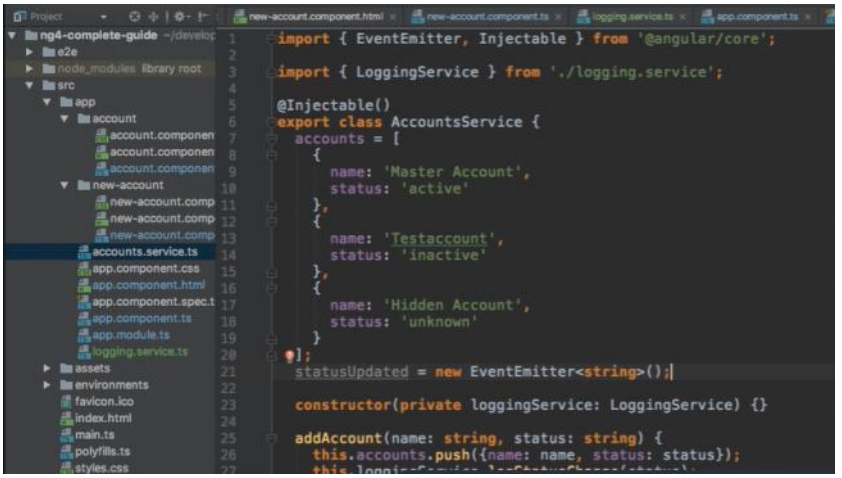
  constructor(private loggingService: LoggingService,
              private accountsService: AccountsService) {
  }

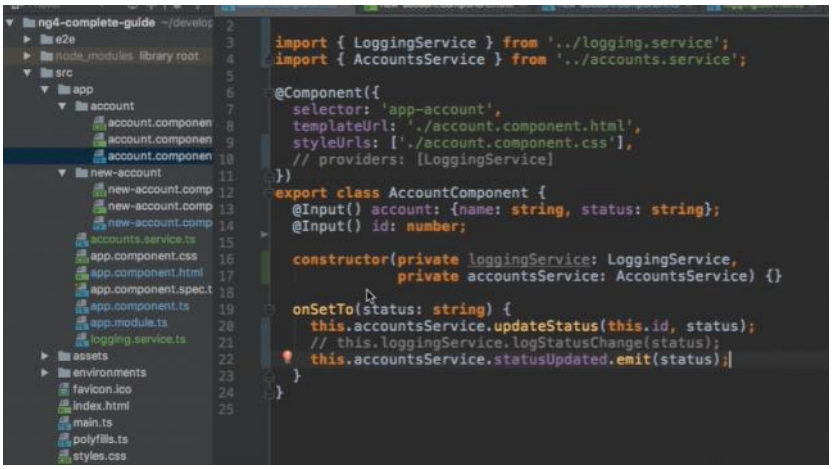
  onCreateAccount(accountName: string, accountStatus: string) {
    this.accountsService.addAccount(accountName, accountStatus);
    this.loggingService.logStatusChange(accountStatus);
  }
}
```

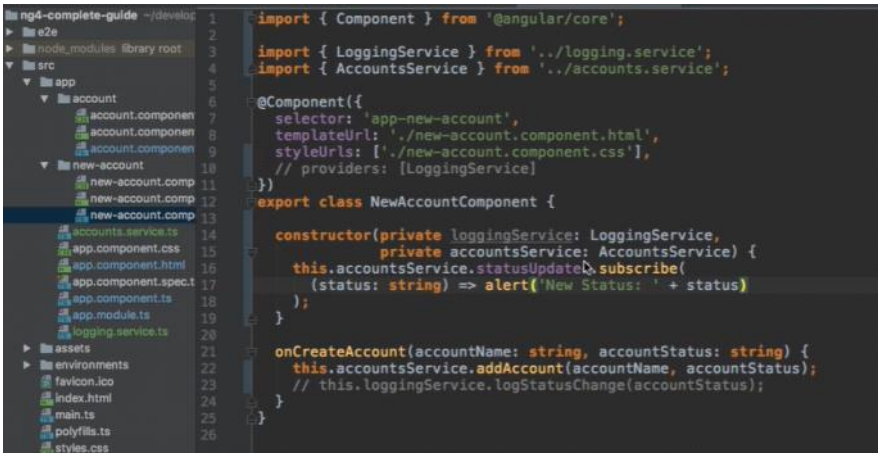
- 

- Injecting services
  - Use @Injectable() for the service to be used in providers of component metadata
  - Use @Injectable({ providedIn: 'root' }) to make the service available across the application

- Using services for cross component communication

- 

- 

- 

- Services in Angular 6+
  - 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()`:

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

    This is exactly the same as:

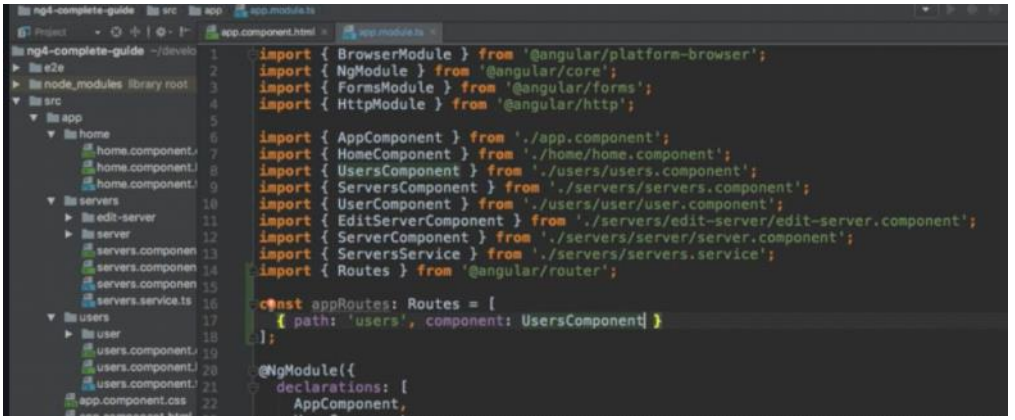
    ```
    export class MyService { ... }
    ```

    ```
    import { MyService } from './path/to/my.service';

    @NgModule({
        ...
        providers: [MyService]
    })
    export class AppModule { ... }
    ```

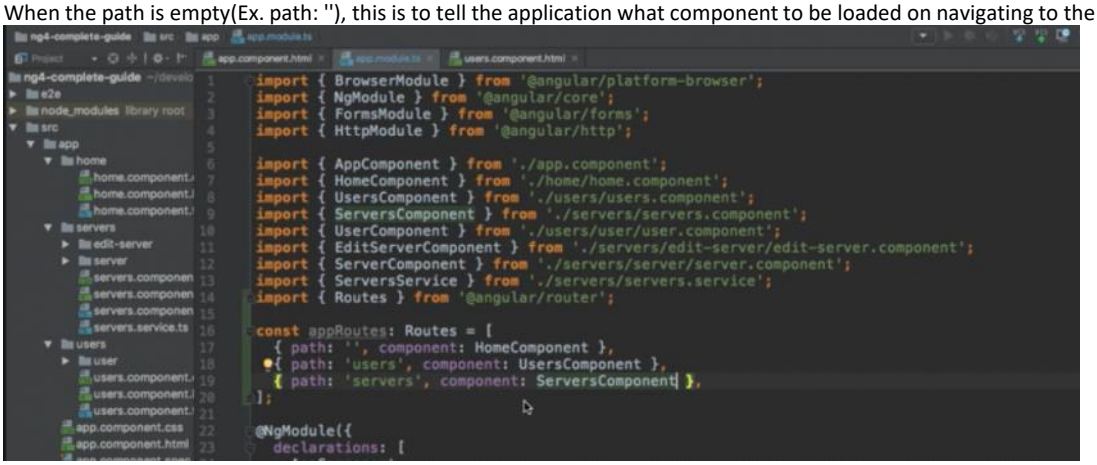
  - Using this new syntax is **completely optional**, the traditional syntax (using `providers[]`) will still work. The "new syntax" does offer one advantage though: Services **can be loaded lazily** by Angular (behind the scenes) and redundant code can be removed automatically. This can lead to a better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

- Routing
  - To route through the different pages in the application
  - We register the routes in the app module
  - We import Routes from library '@angular/router'

- 
```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';

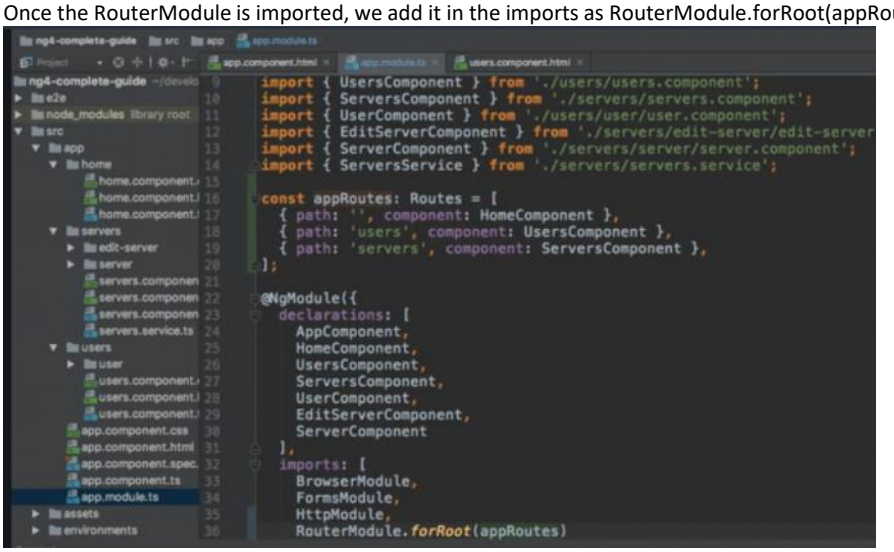
import { AppComponent } from './app.component';
import { HomeComponent } from './home/home.component';
import { UsersComponent } from './users/users.component';
import { ServersComponent } from './servers/servers.component';
import { UserComponent } from './users/user/user.component';
import { EditServerComponent } from './servers/edit-server/edit-server.component';
import { ServerComponent } from './servers/server/server.component';
import { ServersService } from './servers/servers.service';
import { Routes } from '@angular/router';

const appRoutes: Routes = [
  { path: 'users', component: UsersComponent }
];

@NgModule({
  declarations: [
    AppComponent,
```
- No need to add slash in the path.
- We need to mention the component to be loaded once the path has been navigated to.
- When the path is empty(Ex. path: ''), this is to tell the application what component to be loaded on navigating to the base url
- 
```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';

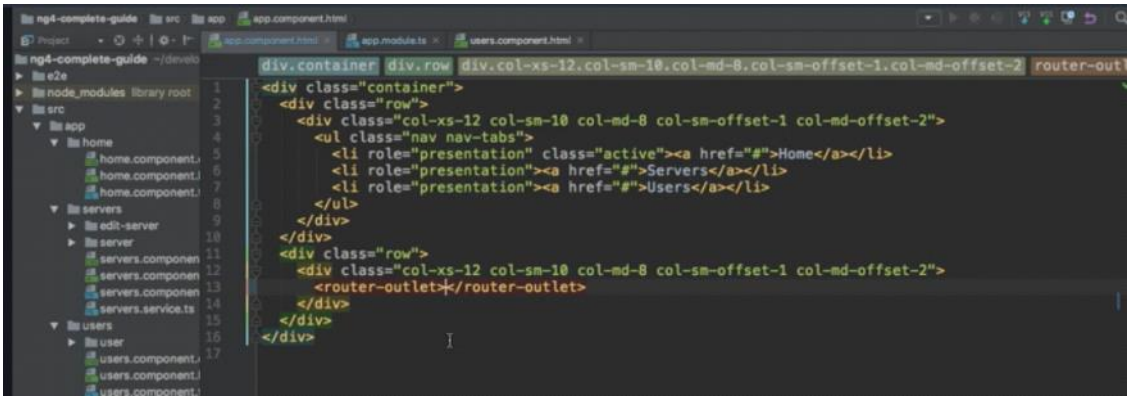
import { AppComponent } from './app.component';
import { HomeComponent } from './home/home.component';
import { UsersComponent } from './users/users.component';
import { ServersComponent } from './servers/servers.component';
import { UserComponent } from './users/user/user.component';
import { EditServerComponent } from './servers/edit-server/edit-server.component';
import { ServerComponent } from './servers/server/server.component';
import { ServersService } from './servers/servers.service';
import { Routes } from '@angular/router';

const appRoutes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'users', component: UsersComponent },
  { path: 'servers', component: ServersComponent },
];

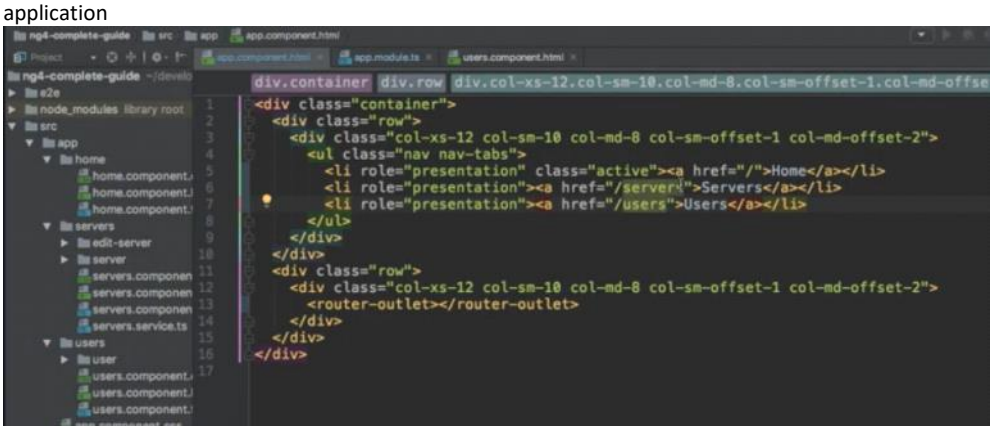
@NgModule({
  declarations: [
```
- Once the appRoutes has been defined, we need to register these routes by importing the RouterModule from library '@angular/router'
  - Once the RouterModule is imported, we add it in the imports as RouterModule.forRoot(appRoutes)
  - 
```
import { UsersComponent } from './users/users.component';
import { ServersComponent } from './servers/servers.component';
import { UserComponent } from './users/user/user.component';
import { EditServerComponent } from './servers/edit-server/edit-server
import { ServerComponent } from './servers/server/server.component';
import { ServersService } from './servers/servers.service';

const appRoutes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'users', component: UsersComponent },
  { path: 'servers', component: ServersComponent },
];

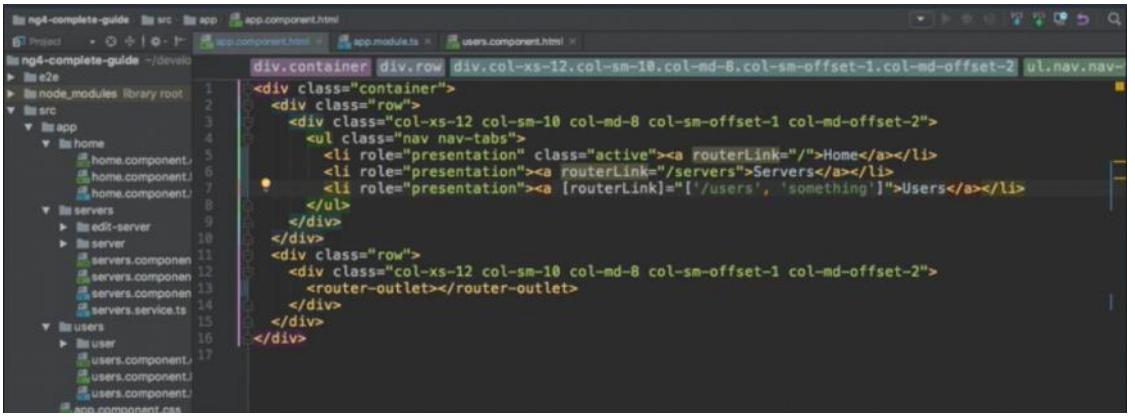
@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    UsersComponent,
    ServersComponent,
    UserComponent,
    EditServerComponent,
    ServerComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    RouterModule.forRoot(appRoutes)
```

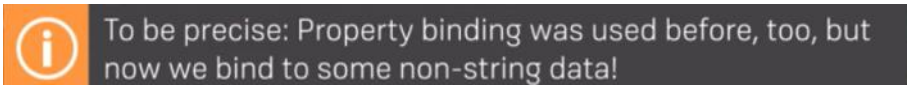
- 

  - Next we need to mention <router-outlet></router-outlet> to let Angular mark a place in the document where the router needs to load the component for that particular route.
  - We can also load modules related to a route
    -

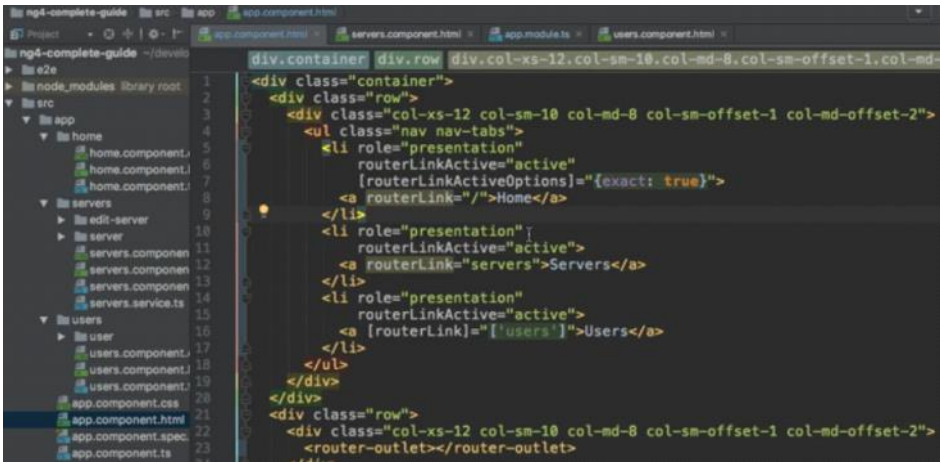
- Navigating with Router Links
  - We can directly give the routes to the href links as in the below screenshot
    - If we directly add the routes to href, the app refreshes the whole application on every route change and with this the application state also gets refreshed which is not intended and not a best experience for the user in an single page application
    - 

  - So, to overcome the above we use routerLink directive
    - routerLink tells angular

- 

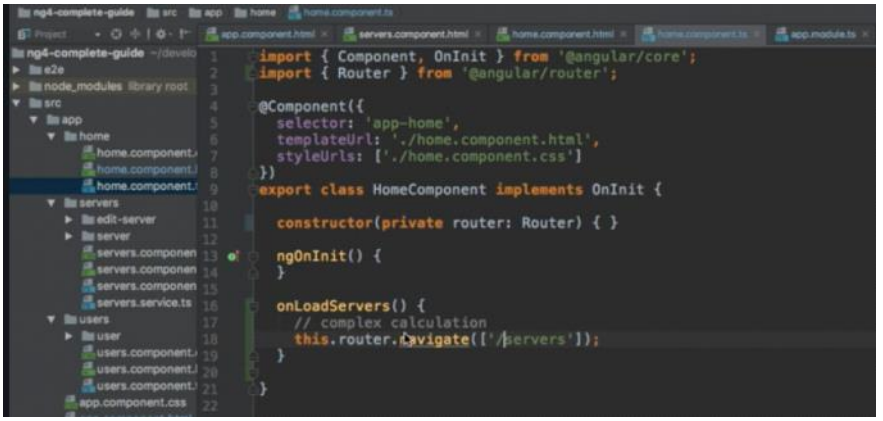
- 

- In routerLink directive
  - The relative paths are pre-fixed with "./" or nothing
    - So with a relative path, it always append the path you specify in the routerLink to the end of your current path and important here, the current path depends on which component you are currently on.
  - The absolute path are pre-fixed with "/"
  - We need to make sure we are not loading the relative path instead of the absolute path.

- Styling Active Router Links
  - routerLinkActive="<class-name>" will allow us to specify the class for making the link active
  - routerLinkActive analyzes your currently loaded path and then checks which links lead to a route which uses this path and by default, it marks an element as active, it marks this CSS class if it contains the path you are on or if this link is part of the path which is currently loaded.
  - routerLinkActiveOptions is an attribute directive in which we can send an object to pass on the information to make the link active based on the options
    - Ex.
    - 

    - Here as shown in the above screenshot, we pass a Javascript object and that would not work without having our square brackets.
    - So with square brackets, we can pass anything which will be resolved dynamically, like this Javascript object
    - Also here, we can basically add one important configuration, say exact and set this to true. So exact is a kind of a reserved property on this object you passed to routerLink active options and this will now basically tell Angular only add this routerLink active class.

- Navigating programmatically

- 
```
import { Component, OnInit } from '@angular/core';
import { Router } from '@angular/router';

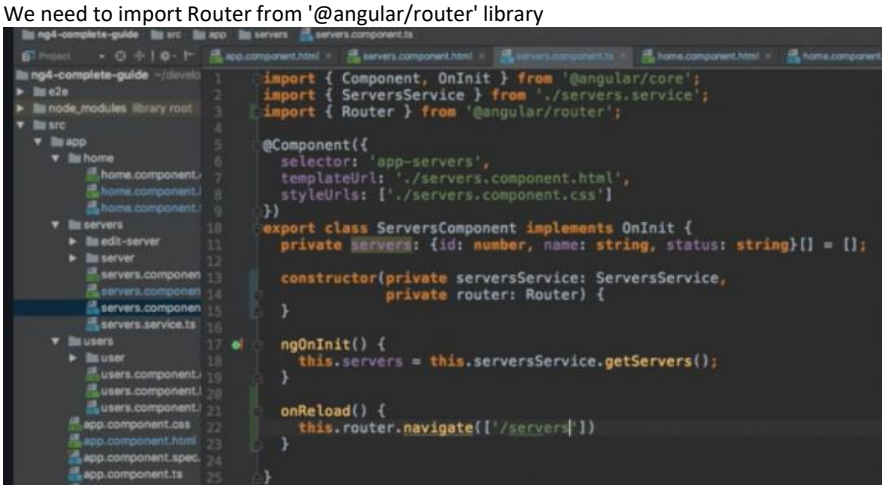
@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.css']
})
export class HomeComponent implements OnInit {

  constructor(private router: Router) { }

  ngOnInit() {
  }

  onLoadServers() {
    // complex calculation
    this.router.navigate(['/servers']);
  }

}
```
  - Router needs to be imported from '@angular/router' and added in the constructor.
  - Use this.router.navigate to navigate to the specified route.

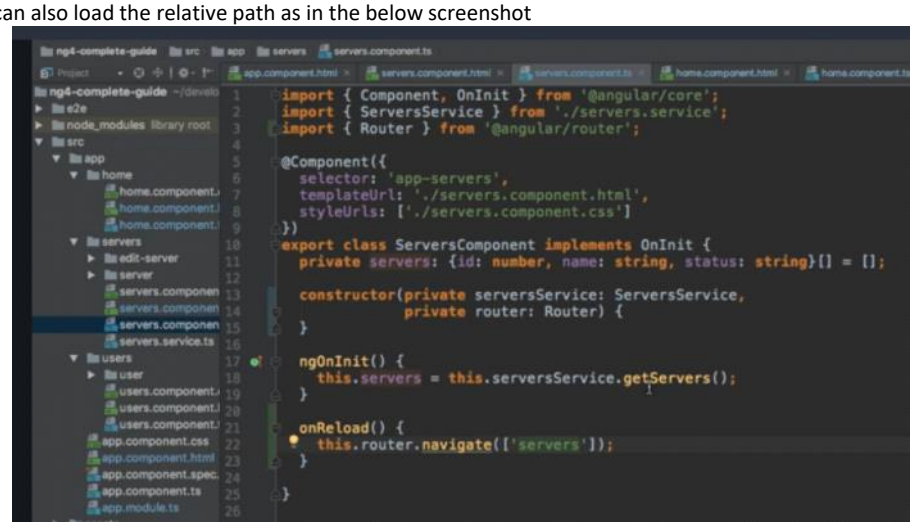
- Using Relative paths for programmatic navigation
  - We need to import Router from '@angular/router' library
  - 
```
import { Component, OnInit } from '@angular/core';
import { ServersService } from './servers.service';
import { Router } from '@angular/router';

@Component({
  selector: 'app-servers',
  templateUrl: './servers.component.html',
  styleUrls: ['./servers.component.css']
})
export class ServersComponent implements OnInit {
  private servers: {id: number, name: string, status: string}[] = [];

  constructor(private serversService: ServersService,
              private router: Router) {
  }

  ngOnInit() {
    this.servers = this.serversService.getServers();
  }

  onReload() {
    this.router.navigate(['/servers'])
  }

}
```
  - We can also load the relative path as in the below screenshot
    - 
```
import { Component, OnInit } from '@angular/core';
import { ServersService } from './servers.service';
import { Router } from '@angular/router';

@Component({
  selector: 'app-servers',
  templateUrl: './servers.component.html',
  styleUrls: ['./servers.component.css']
})
export class ServersComponent implements OnInit {
  private servers: {id: number, name: string, status: string}[] = [];

  constructor(private serversService: ServersService,
              private router: Router) {
  }

  ngOnInit() {
    this.servers = this.serversService.getServers();
  }

  onReload() {
    this.router.navigate(['servers']);
  }

}
```
    - Unlike the routerLink, the navigate method doesn't know on which route you are currently on, so to say. The routerLink always knows in which component it sits, in which components template and therefore it knows what the currently loaded route is.

    - We can also add an second argument to configure the navigation action.
      - Here we define relative to which route this link should be loaded and by default.
      - Here we have to give a route though, so we don't pass a string here, instead the route is something we can inject here too.
      - We can get the currently active route by injecting route which is of type ActivatedRoute and make sure to import ActivatedRoute from the @angular/router package too.

- Now ActivatedRoute like the name implies simply injects the currently active routes, so for the component you loaded, this will be the route which loaded this component and the route simply is kind of a complex js object which keeps a lot of meta information about the currently active route.
- Now we can set this as a value, so this route, this injected route for the relativeTo property and now with that extra piece of information, Angular knows what our currently active route is and theoretically you could also set this to some other route which you would have to get by some way.
- So here, we are telling it now our currently active routes is this ActivatedRoute, so relative to this route you should navigate and then it will simply resolve all the relative paths you might have here relative to this route.
-

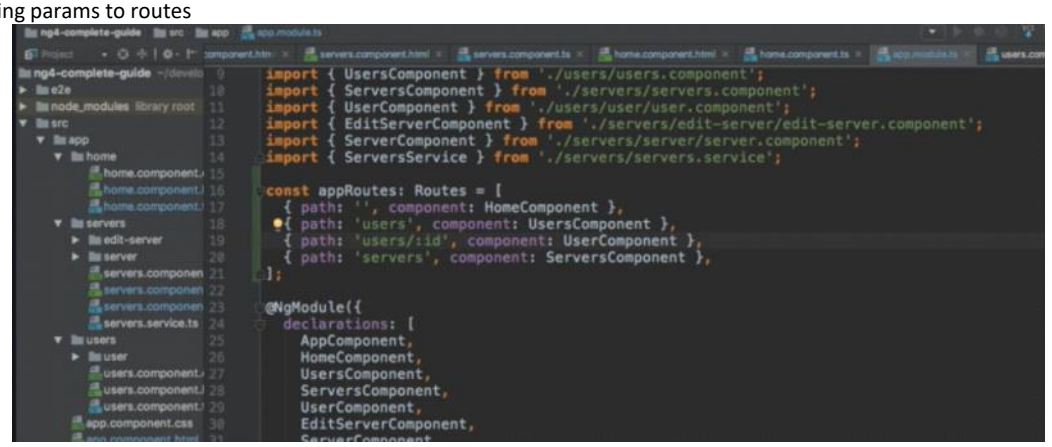
```
import { Component, OnInit } from '@angular/core';
import { ServersService } from './servers.service';
import { Router, ActivatedRoute } from '@angular/router';

@Component({
  selector: 'app-servers',
  templateUrl: './servers.component.html',
  styleUrls: ['./servers.component.css']
})
export class ServersComponent implements OnInit {
  private servers: {id: number, name: string, status: string}[] = [];

  constructor(private serversService: ServersService,
              private router: Router,
              private route: ActivatedRoute) {
  }

  ngOnInit() {
    this.servers = this.serversService.getServers();
  }

  onReload() {
    this.router.navigate(['servers'], {relativeTo: this.route});
  }
```

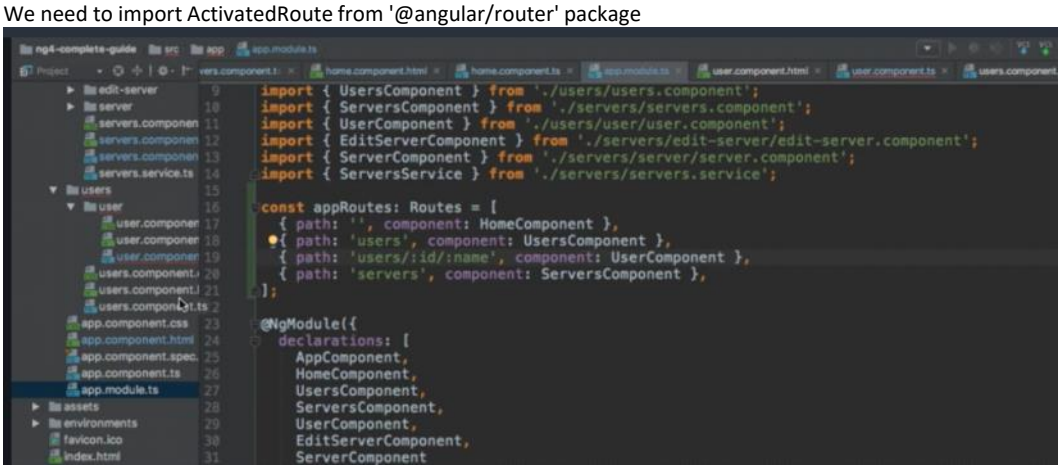
- Passing params to routes
- 

```
import { UsersComponent } from './users/users.component';
import { ServersComponent } from './servers/servers.component';
import { UserComponent } from './users/user/user.component';
import { EditServerComponent } from './servers/edit-server/edit-server.component';
import { ServerComponent } from './servers/server/server.component';
import { ServersService } from './servers/servers.service';

const appRoutes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'users', component: UsersComponent },
  { path: 'users/:id', component: UserComponent },
  { path: 'servers', component: ServersComponent },
];

@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    UsersComponent,
    ServersComponent,
    UserComponent,
    EditServerComponent,
```

  - Colon(:) when prefixed to a route tells angular that it is a dynamic parameter

- Fetching route params
  - We need to import ActivatedRoute from '@angular/router' package
  - 

```
import { UsersComponent } from './users/users.component';
import { ServersComponent } from './servers/servers.component';
import { UserComponent } from './users/user/user.component';
import { EditServerComponent } from './servers/edit-server/edit-server.component';
import { ServerComponent } from './servers/server/server.component';
import { ServersService } from './servers/servers.service';

const appRoutes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'users', component: UsersComponent },
  { path: 'users/:id/:name', component: UserComponent },
  { path: 'servers', component: ServersComponent },
];

@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    UsersComponent,
    ServersComponent,
    UserComponent,
    EditServerComponent,
    ServerComponent
```

- To avoid confusion: The ActivatedRoute object we injected will give us access to the id passed in the URL => Selected User

- ```
  import { Component, OnInit } from '@angular/core';
  import { ActivatedRoute } from '@angular/router';

  @Component({
    selector: 'app-user',
    templateUrl: './user.component.html',
    styleUrls: ['./user.component.css']
  })
  export class UserComponent implements OnInit {
    user: {id: number, name: string};

    constructor(private route: ActivatedRoute) { }

    ngOnInit() {
    }

  }
  ```
- To fetch the param from the url we need to code as in below screenshot
- ```
  import { Component, OnInit } from '@angular/core';
  import { ActivatedRoute } from '@angular/router';

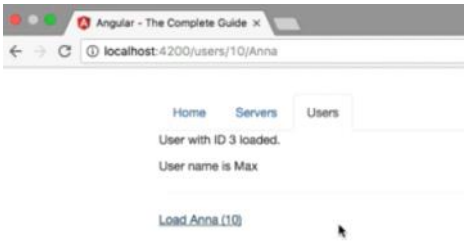
  @Component({
    selector: 'app-user',
    templateUrl: './user.component.html',
    styleUrls: ['./user.component.css']
  })
  export class UserComponent implements OnInit {
    user: {id: number, name: string};

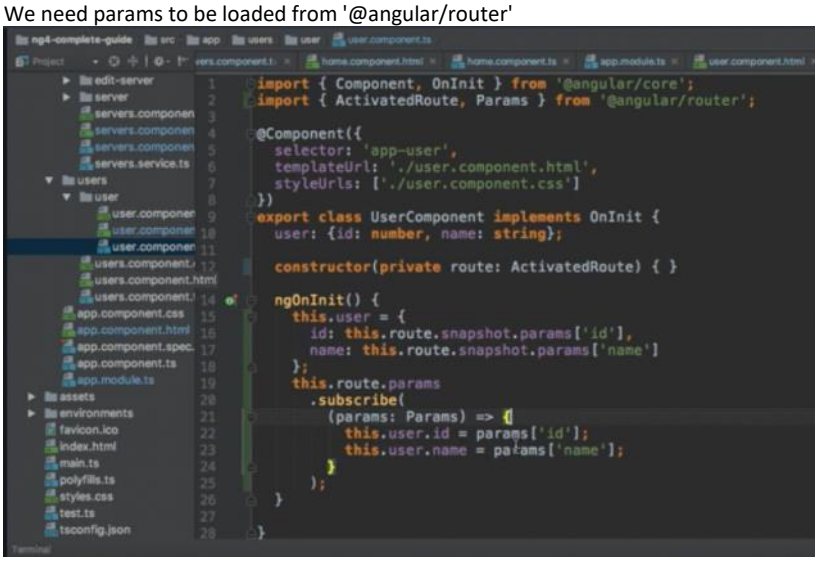
    constructor(private route: ActivatedRoute) { }

    ngOnInit() {
      this.user = {
        id: this.route.snapshot.params['id'],
        name: this.route.snapshot.params['name']
      };
    }

  }
  ```

- Fetching route params reactively
- ```
  User with ID {{ user.id }} loaded.
  User name is {{ user.name }}
  <hr>
  <a [routerLink]="['/users', 10, 'Anna']"></a>
  ```
  /users/10/Anna
- ```
  <p>User with ID {{ user.id }} loaded.</p>
  <p>User name is {{ user.name }}</p>
  <hr>
  <a [routerLink]="['/users', 10, 'Anna']">Load Anna (10)</a>
  ```
- In the above approach the url gets updated but not the text in this template

- 

- Because as we wrote the data in ngOnInit and the component will not know that the route params changed.
- We can use this.route.params to resolve this, which is an observable by which we can follow the changes in the route params by subscribing to it
  - We need params to be loaded from '@angular/router'
  - 
```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, Params } from '@angular/router';

@Component({
  selector: 'app-user',
  templateUrl: './user.component.html',
  styleUrls: ['./user.component.css']
})
export class UserComponent implements OnInit {
  user: {id: number, name: string};

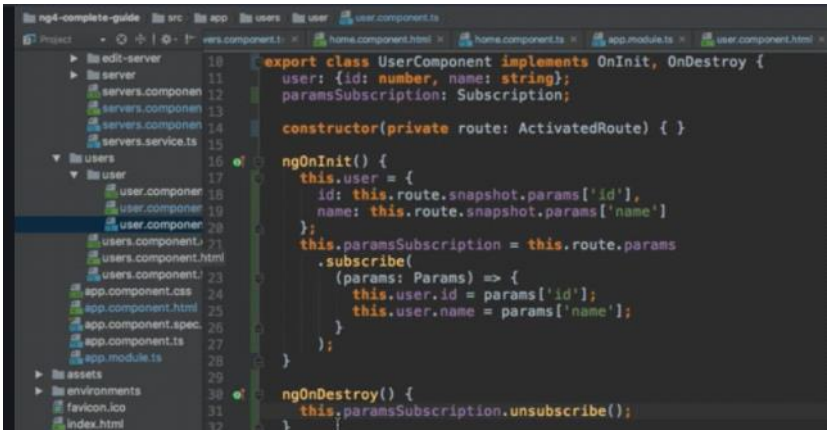
  constructor(private route: ActivatedRoute) { }

  ngOnInit() {
    this.user = {
      id: this.route.snapshot.params['id'],
      name: this.route.snapshot.params['name']
    };
    this.route.params
      .subscribe(
        (params: Params) => {
          this.user.id = params['id'];
          this.user.name = params['name'];
        }
      );
  }

}
```
  - The subscriptions for an observable will be cleaned up when the component is destroyed.
  - But to clean up the subscription we need to import subscription from 'rxjs/subscription' and do the following when we are writing our own observables
  -
```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { ActivatedRoute, Params } from '@angular/router';
import { Subscription } from 'rxjs/Subscription';

@Component({
  selector: 'app-user',
  templateUrl: './user.component.html',
  styleUrls: ['./user.component.css']
})
export class UserComponent implements OnInit, OnDestroy {
  user: {id: number, name: string};
  paramsSubscription: Subscription;

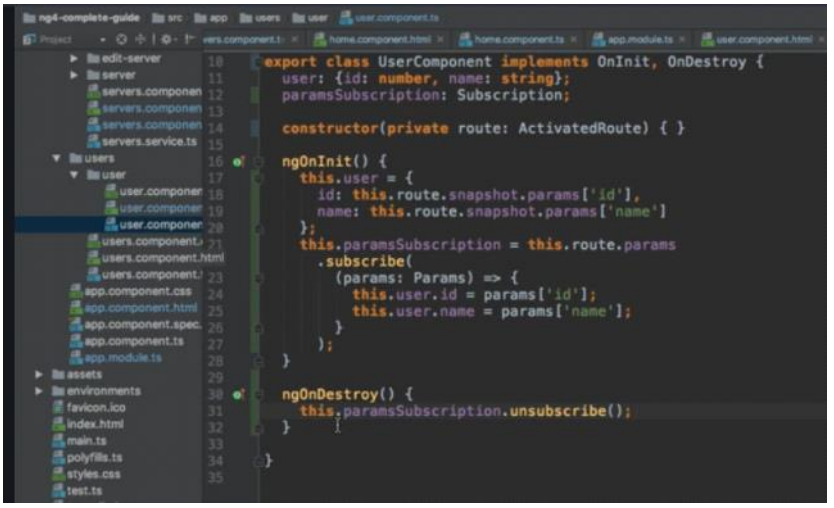
  constructor(private route: ActivatedRoute) { }

  ngOnInit() {
    this.user = {
      id: this.route.snapshot.params['id'],
      name: this.route.snapshot.params['name']
    };
```
  - 
```
export class UserComponent implements OnInit, OnDestroy {
  user: {id: number, name: string};
  paramsSubscription: Subscription;

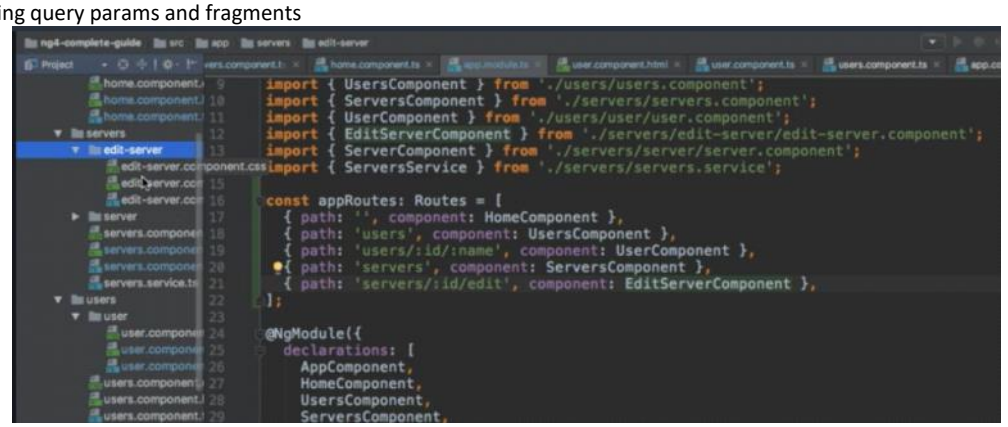
  constructor(private route: ActivatedRoute) { }

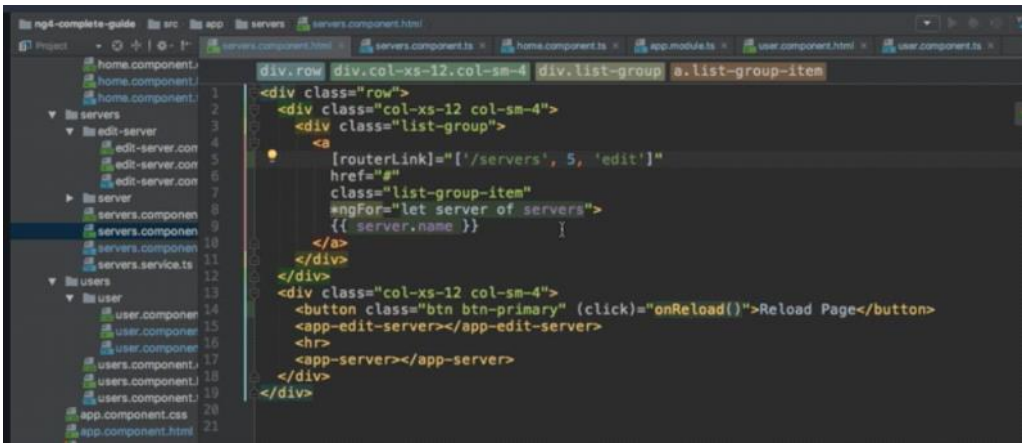
  ngOnInit() {
    this.user = {
      id: this.route.snapshot.params['id'],
      name: this.route.snapshot.params['name']
    };
    this.paramsSubscription = this.route.params
      .subscribe(
        (params: Params) => {
          this.user.id = params['id'];
          this.user.name = params['name'];
        }
      );
  }

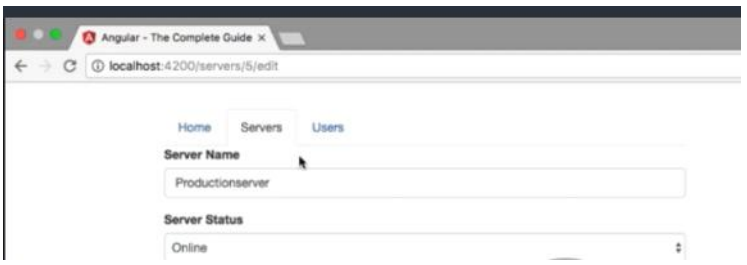
  ngOnDestroy() {
    this.paramsSubscription.unsubscribe();
```

- 

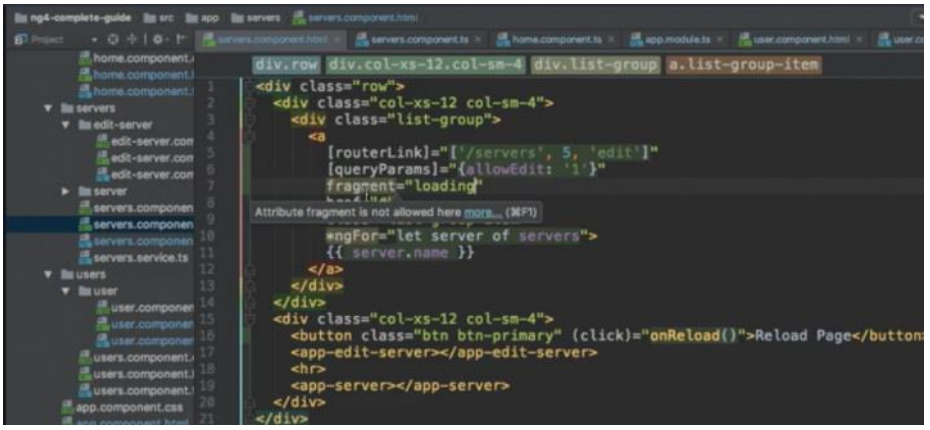
□ Passing query params and fragments

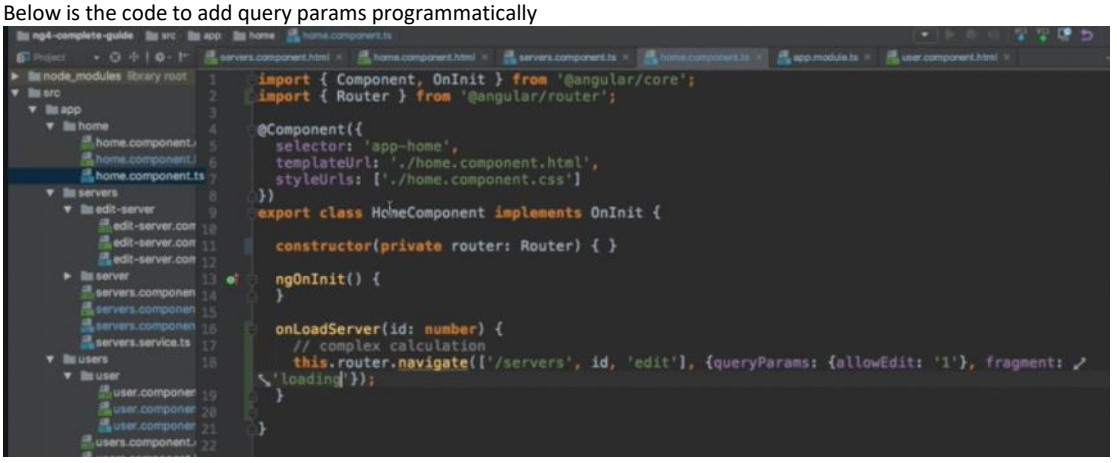
- 

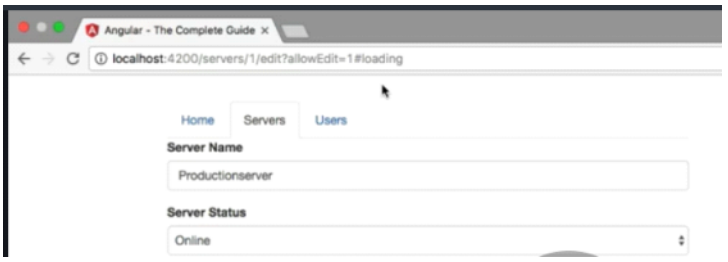
- 

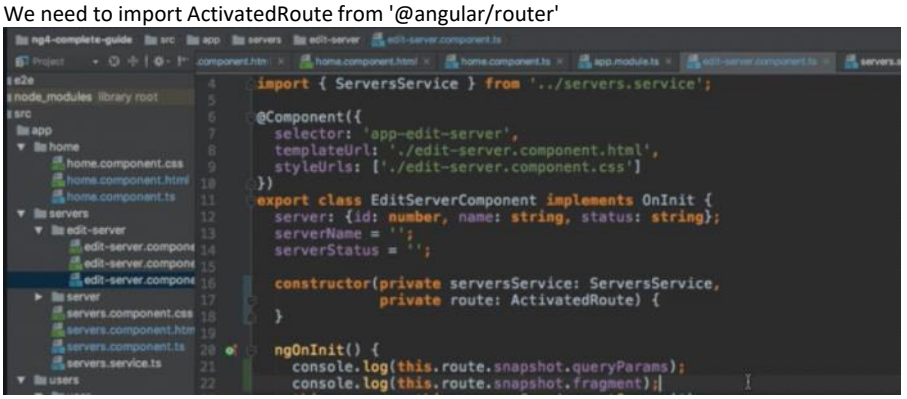
- 

- Now to add the query params
  - We have queryParams as inbuilt in routerLink to add the queryParams
  - We also have fragments to add #loading to the url

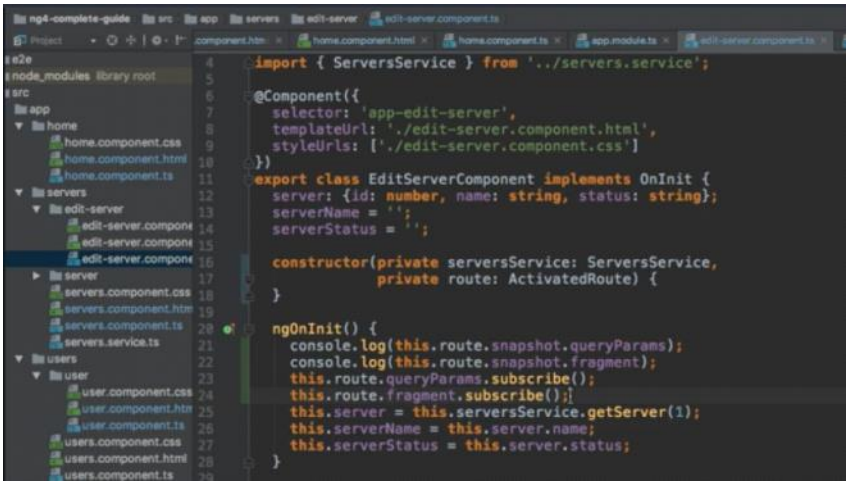
- 

- Below is the code to add query params programmatically
- 

- 

- Retrieving query params and fragments
  - We need to import ActivatedRoute from '@angular/router'
  - 

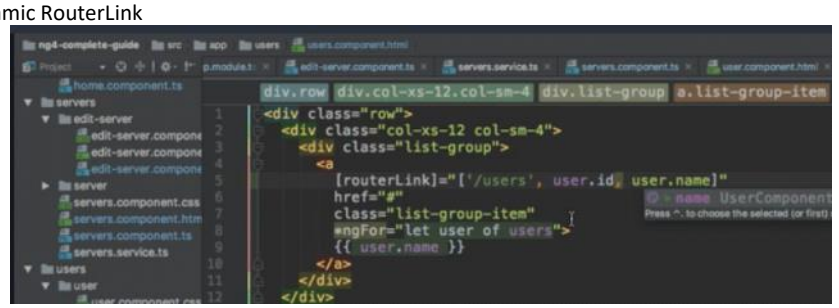
  - To make it reactive, we need to use the queryParams observable

- 
```
import { ServersService } from '../servers.service';

@Component({
  selector: 'app-edit-server',
  templateUrl: './edit-server.component.html',
  styleUrls: ['./edit-server.component.css']
})
export class EditServerComponent implements OnInit {
  server: {id: number, name: string, status: string};
  serverName = '';
  serverStatus = '';

  constructor(private serversService: ServersService,
              private route: ActivatedRoute) {
  }

  ngOnInit() {
    console.log(this.route.snapshot.queryParams);
    console.log(this.route.snapshot.fragment);
    this.route.queryParams.subscribe();
    this.route.fragment.subscribe();
    this.server = this.serversService.getServer(1);
    this.serverName = this.server.name;
    this.serverStatus = this.server.status;
  }
```

- Dynamic RouterLink
  - 
```
<div class="row">
  <div class="col-xs-12 col-sm-4">
    <div class="list-group">
      <a
        [routerLink]="['/users', user.id, user.name]"
        href="#"
        class="list-group-item"
        *ngFor="let user of users">
        {{ user.name }}
      </a>
    </div>
  </div>
```

- Nested Routing
  - Nested Routing is done using the children array in an particular path
    -
```
import { HttpModule } from '@angular/http';
import { Routes, RouterModule } from '@angular/router';

import { AppComponent } from './app.component';
import { HomeComponent } from './home/home.component';
import { UsersComponent } from './users/users.component';
import { ServersComponent } from './servers/servers.component';
import { UserComponent } from './users/user/user.component';
import { EditServerComponent } from './servers/edit-server/edit-server
import { ServerComponent } from './servers/server/server.component';
import { ServersService } from './servers/servers.service';

const appRoutes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'users', component: UsersComponent },
  { path: 'users/:id/:name', component: UserComponent },
  { path: 'servers', component: ServersComponent, children: [
    { path: ':id', component: ServerComponent },
    { path: ':id/edit', component: EditServerComponent }
  ] }
];

@NgModule({
```
  - Nested routing needs a <router-outlet> of its own
    -
```
<div class="row">
  <div class="col-xs-12 col-sm-4">
    <div class="list-group">
      <a
        [routerLink]="['/servers', server.id]"
        [queryParams]="{allowEdit: '1'}"
        fragment="loading"
        href="#"
        class="list-group-item"
        *ngFor="let server of servers">
        {{ server.name }}
      </a>
    </div>
  </div>
  <div class="col-xs-12 col-sm-4">
    <router-outlet></router-outlet>
    <!--<button class="btn btn-primary" (click)="onReload()">Reload Pag
    <!--<app-edit-server></app-edit-server>-->
    <!--<hr>-->
    <!--&lt;!&ndash;<app-server></app-server>&ndash;&gt;-->
  </div>
</div>
```
  - All the children of servers are loaded in the router-outlet

- Configuring the handling of query params
  - We have an queryParamHandling in the dynamic navigate

- 'merge' -> To merge with the existing query params with the new query params
- 'preserve' -> Overrides the default behavior drop them and make sure the old ones are preserved
-

```
constructor(private serversService: ServersService,
            private route: ActivatedRoute,
            private router: Router) {
}

ngOnInit() {
  const id = +this.route.snapshot.params['id'];
  this.server = this.serversService.getServer(id);
  this.route.params
    .subscribe(
      (params: Params) => {
        this.server = this.serversService.getServer(+params['id']);
      }
    );
}

onEdit() {
  this.router.navigate(['edit'], {relativeTo: this.route, queryParamsHandling: 'preserve'});
}
}
```

□ Redirecting and Wild card routes

-

```
import { ServersService } from './servers/servers.service';
import { PageNotFoundComponent } from './page-not-found/page-not-found.

const appRoutes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'users', component: UsersComponent, children: [
    { path: ':id/:name', component: UserComponent }
  ] },
  { path: 'servers', component: ServersComponent, children: [
    { path: ':id', component: ServerComponent },
    { path: ':id/edit', component: EditServerComponent }
  ] },
  { path: 'something', component: PageNotFoundComponent }
];

@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    UsersComponent,
    ServersComponent,
    UserComponent,
    EditServerComponent,
    ServerComponent,
    PageNotFoundComponent
  ],
  imports: [
```

- To redirect to another path instead of loading a component can be done in the following way
-

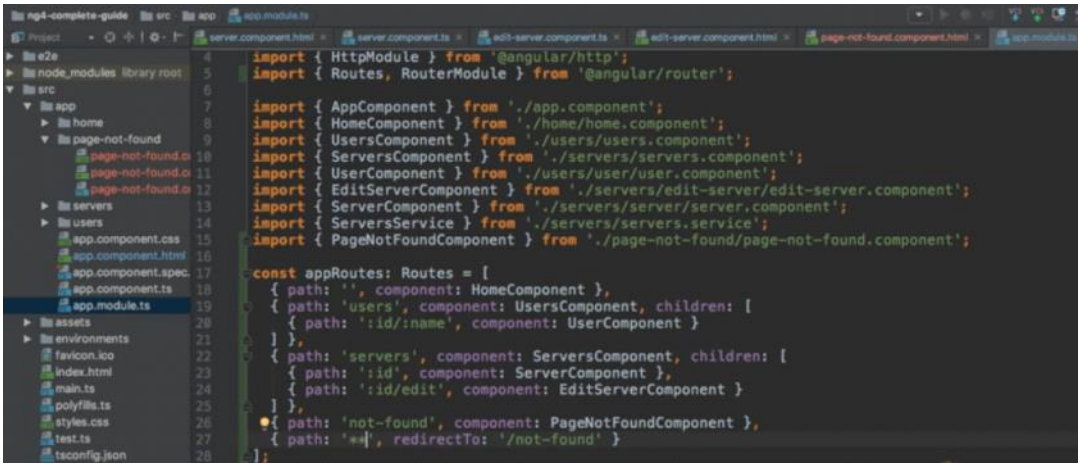
```
import { HttpModule } from '@angular/http';
import { Routes, RouterModule } from '@angular/router';

import { AppComponent } from './app.component';
import { HomeComponent } from './home/home.component';
import { UsersComponent } from './users/users.component';
import { ServersComponent } from './servers/servers.component';
import { UserComponent } from './users/user/user.component';
import { EditServerComponent } from './servers/edit-server/edit-server
import { ServerComponent } from './servers/server/server.component';
import { ServersService } from './servers/servers.service';
import { PageNotFoundComponent } from './page-not-found/page-not-found

const appRoutes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'users', component: UsersComponent, children: [
    { path: ':id/:name', component: UserComponent }
  ] },
  { path: 'servers', component: ServersComponent, children: [
    { path: ':id', component: ServerComponent },
    { path: ':id/edit', component: EditServerComponent }
  ] },
  { path: 'not-found', component: PageNotFoundComponent },
  { path: 'something', redirectTo: '/not-found' }
];

@NgModule({
```

- To redirect to a page where the routes are not handled in the application we use '**' to handle all the unhandled routes
  - We need to make sure this is the last route in the array of routes
  - If it is the beginning it will always be re-routed to not-found

- 

- Redirection path matching
  - By default, Angular matches paths by prefix. That means, that the following route will match both `/recipes` and just `/`
  - `{ path: '', redirectTo: '/somewhere-else' }`

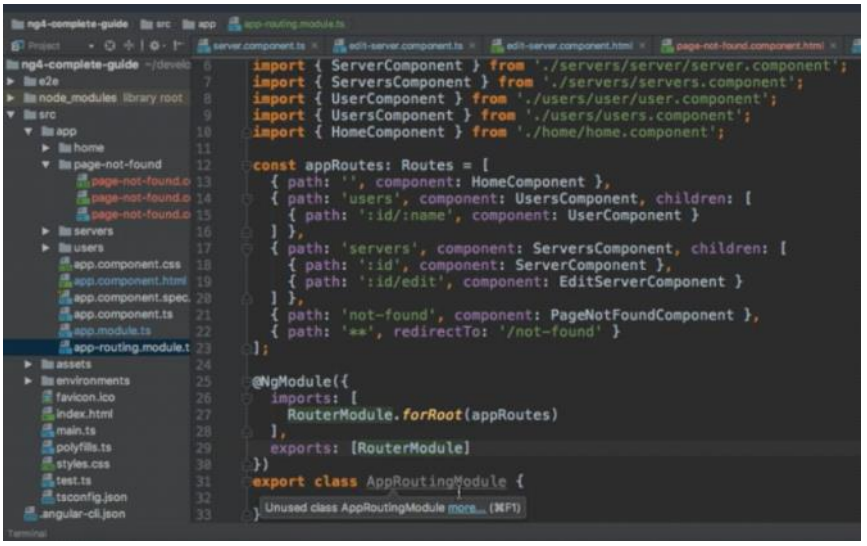
    Actually, Angular will give you an error here, because that's a common gotcha: This route will now **ALWAYS** redirect you! Why?

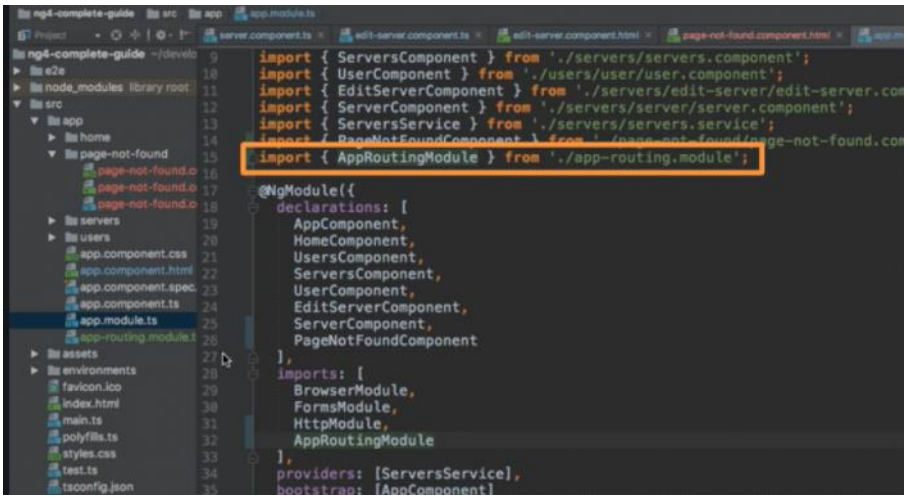
    Since the default matching strategy is `"prefix"`, Angular checks if the path you entered in the URL does **start with the path** specified in the route. Of course every path starts with `''` (Important: That's no whitespace, it's simply "nothing").
  - To fix this behavior, you need to change the matching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

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

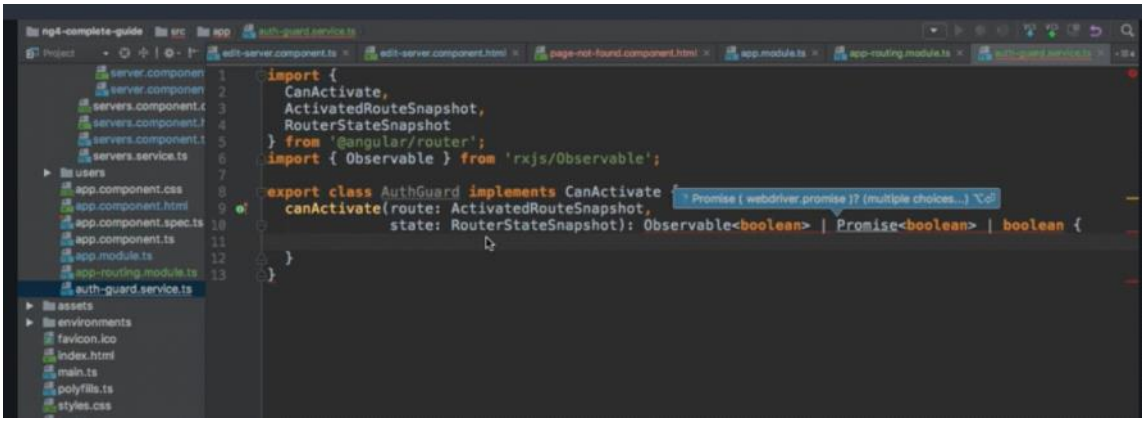
    Now, you only get redirected, if the full path is `''` (so only if you got NO other content in your path in this example).

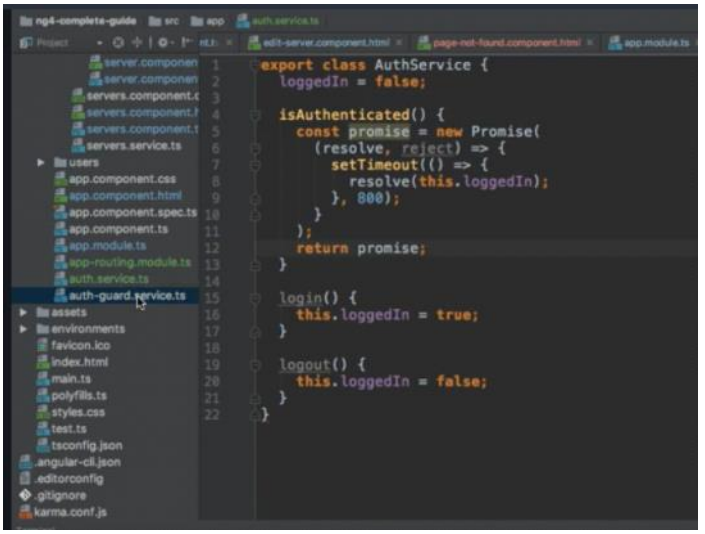
- Outsourcing the route config
  - We can move the routes config from app.module.ts to app-routing.module.ts
  - It needs
    - NgModule decorator
    - Routes config
      - imports: an array which mentions RouterModule.forRoot(appRoutes)
      - exports: an array which mentions RouterModule

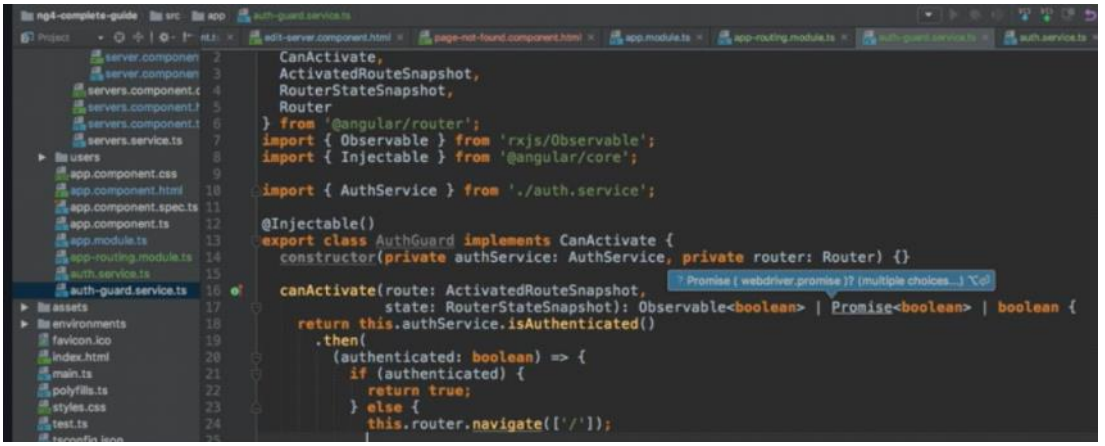
- 

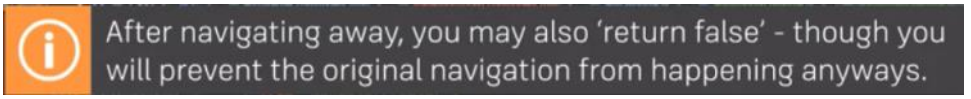
- In the app.module.ts we import and use it as in the below screenshot
- 

- Route Guards
  - Guards are used to protect some of the routes. For Ex. Authorization based on Persona where some routes needs to be protected
  - We need to use canActivate to protect the routes
  - We can implement an auth-guard.service.ts class which uses canActivate interface
  - canActivate interface is provided by '@angular/router'
  - We need @Injectable for the class to be injected
  - We can have a canActivate method in the class which will have two arguments
    - route of type 'ActivatedRouteSnapshot'
    - state of type 'RouterStateSnapshot'
    - Both these will be provided by '@angular/router'
    - This canActivate method can be returned as an observable or an promise or an boolean

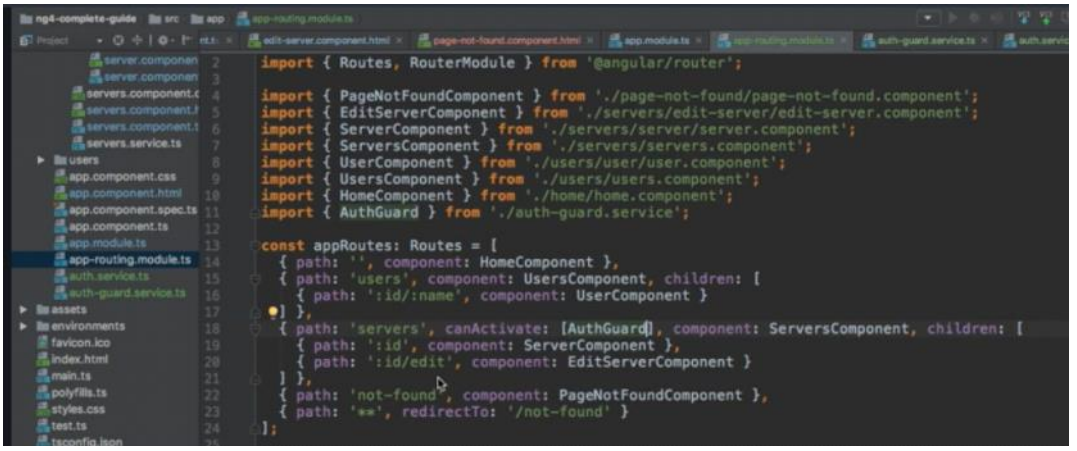
- 

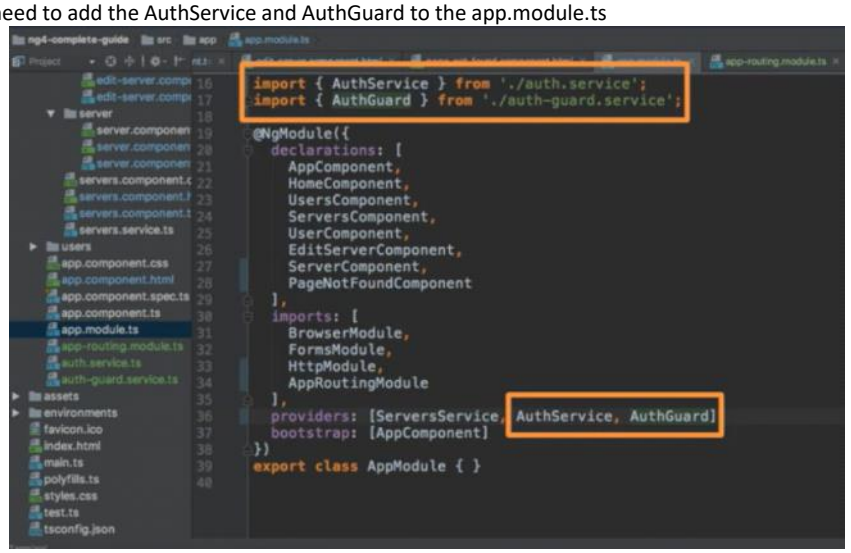
- Fake auth service to check canActivate
- 

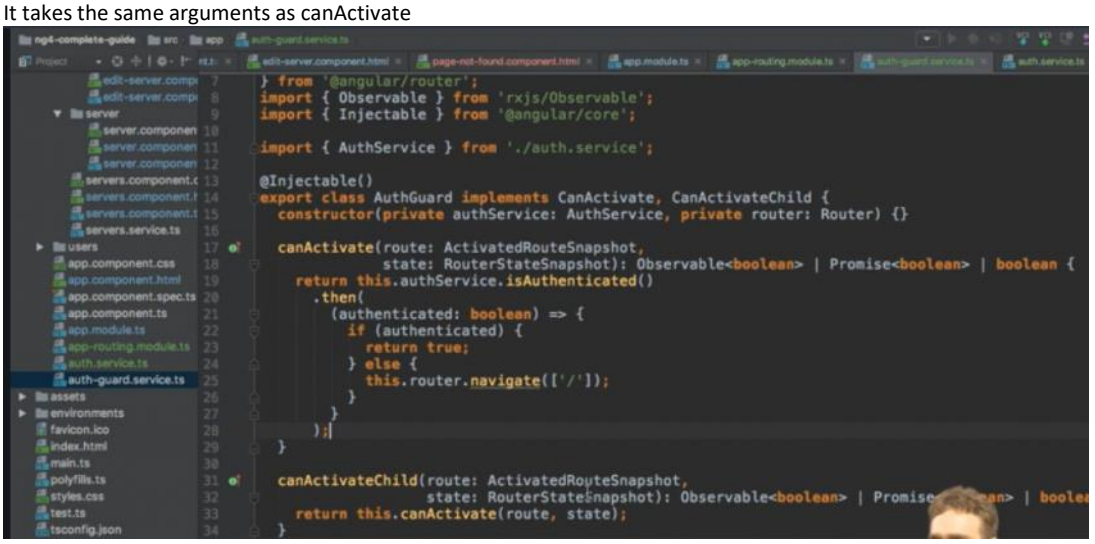
- 

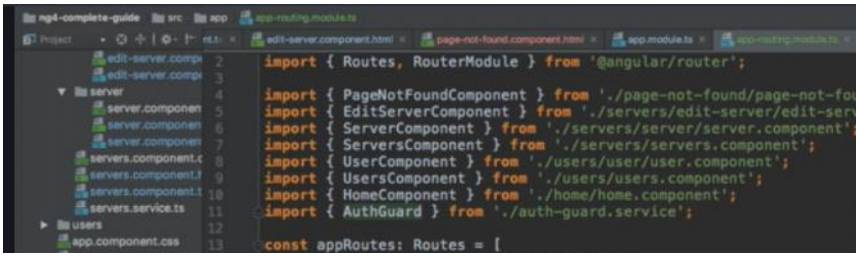
- 

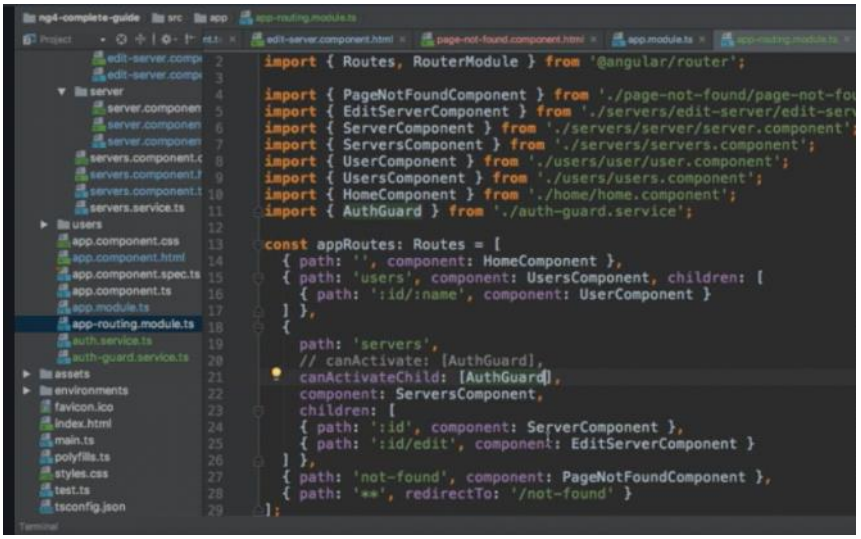
- Now in the app-routing.module.ts, we need to use this canActivate

- 

  - We need to add the AuthService and AuthGuard to the app.module.ts
- 

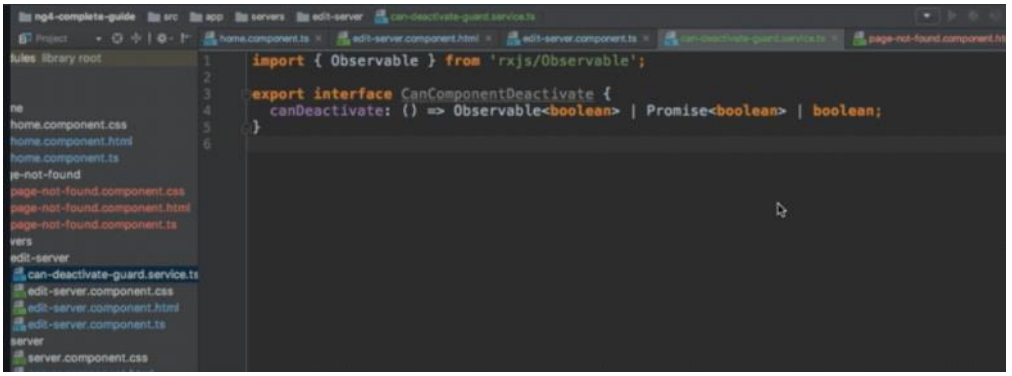
- Protecting Child (Nested) routes with canActivateChild
  - It takes the same arguments as canActivate
  - 

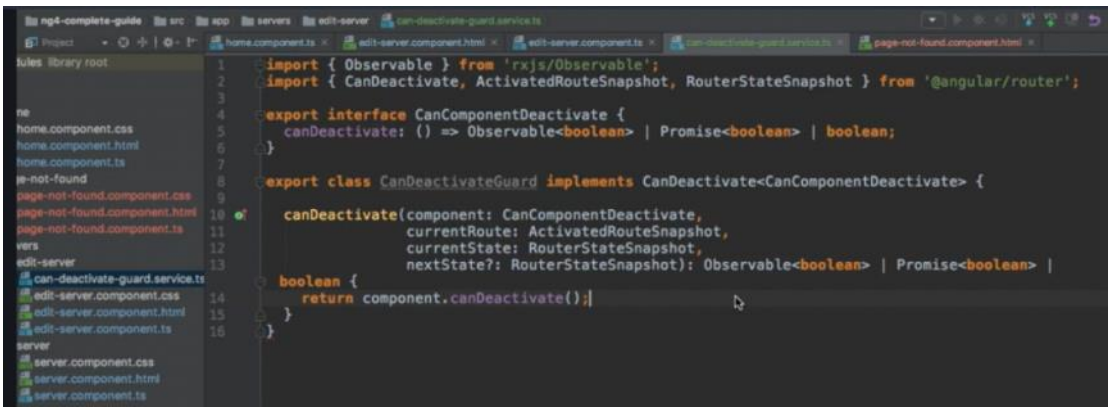


- 
```
import { Routes, RouterModule } from '@angular/router';

import { PageNotFoundComponent } from './page-not-found/page-not-fo
import { EditServerComponent } from './servers/edit-server/edit-ser
import { ServerComponent } from './servers/server/server.component'
import { ServersComponent } from './servers/servers.component';
import { UserComponent } from './users/user/user.component';
import { UsersComponent } from './users/users.component';
import { HomeComponent } from './home/home.component';
import { AuthGuard } from './auth-guard.service';

const appRoutes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'users', component: UsersComponent, children: [
    { path: ':id/:name', component: UserComponent }
  ] },
  {
    path: 'servers',
    // canActivate: [AuthGuard],
    canActivateChild: [AuthGuard],
    component: ServersComponent,
    children: [
    { path: ':id', component: ServerComponent },
    { path: ':id/edit', component: EditServerComponent }
  ] },
  { path: 'not-found', component: PageNotFoundComponent },
  { path: '**', redirectTo: '/not-found' }
];
```
- With the above we will be able to rote to Servers, but not the children routes of servers.

- Controlling navigation with canDeactivate
  - This to control whether we can leave a route or not
  - Ex. Accidental navigation without saving a form
  - In the component folder we can create a new file can-deactivate-guard.service.ts
    - We create an interface in this file
    - 
```
import { Observable } from 'rxjs/Observable';

export interface CanComponentDeactivate {
  canDeactivate: () => Observable<boolean> | Promise<boolean> | boolean;
}
```
    - 
```
import { Observable } from 'rxjs/Observable';
import { CanDeactivate, ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router';

export interface CanComponentDeactivate {
  canDeactivate: () => Observable<boolean> | Promise<boolean> | boolean;
}

export class CanDeactivateGuard implements CanDeactivate<CanComponentDeactivate> {

  canDeactivate(component: CanComponentDeactivate,
                currentRoute: ActivatedRouteSnapshot,
                currentState: RouterStateSnapshot,
                nextState?: RouterStateSnapshot): Observable<boolean> | Promise<boolean> |
 boolean {
    return component.canDeactivate();
  }
}
```

- ```
  import { PageNotFoundComponent } from './page-not-found/page-not-found.component';
  import { EditServerComponent } from './servers/edit-server/edit-server.component';
  import { ServerComponent } from './servers/server/server.component';
  import { ServersComponent } from './servers/servers.component';
  import { UserComponent } from './users/user/user.component';
  import { UsersComponent } from './users/users.component';
  import { HomeComponent } from './home/home.component';
  import { AuthGuard } from './auth-guard.service';
  import { CanDeactivateGuard } from './servers/edit-server/can-deactivate-guard.service';

  const appRoutes: Routes = [
    { path: '', component: HomeComponent },
    { path: 'users', component: UsersComponent, children: [
      { path: ':id/:name', component: UserComponent }
    ] },
    {
      path: 'servers',
      // canActivate: [AuthGuard],
      canActivateChild: [AuthGuard],
      component: ServersComponent,
      children: [
      { path: ':id', component: ServerComponent },
      { path: ':id/edit', component: EditServerComponent, canDeactivate: [CanDeactivateGuard] }
    ] },
  ```
- ```
  import { AuthService } from './auth.service';
  import { AuthGuard } from './auth-guard.service';
  import { CanDeactivateGuard } from './servers/edit-server/can-deactivate-gu

  @NgModule({
    declarations: [
      AppComponent,
      HomeComponent,
      UsersComponent,
      ServersComponent,
      UserComponent,
      EditServerComponent,
      ServerComponent,
      PageNotFoundComponent
    ],
    imports: [
      BrowserModule,
      FormsModule,
      HttpModule,
      AppRoutingModule
    ],
    providers: [ServersService, AuthService, AuthGuard, CanDeactivateGuard],
    bootstrap: [AppComponent]
  })
  export class AppModule { }
  ```
- ```
      }
      );
      this.route.fragment.subscribe();
      this.server = this.serversService.getServer(1);
      this.serverName = this.server.name;
      this.serverStatus = this.server.status;
    }

    onUpdateServer() {
      this.serversService.updateServer(this.server.id, {name: this.serverName, status: this
    .serverStatus});
      this.changesSaved = true;
      this.router.navigate(['../'], {relativeTo: this.route});
    }

    canDeactivate(): Observable<boolean> | Promise<boolean> | boolean {
      if (!this.allowEdit) {
        return true;
      }
      if ((this.serverName !== this.server.name || this.serverStatus !== this.server.status) &&
    !this.changesSaved) {
        return confirm('Do you want to discard the changes?');
      } else {
        return true;
      }
    }
  }
  ```

- Passing static data to a route

- ```
  import { Component, OnInit } from '@angular/core';
  import { ActivatedRoute, Data } from '@angular/router';

  @Component({
    selector: 'app-error-page',
    templateUrl: './error-page.component.html',
    styleUrls: ['./error-page.component.css']
  })
  export class ErrorPageComponent implements OnInit {
    errorMessage: string;

    constructor(private route: ActivatedRoute) { }

    ngOnInit() {
      this.errorMessage = this.route.snapshot.data['message'];
      this.route.data.subscribe(
  ```

- ```
  import { Component, OnInit } from '@angular/core';
  import { ActivatedRoute, Data } from '@angular/router';

  @Component({
    selector: 'app-error-page',
    templateUrl: './error-page.component.html',
    styleUrls: ['./error-page.component.css']
  })
  export class ErrorPageComponent implements OnInit {
    errorMessage: string;

    constructor(private route: ActivatedRoute) { }

    ngOnInit() {
      this.errorMessage = this.route.snapshot.data['message'];
      this.route.data.subscribe(
        (data: Data) => {
          this.errorMessage = data['message'];
        }
      );
    }

  }
  ```
- ```
  import { HomeComponent } from './home/home.component';
  import { AuthGuard } from './auth-guard.service';
  import { CanDeactivateGuard } from './servers/edit-server/can-deactivate-guard.service';
  import { ErrorPageComponent } from './error-page/error-page.component';

  const appRoutes: Routes = [
    { path: '', component: HomeComponent },
    { path: 'users', component: UsersComponent, children: [
      { path: ':id/:name', component: UserComponent }
    ] },
    {
      path: 'servers',
      // canActivate: [AuthGuard],
      canActivateChild: [AuthGuard],
      component: ServersComponent,
      children: [
      { path: ':id', component: ServerComponent },
      { path: ':id/edit', component: EditServerComponent, canDeactivate: [CanDeactivateGuard] }
    ] },
    // { path: 'not-found', component: PageNotFoundComponent },
    { path: 'not-found', component: ErrorPageComponent, data: {message: 'Page not found!'} },
    { path: '**', redirectTo: '/not-found' }
  ];
  ```

- Passing dynamic data to route
  - This is to load the route only after resolving the data i.e to load the data in advance before the route is loaded.
  - We can create a new file in the component folder called 'server-resolver.service.ts'
  - ```
    import { Resolve, ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router';
    import { Observable } from 'rxjs/Observable';
    import { Injectable } from '@angular/core';

    import { ServersService } from '../servers.service';

    interface Server {
      id: number;
      name: string;
      status: string;
    }

    @Injectable()
    export class ServerResolver implements Resolve<Server> {
      constructor(private serversService: ServersService) {}

      resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<Server> |
    Promise<Server> | Server {
        return this.serversService.getServer(+route.params['id']);
      }
    }
    ```
  - We need to add it in the app.module.ts
    ```

- ```
  import { AuthGuard } from './auth-guard.service';
  import { CanDeactivateGuard } from './servers/edit-server/can-deactivate-guard.service';
  import { ErrorPageComponent } from './error-page/error-page.component';
  import { ServerResolver } from './servers/server/server-resolver.service';

  @NgModule({
    declarations: [
      AppComponent,
      HomeComponent,
      UsersComponent,
      ServersComponent,
      UserComponent,
      EditServerComponent,
      ServerComponent,
      PageNotFoundComponent,
      ErrorPageComponent
    ],
    imports: [
      BrowserModule,
      FormsModule,
      HttpModule,
      AppRoutingModule
    ],
    providers: [ServersService, AuthService, AuthGuard, CanDeactivateGuard, ServerResolver],
    bootstrap: [AppComponent]
  })
  ```
- We also need to add it in the app-routing.module.ts
- ```
  import { AuthGuard } from './auth-guard.service';
  import { CanDeactivateGuard } from './servers/edit-server/can-deactivate-guard.service';
  import { ErrorPageComponent } from './error-page/error-page.component';
  import { ServerResolver } from './servers/server/server-resolver.service';

  const appRoutes: Routes = [
    { path: '', component: HomeComponent },
    { path: 'users', component: UsersComponent, children: [
      { path: ':id/:name', component: UserComponent }
    ] },
    {
      path: 'servers',
      // canActivate: [AuthGuard],
      canActivateChild: [AuthGuard],
      component: ServersComponent,
      children: [
      { path: ':id', component: ServerComponent, resolve: {server: ServerResolver} },
      { path: ':id/edit', component: EditServerComponent, canDeactivate: [CanDeactivateGuard] }
    ] },
    // { path: 'not-found', component: PageNotFoundComponent },
    { path: 'not-found', component: ErrorPageComponent, data: {message: 'Page not found!'} },
    { path: '**', redirectTo: '/not-found' }
  ];
  ```
  - The resolve is an object which has a key value pair in which the data is stored
  - When using in the component we get the data by using the subscribe method
    - We also need to import Data from '@angular/router'
    - ```
      import { Component, OnInit } from '@angular/core';
      import { ActivatedRoute, Params, Router, Data } from '@angular/router';

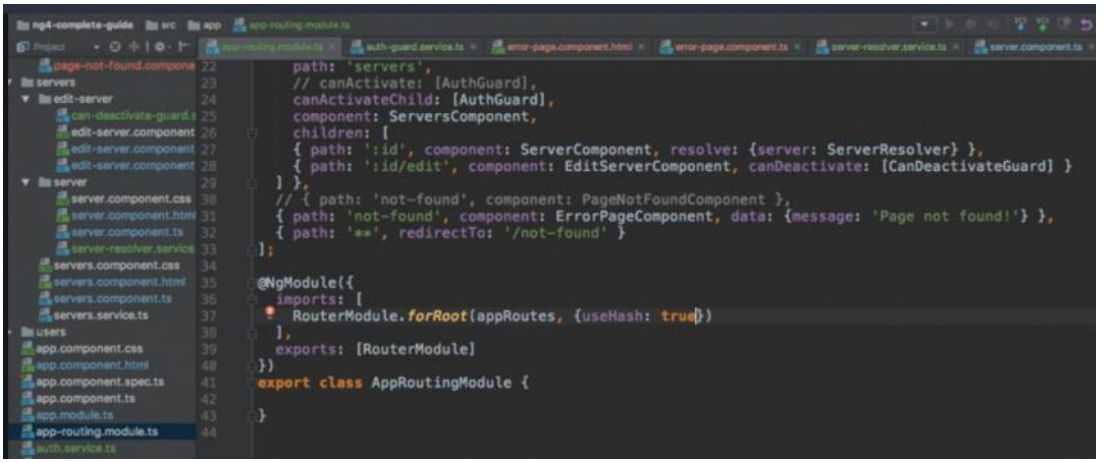
      import { ServersService } from '../servers.service';

      @Component({
        selector: 'app-server',
        templateUrl: './server.component.html',
        styleUrls: ['./server.component.css']
      })
      export class ServerComponent implements OnInit {
        server: {id: number, name: string, status: string};

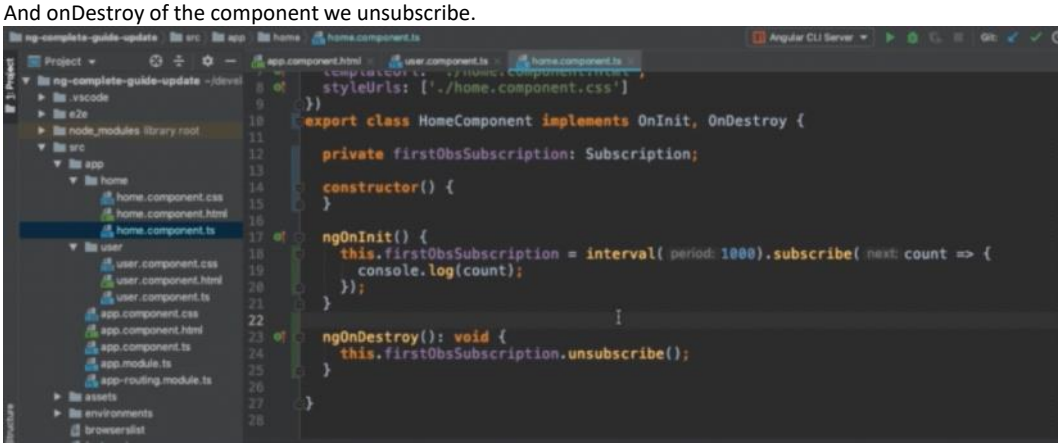
        constructor(private serversService: ServersService,
                    private route: ActivatedRoute,
                    private router: Router) {
        }

        ngOnInit() {
          this.route.data
            .subscribe(
              (data: Data) => {
                this.server = data['server'];
              }
            );
          // const id = +this.route.snapshot.params['id'];
          // this.server = this.serversService.getServer(id);
          // this.route.params
      ```

- Understanding Location strategies
  - Hash informs the webserver to take into consideration only the url before the hash tag by providing the below config
  - So that the part after the hash tag will be taken care by angular for routing
      ```

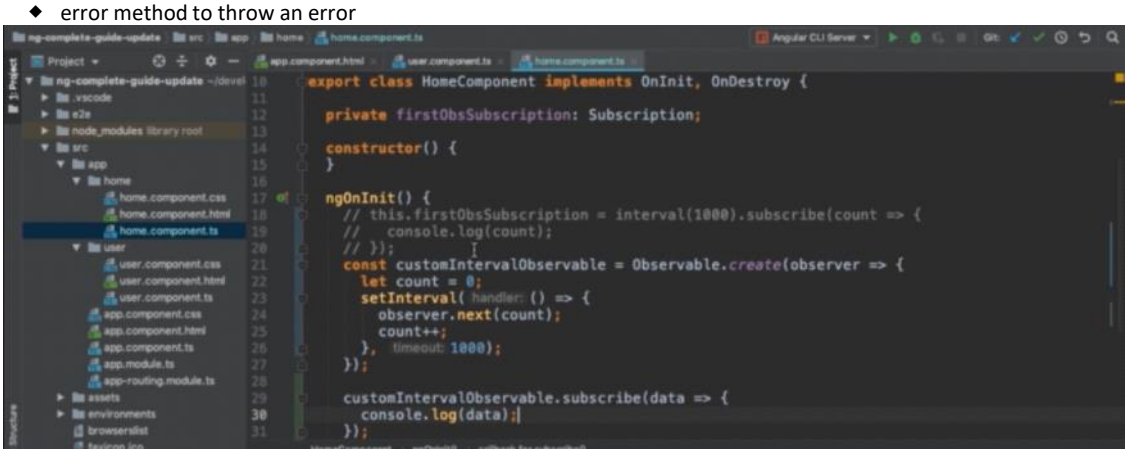
- 

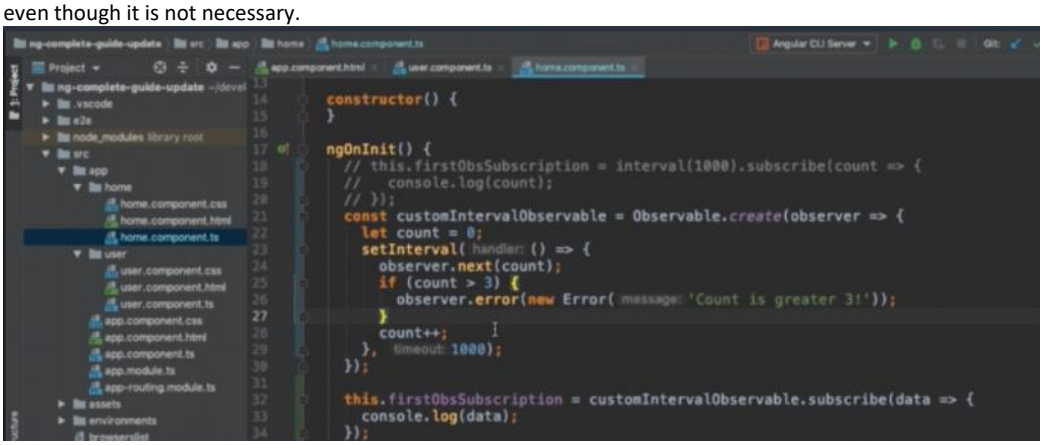
- Observables
  - Observables are constructs to which you subscribe to be informed about the changes in data because remember, observables are that stream of data and whenever a new data piece is emitted, our subscription will know about it.
  - Observables are the stream of data that gives us new values which need not be created but should be subscribed to use them.
  - Observables are used from package 'rxjs'
  - We can interval method from rxjs to create an observable which is similar to setInterval in js
    -

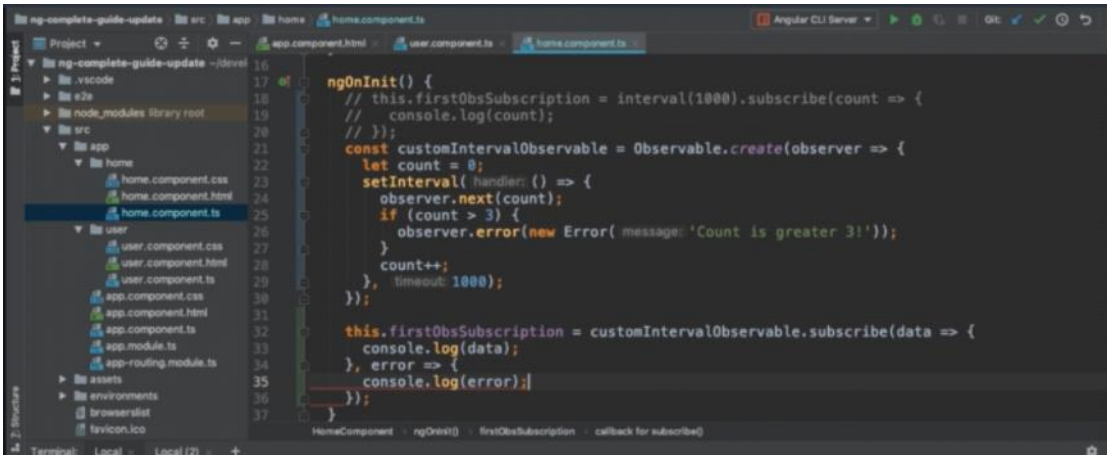
  - Observables do not stop emitting values until we explicitly unsubscribe
  - But there are some observables that emit the value only once and are done.(Ex. httpRequests)
  - To unsubscribe to an observable
    - We need to import Subscription from 'rxjs' library
    - Create a private variable of type Subscription and assign the observable to it
    - And onDestroy of the component we unsubscribe.
    - 

  - For observables provided by angular such as route params, http requests they are unsubscribed automatically by angular as soon as he component is destroyed.
  - Building custom observables
    - To create a custom observable we need to import Observable from 'rxjs' package
    - Observer has important methods. Below are some of them
      - next method to emit a new value.

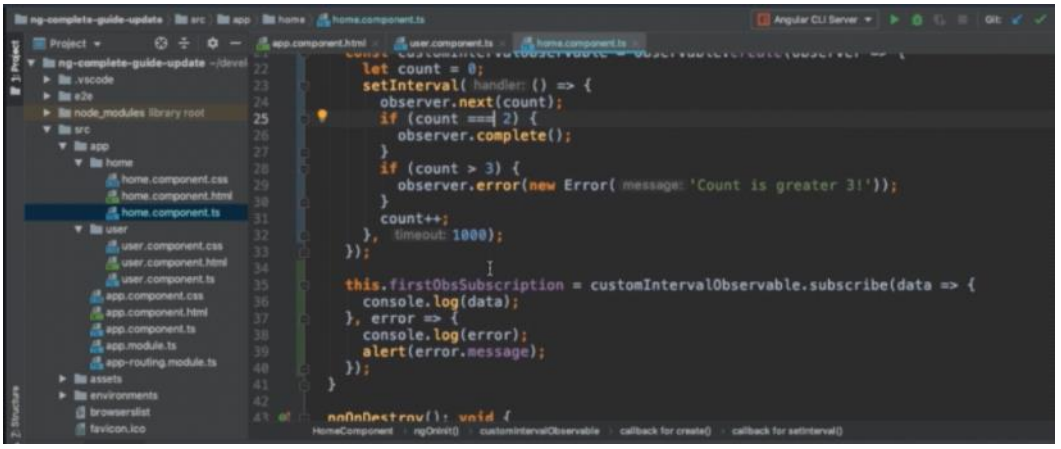
- complete method to let the observer know that we are done.
- error method to throw an error

- 

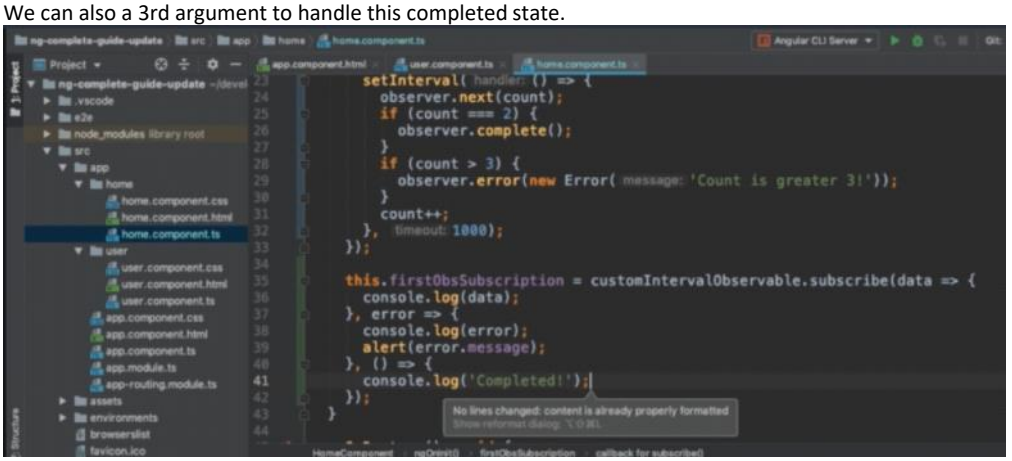
- Handling errors in observable to throw errors on need basis
  - Once the observable throws an error it will no longer be active but we can still unsubscribe it on destroy of the component even though it is not necessary.
  - 

  - We can also get the error method as an second argument while subscribing to the observable
  - Also we can show a toast or an alert based on the error thrown.
  - 

- Completing an observable
  - The http requests are completed once they receive the response.
  - Similarly we can manually complete an observable

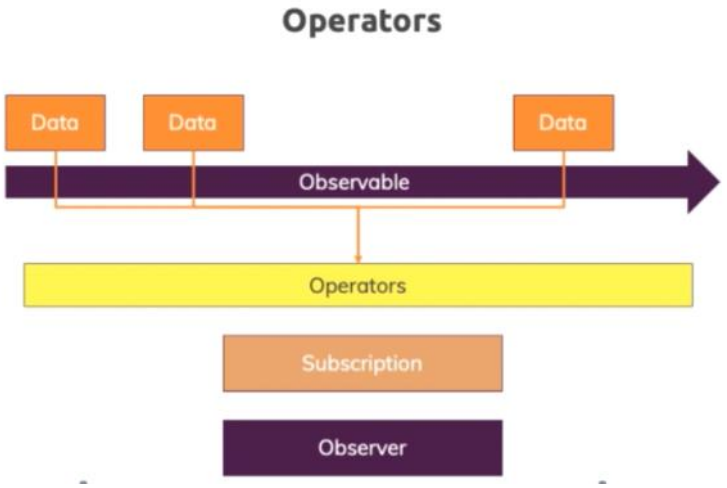
- 
```
let count = 0;
setInterval( handler: () => {
  observer.next(count);
  if (count ===  2) {
    observer.complete();
  }
  if (count > 3) {
    observer.error(new Error( message: 'Count is greater 3!'));
  }
  count++;
},  timeout: 1000);
});

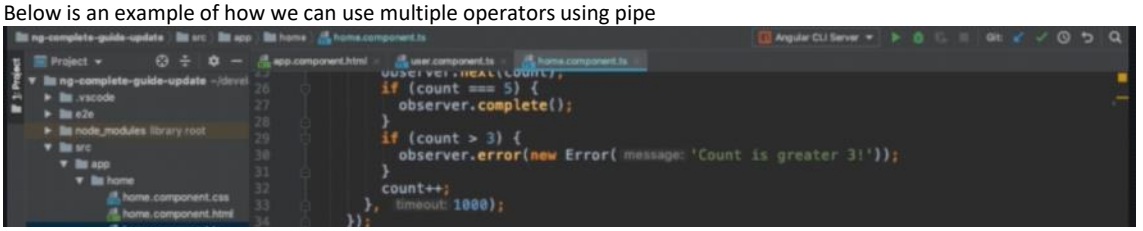
this.firstObsSubscription = customIntervalObservable.subscribe(data => {
  console.log(data);
}, error => {
  console.log(error);
  alert(error.message);
});
}

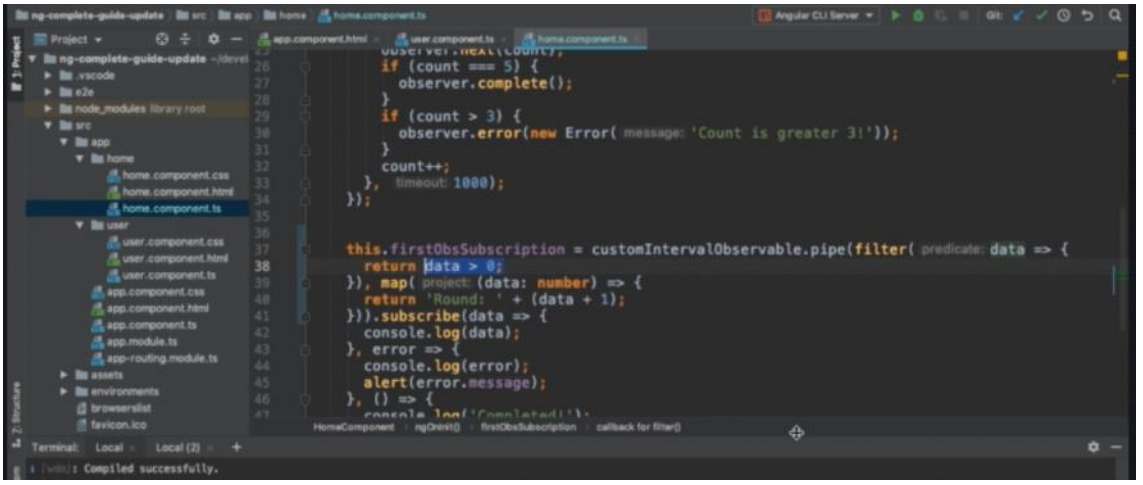
ngOnDestroy(): void {
```
  - By doing this the observable comes to an halt
  - We can also a 3rd argument to handle this completed state.
- 
```
setInterval( handler: () => {
  observer.next(count);
  if (count === 2) {
    observer.complete();
  }
  if (count > 3) {
    observer.error(new Error( message: 'Count is greater 3!'));
  }
  count++;
},  timeout: 1000);
});

this.firstObsSubscription = customIntervalObservable.subscribe(data => {
  console.log(data);
}, error => {
  console.log(error);
  alert(error.message);
}, () => {
  console.log('Completed!');
});
}
```
  - It is not mandatory that we unsubscribe to the completed observable but doing it will not do any harm.

□ Note: Error cancels the observable but does not complete it.

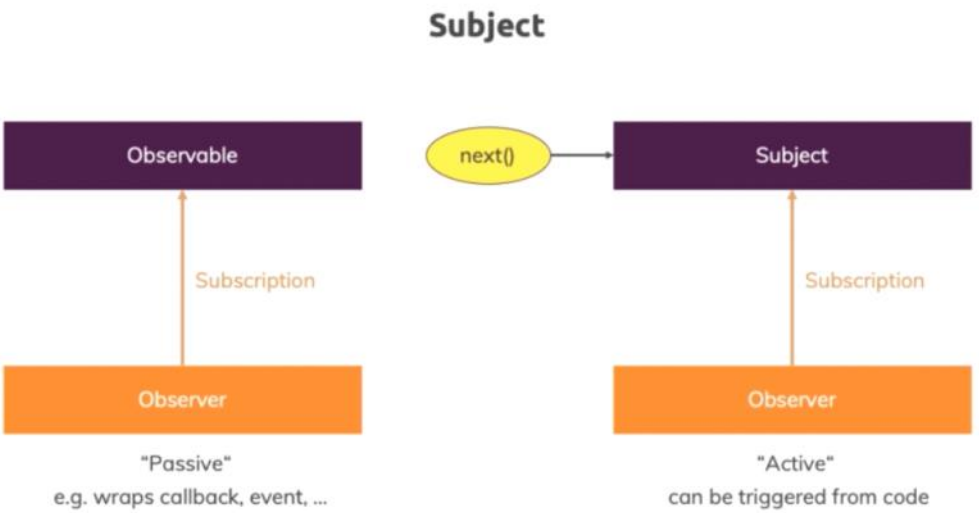
□ Understanding Operators
  - We can use built-in operators to transform the data of an observable before emitting it
  - 

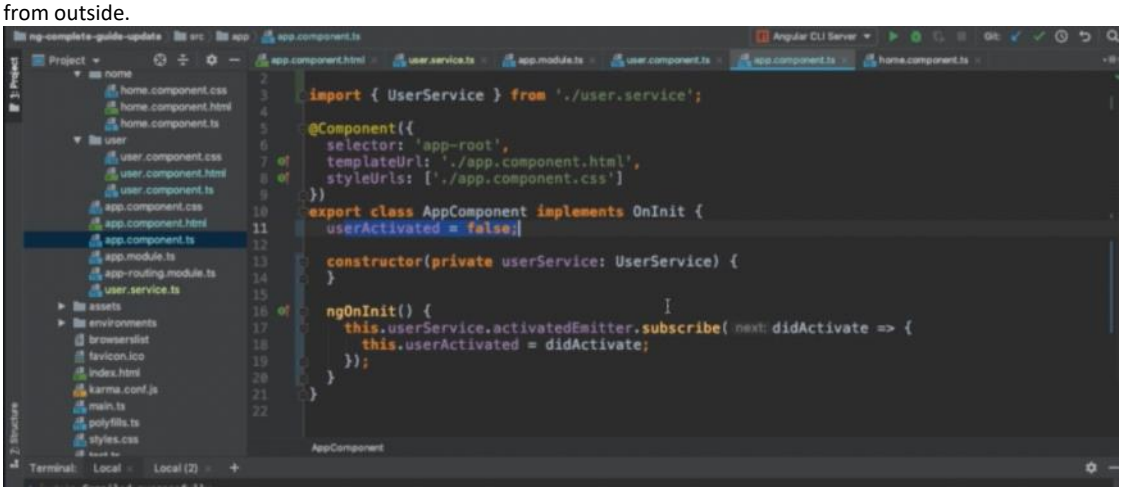
  - We can do this by using pipe
  - Pipe, map, filter operators is available from 'rxjs/operators' package
  - Pipe method allows us to add one or more operators
  - Below is an example of how we can use multiple operators using pipe
- 
```
if (count === 5) {
  observer.complete();
}
if (count > 3) {
  observer.error(new Error( message: 'Count is greater 3!'));
}
count++;
},  timeout: 1000);
```

- 

- Subjects
  - Subjects are recommended alternatives for events that are emitted from a service and are subscribed in the required components.
  - Subject is also a special kind of observable.
  - Subject can be imported from 'rxjs' library
  - Instead of EventEmitter we can use Subject in the service.
  -

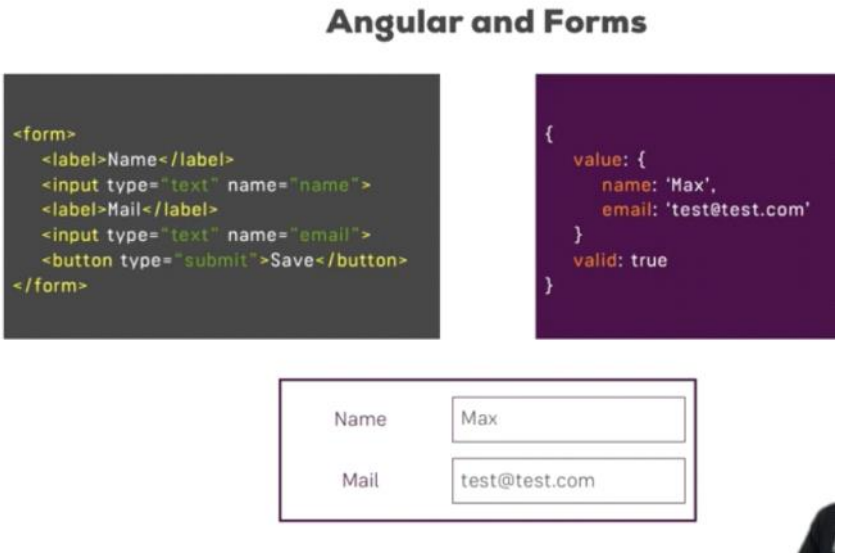
  - Instead of emit when we use EventEmitter we use next to emit the value when using Subject
  -

- 

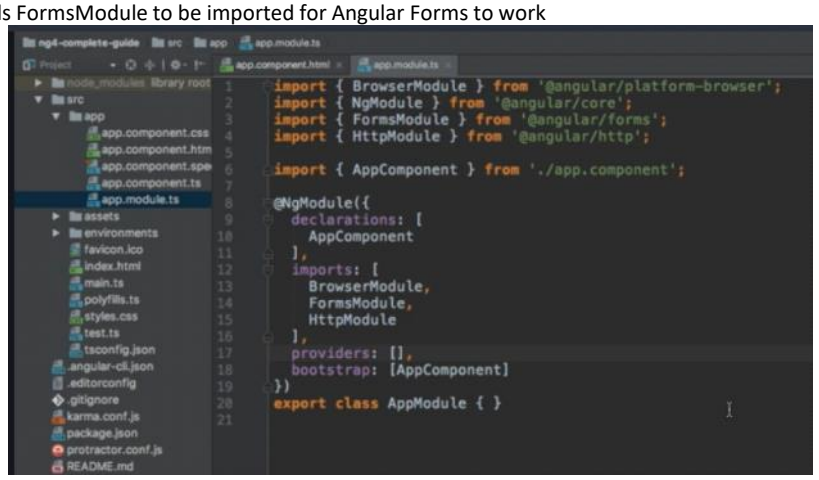
  - Subject has next method which can be called from outside the observer but in observables we can call it only from inside the observer.
  - Observables are passive where it wraps a callback or an event but Subject is Active as we can actively call next method from outside.
  - 

  - As we do for observables, we need to unsubscribe Subjects also on destroy of the component.
  - Subjects are used to communicate between components using services and is the recommended way instead of using @Output

- Angular Forms
  - Helps in development by returning an object representation of the form

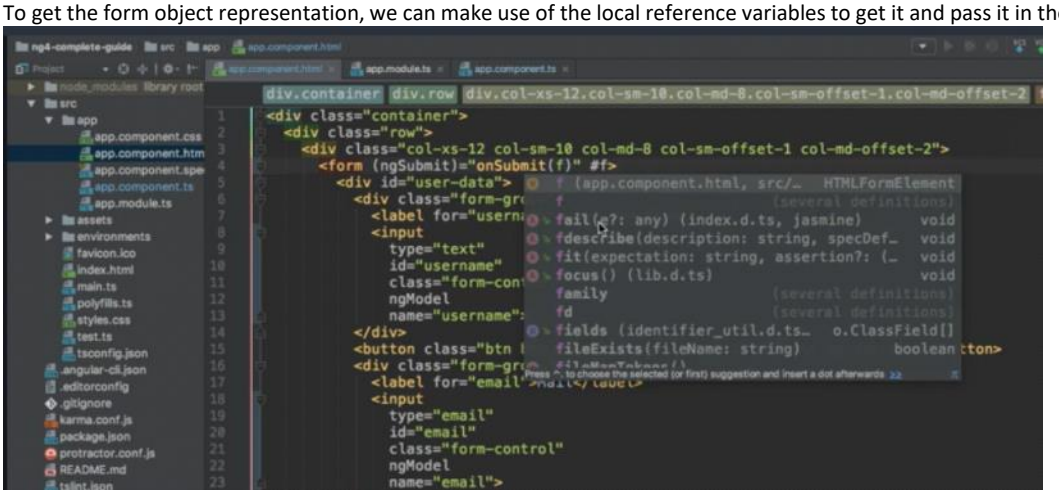
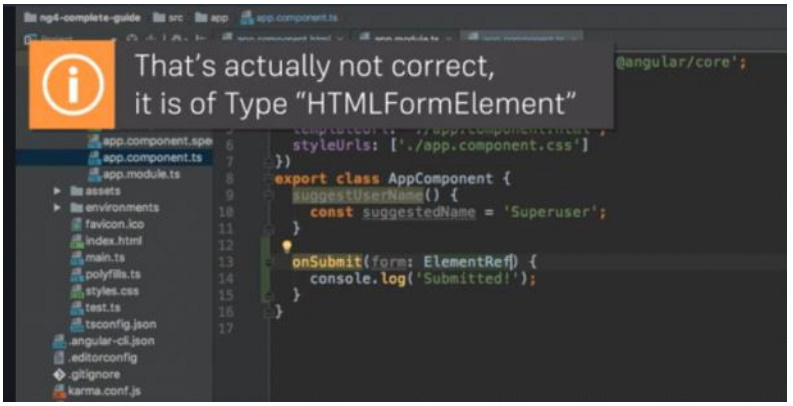
- 

-

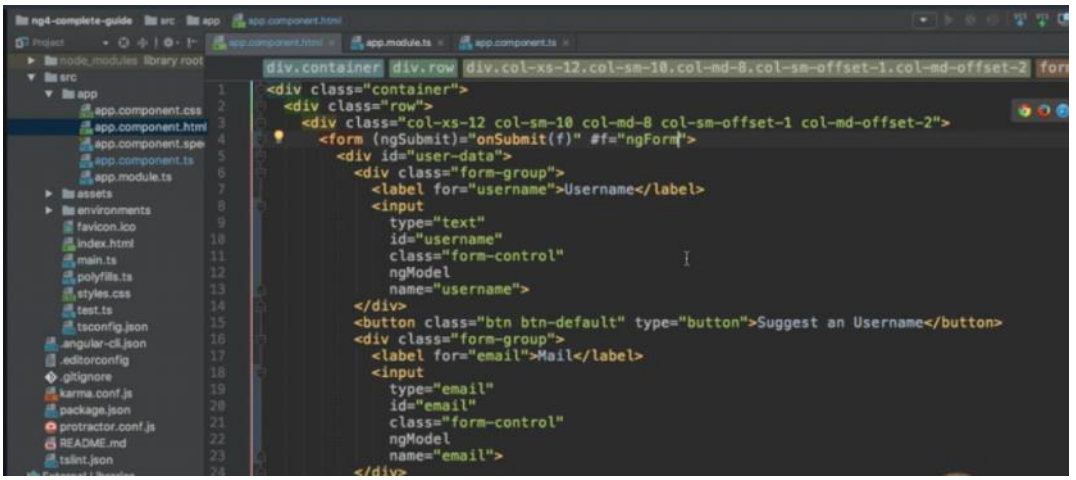
- Template Driven Approach
  - Needs FormsModule to be imported for Angular Forms to work
    - 
```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';

import { AppComponent } from './app.component';

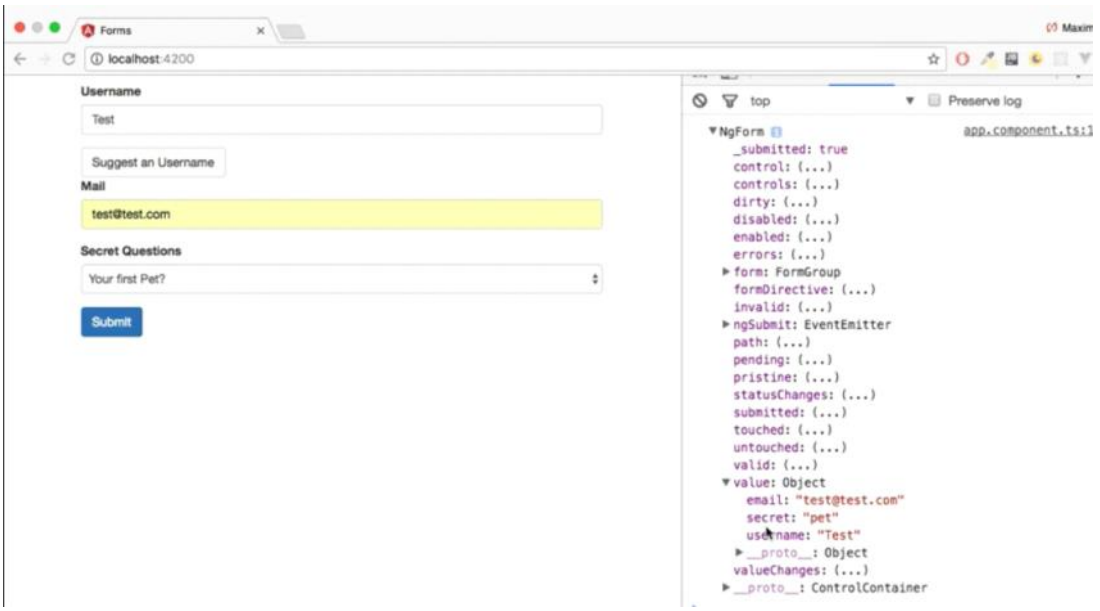
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```
  - We need to mention explicitly '**ngModel**' to the inputs we need in the object representation of the form
    - We also need to mention the **name** attribute of the input
    -
```
<form>
  <div id="user-data">
    <div class="form-group">
      <label for="username">Username</label>
      <input
        type="text"
        id="username"
        class="form-control"
        ngModel
        name="username">
    </div>
    <button class="btn btn-default" type="button">Suggest an Username</button>
```

- Angular provides an ngSubmit event as a form attribute to handle the form submission
  -
```
<div class="container">
  <div class="row">
    <div class="col-xs-12 col-sm-10 col-md-8 col-sm-offset-1 col-md-offset-2">
      <form (ngSubmit)="onSubmit()">
        <div id="user-data">
          <div class="form-group">
            <label for="username">Username</label>
            <input
              type="text"
              id="username"
              class="form-control"
              ngModel
              name="username">
          </div>
          <button class="btn btn-default" type="button">Suggest an Username</button>
          <div class="form-group">
            <label for="email">Mail</label>
            <input
              type="email"
              id="email"
              class="form-control"
              ngModel
              name="email">
```
- To get the form object representation, we can make use of the local reference variables to get it and pass it in the ngSubmit event
  - 
```
<div class="container">
  <div class="row">
    <div class="col-xs-12 col-sm-10 col-md-8 col-sm-offset-1 col-md-offset-2">
      <form (ngSubmit)="onSubmit(f)" #f>
        <div id="user-data">
          <div class="form-gr
            <label for="usern
            <input
              type="text"
              id="username"
              class="form-con
              ngModel
              name="username":
          </div>
          <button class="btn
          <div class="form-gr
            <label for="email"
            <input
              type="email"
              id="email"
              class="form-control"
              ngModel
              name="email">
```
  - 

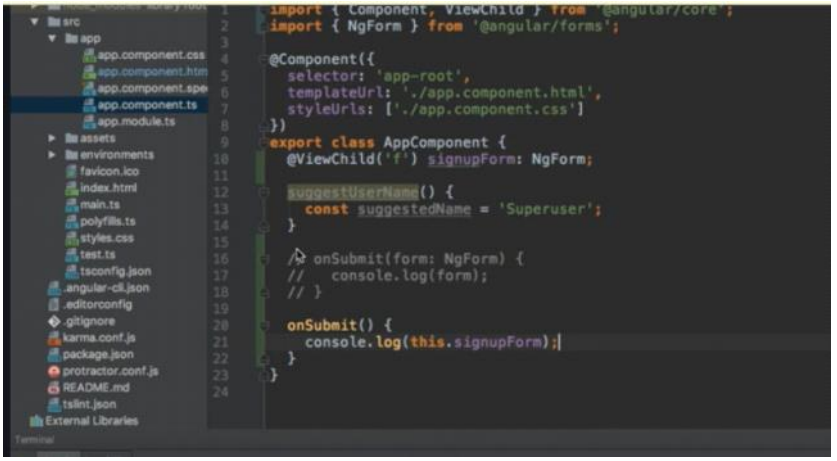
```
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  suggestUserName() {
    const suggestedName = 'Superuser';
  }

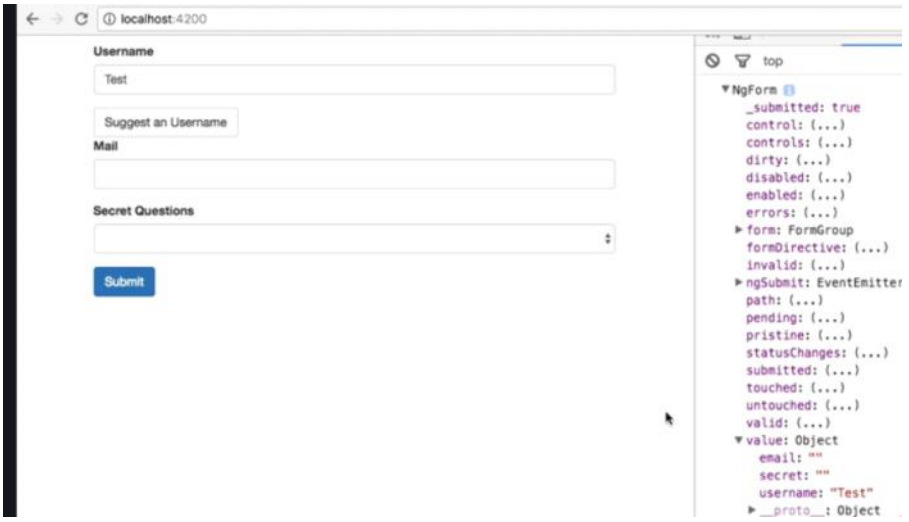
  onSubmit(form: ElementRef) {
    console.log('Submitted!');
  }
}
```
    - The form is of type HTMLFormElement
  - 
```
<div class="container">
  <div class="row">
    <div class="col-xs-12 col-sm-10 col-md-8 col-sm-offset-1 col-md-offset-2">
      <form (ngSubmit)="onSubmit(f)" #f="ngForm">
        <div id="user-data">
          <div class="form-group">
            <label for="username">Username</label>
            <input
              type="text"
              id="username"
              class="form-control"
              ngModel
              name="username">
          </div>
          <button class="btn btn-default" type="button">Suggest an Username</button>
          <div class="form-group">
            <label for="email">Mail</label>
            <input
              type="email"
              id="email"
              class="form-control"
              ngModel
              name="email">
```

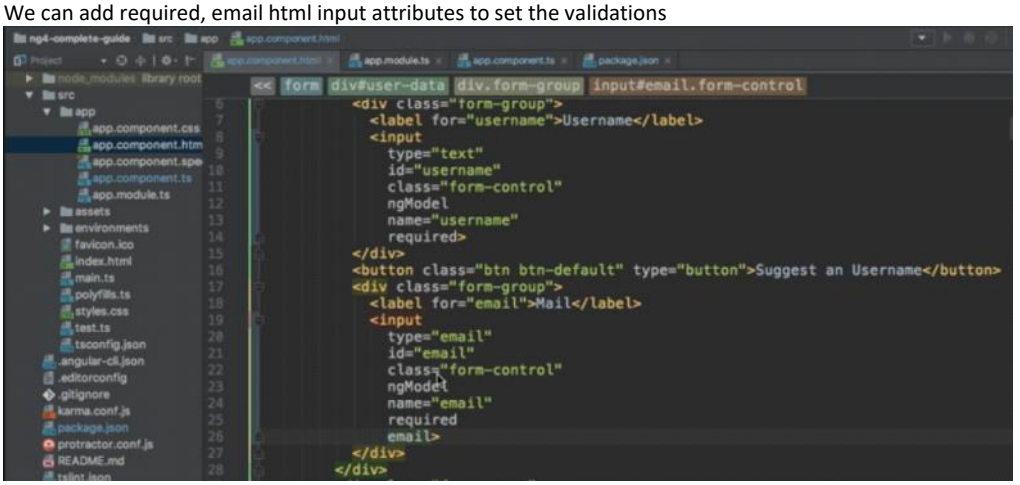
- We can set the local reference to 'ngForm' to tell Angular to give access to the form object created automatically
-

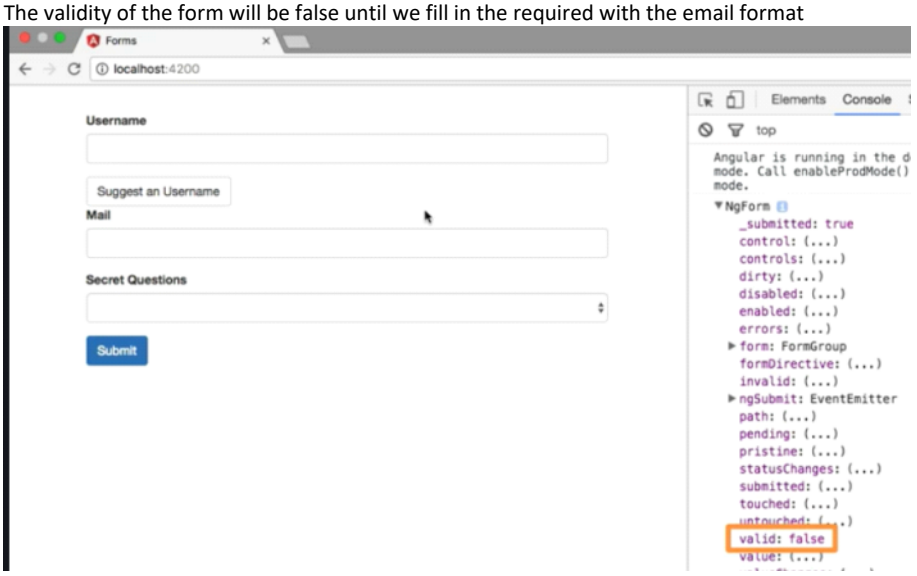
- 

- Forms has different states
  - dirty -> true when something changed in the form
  - valid -> true if it passes all the validation rules
  - disabled
  - enabled
  - errors
  - touched
  - untouched

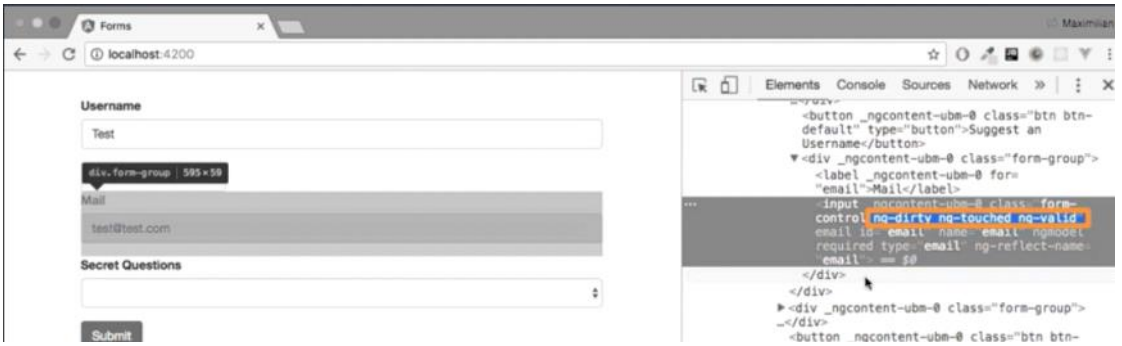
- Accessing form with @ViewChild using local reference (Alternative to get ngForm instead of passing local reference in ngSubmit event)
- 

- 

- Adding validations
  - We can add required, email html input attributes to set the validations
  - 

  - The validity of the form will be false until we fill in the required with the email format
  - 

  - Classes will also be added dynamically based on the actions done in the input which can be used for styling fields on validation errors (ex. ng-invalid will be added if the email field does not follow the email format)

- 

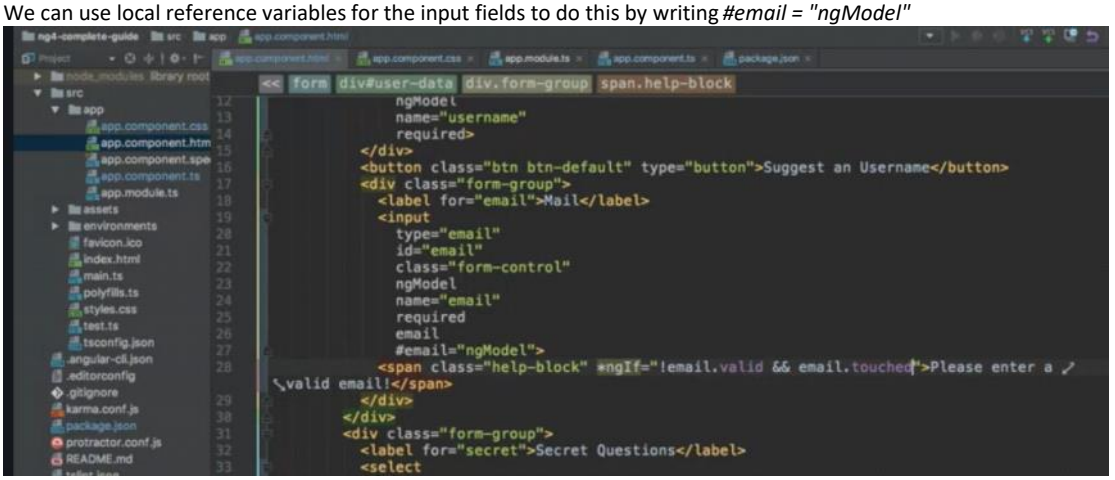
□ Using local reference variable to disable the submit button when the form is invalid

-

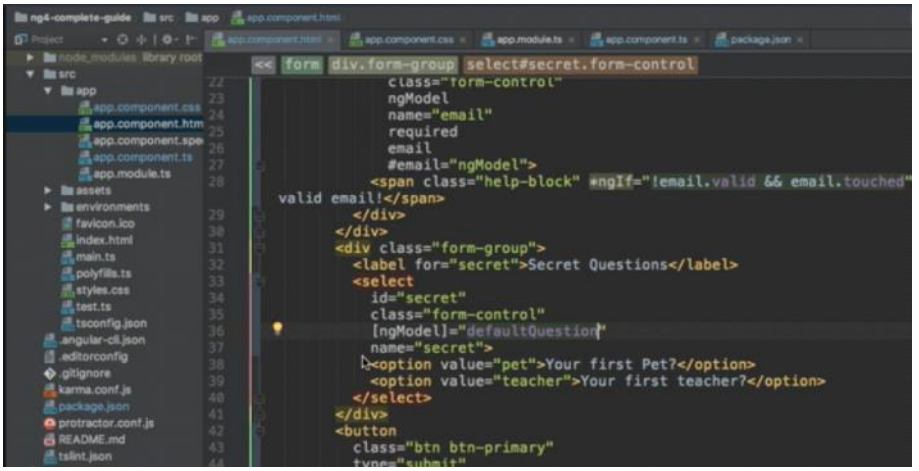
□ To style an required input field when touched and not entered any value

-

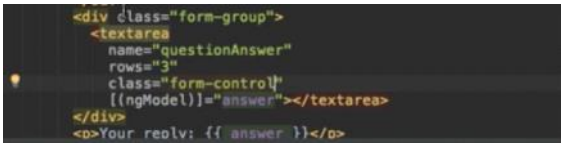
□ Outputting validation error messages

- We can use local reference variables for the input fields to do this by writing *#email = "ngModel"*
- 

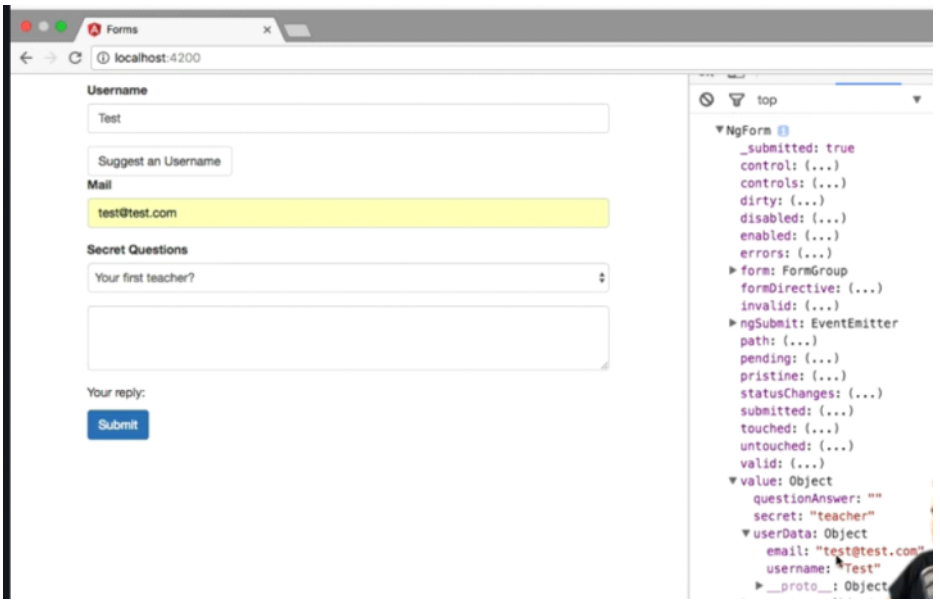
□ We can set the 1-way binding for the ngModel(Property binding) to set the default values by writing *[ngModel]="defaultValue"*

- 

- We can also set 2-way binding for the ngModel in case we want the input element to react on the changes in the input element immediately without clicking on the submit button by writing *[(ngModel)]="answer"*

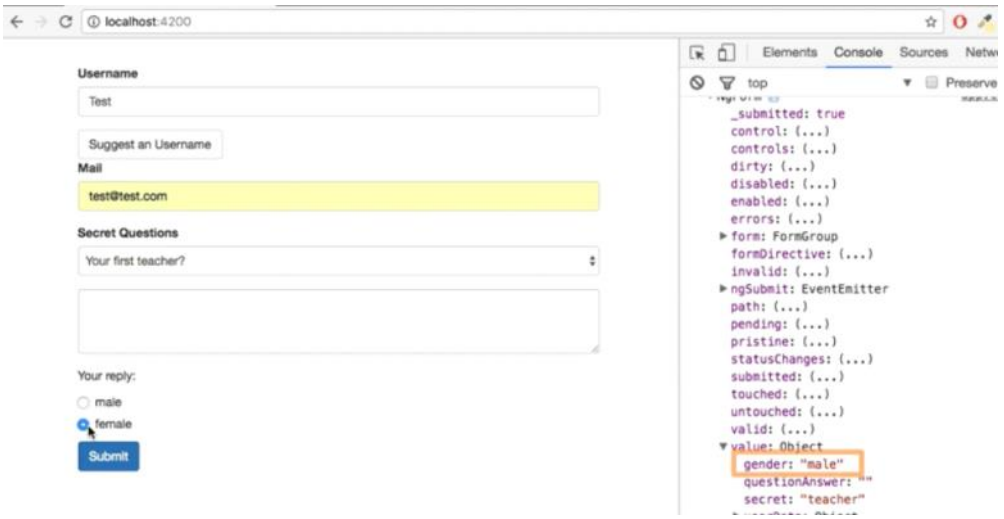
- 

- Grouping form controls
  - We can do this by writing *ngModelGroup="userData"* to the parent container consisting the input elements.
  - userData will be the key in which the data will be stored.
  -

  - 

- Similarly we can set a local reference variable to the container by writing *#userData="ngModelGroup"* to message an output if the whole group is not valid
  -
```
<div class="col-xs-12 col-sm-10 col-md-8 col-sm-offset-1 col-md-offset-2">
  <form (ngSubmit)="onSubmit()" #f="ngForm">
    <div
      id="user-data"
      ngModelGroup="userData"
      #userData="ngModelGroup">
      <div class="form-group">
        <label for="username">Username</label>
        <input
          type="text"
          id="username"
          class="form-control"
          ngModel
          name="username"
          required>
      </div>
      <button class="btn btn-default" type="button">Suggest an Username</button>
      <div class="form-group">
```

- Handling radio buttons
  -
```
</div>
<div class="form-group">
  <textarea
    name="questionAnswer"
    rows="3"
    class="form-control"
    [(ngModel)]="answer"></textarea>
</div>
<p>Your reply: {{ answer }}</p>
<div class="radio" *ngFor="let gender of genders">
  <label>
    <input
      type="radio"
      name="gender"
      ngModel
      [value]="gender">
  </label>
```

  - 

  - We can use 1-way binding to set the default value of the radio button

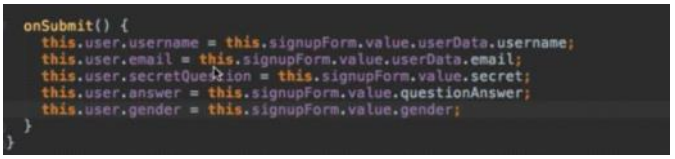
- Setting and Patching Form Values
  - We can set values of the form as in the below mentioned way by duplicating the form object structure
    - We can set values in the form by default to all the fields in the form
    -
```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  @ViewChild('f') signupForm: NgForm;
  defaultQuestion = 'teacher';
  answer = '';
  genders = ['male', 'female'];

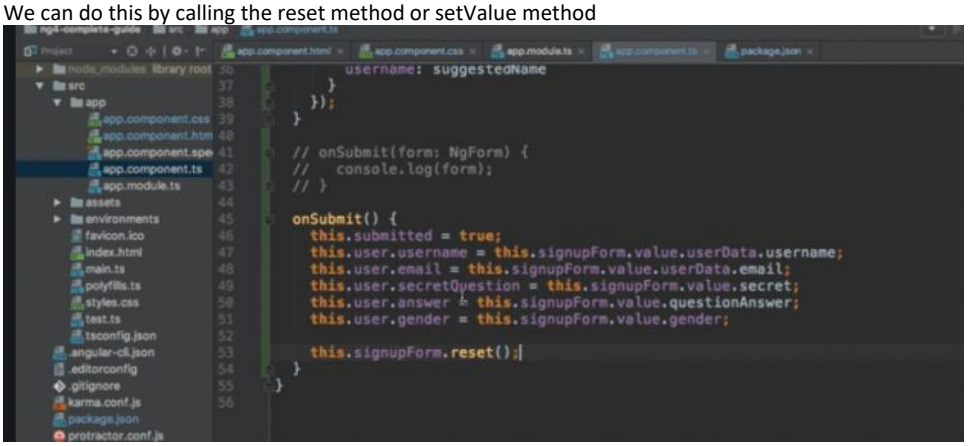
  suggestUserName() {
    const suggestedName = 'Superuser';
    this.signupForm.setValue({
      userData: {
        username: suggestedName,
        email: ''
      },
      secret: 'pet',
      questionAnswer: '',
      gender: 'male'
    });
  }

  // onSubmit(form: NgForm) {
```

- We can also patch the values of the form which is used set values only to certain fields in the form
  -
```
export class AppComponent {
  @ViewChild('f') signupForm: NgForm;
  defaultQuestion = 'teacher';
  answer = '';
  genders = ['male', 'female'];

  suggestUserName() {
    const suggestedName = 'Superuser';
    // this.signupForm.setValue({
    //   userData: {
    //     username: suggestedName,
    //     email: ''
    //   },
    //   secret: 'pet',
    //   questionAnswer: '',
    //   gender: 'male'
    // });
    this.signupForm.form.patchValue({
      userData: {
        username: sug
      }
    });
  }
```

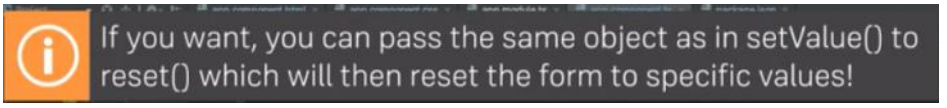
- To get the values in the ts file
  - 
```
onSubmit() {
  this.user.username = this.signupForm.value.userData.username;
  this.user.email = this.signupForm.value.userData.email;
  this.user.secretQuestion = this.signupForm.value.secret;
  this.user.answer = this.signupForm.value.questionAnswer;
  this.user.gender = this.signupForm.value.gender;
}
}
```

- Resetting Forms
  - We can do this by calling the reset method or setValue method
  - 
```
    username: suggestedName
  }
});
}

// onSubmit(form: NgForm) {
//   console.log(form);
// }

onSubmit() {
  this.submitted = true;
  this.user.username = this.signupForm.value.userData.username;
  this.user.email = this.signupForm.value.userData.email;
  this.user.secretQuestion = this.signupForm.value.secret;
  this.user.answer = this.signupForm.value.questionAnswer;
  this.user.gender = this.signupForm.value.gender;

  this.signupForm.reset();
}
}
```

  - 

- Pipes
  - To transform data to the required format in the html template.

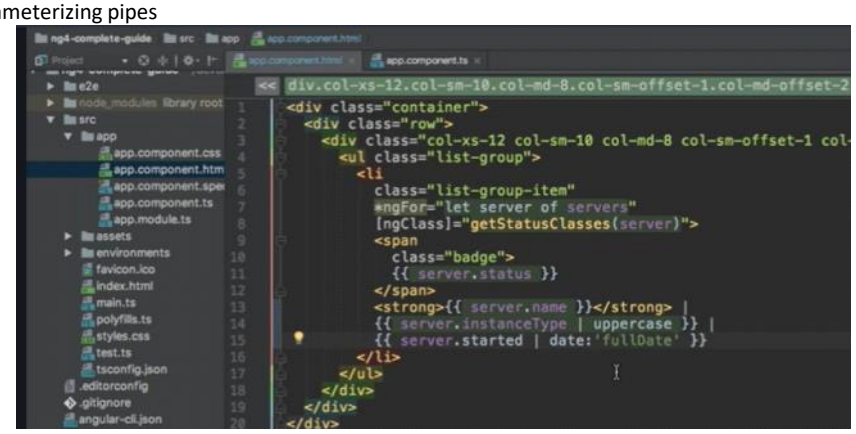
What are Pipes?

- ```
  username = 'Max'
  ```

  ```
  {{ username }}
  ```

  ```
  Max
  ```

  ```
  {{ username | uppercase }}
  ```

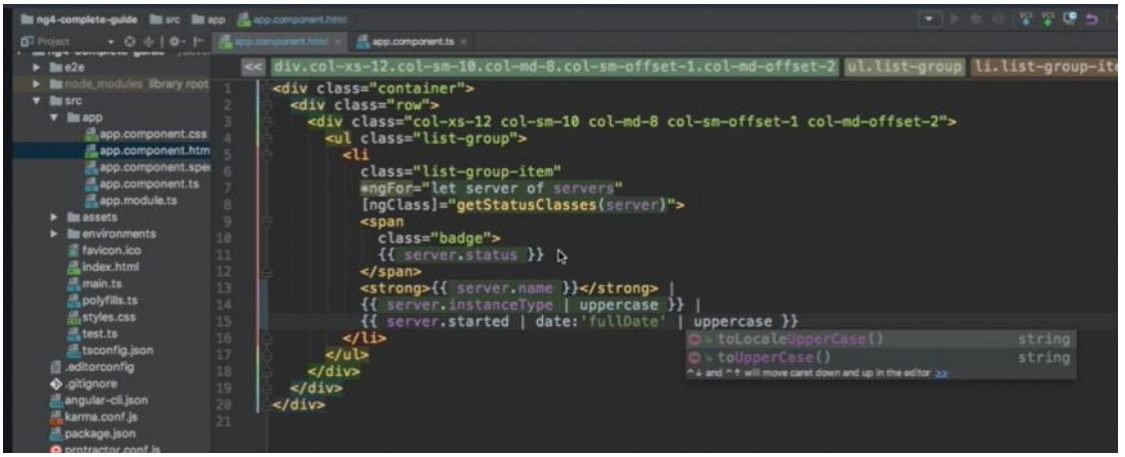
- Parameterizing pipes
  - 

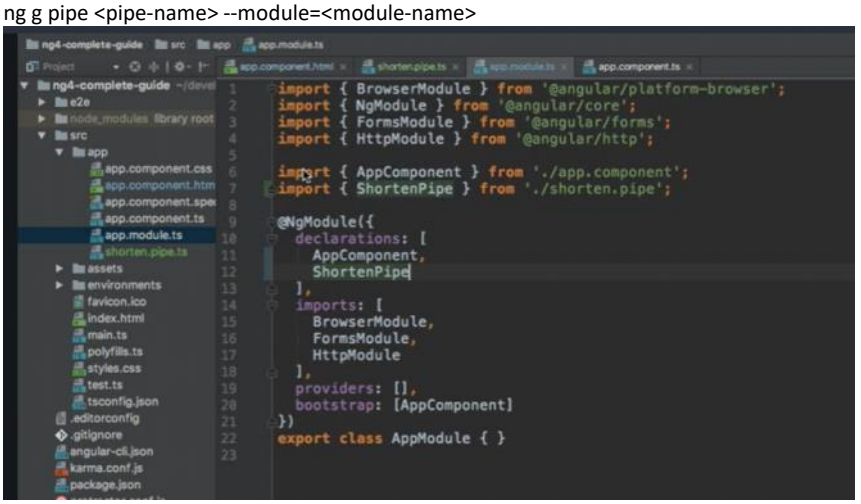
  - We can send parameters to the pipe using ':'

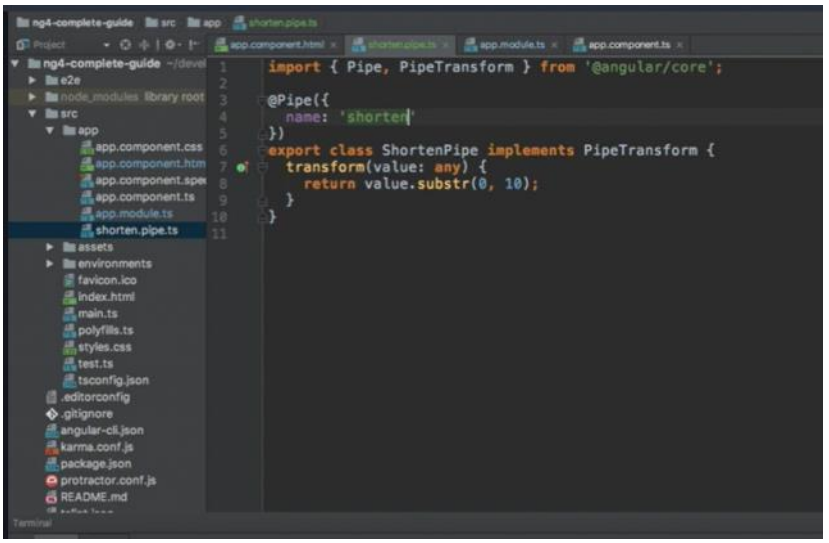
- Built-in pipes in Angular
  -

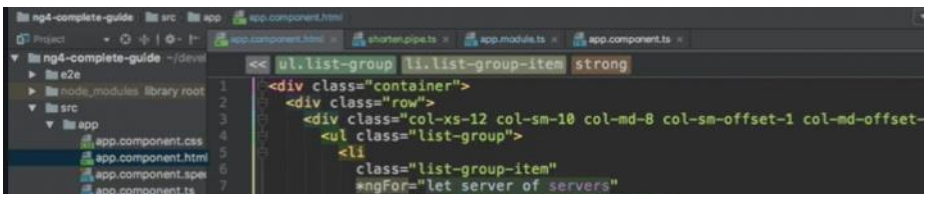
  -

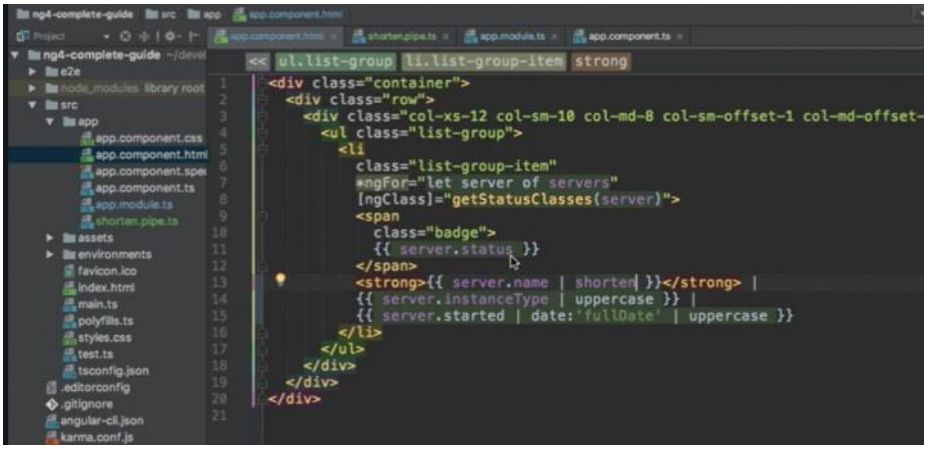
- Chaining multiple pipes
  ```

- 

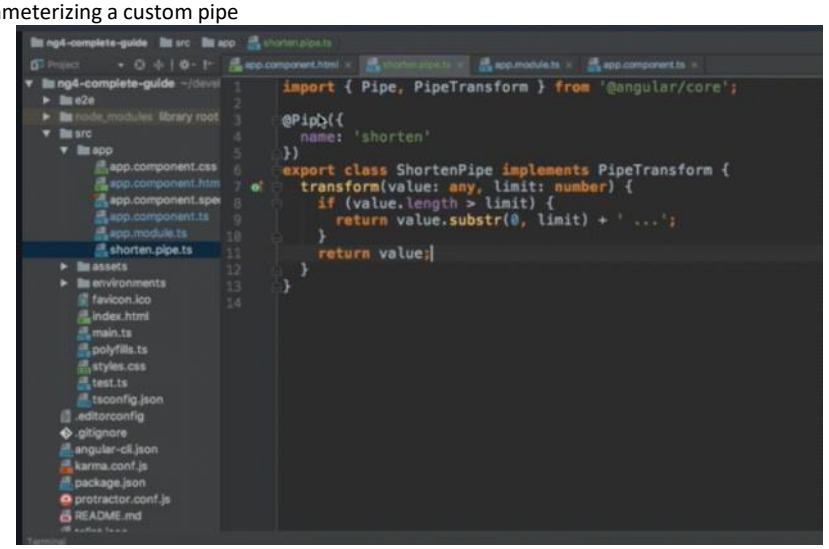
- Creating custom pipes
  - We need to import PipeTransform from '@angular/core' package
  - We need to implement the class with PipeTransform
  - Use the transform method to return the transformed value
  - Add in app.module.ts
  - We can create pipe through cli using the following command
    - ng g pipe <pipe-name> --module=<module-name>
    - 

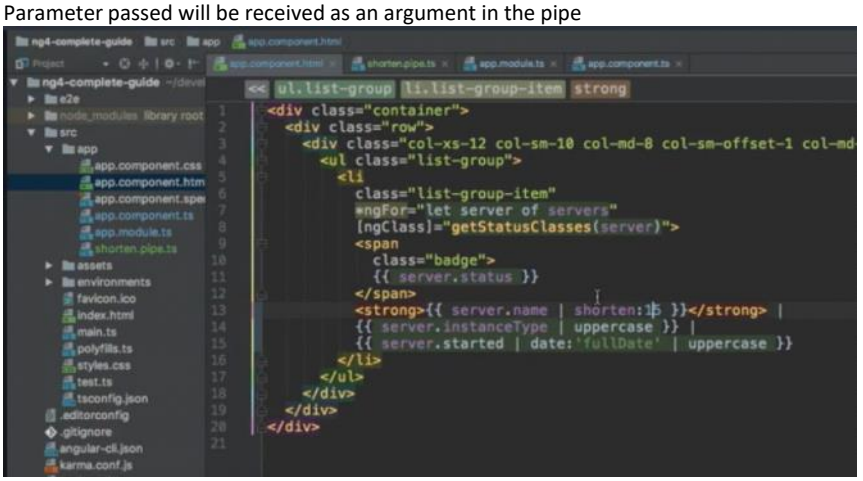
    - 

    - 

- 

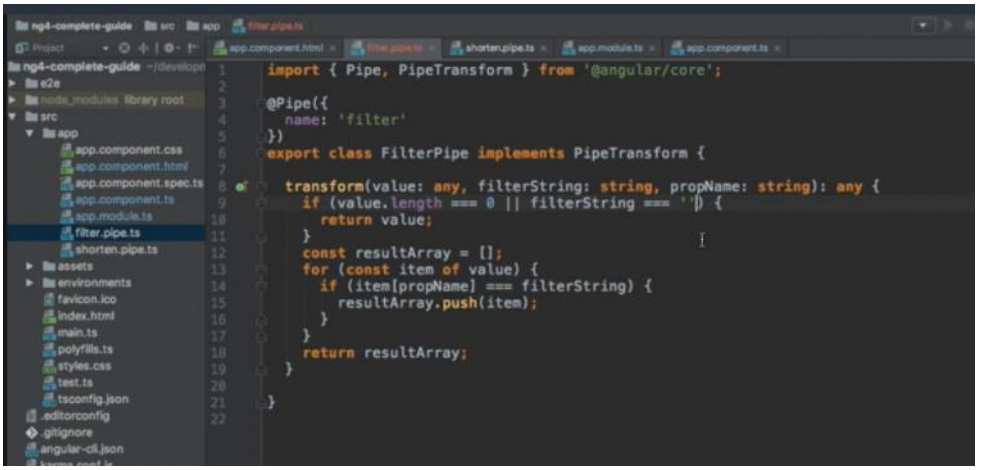
□ Parameterizing a custom pipe

- 

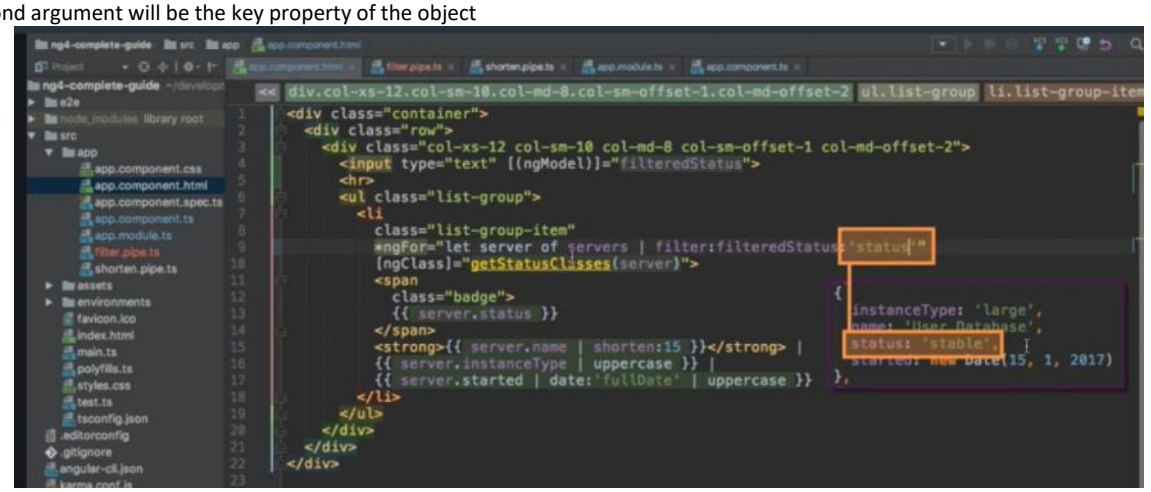
- Parameter passed will be received as an argument in the pipe
- 

- We can chain multiple pipes as well and those will be received as arguments in the pipe.

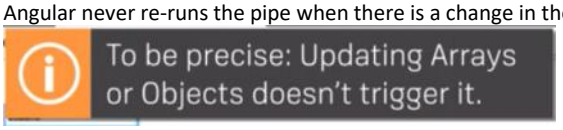
□ Filter Array Example

- 

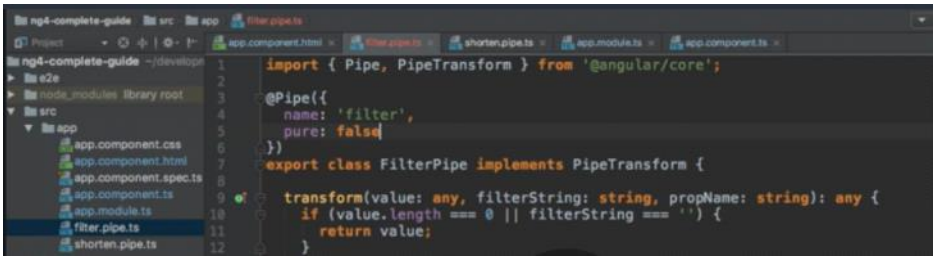
- First argument will be input from a text field as shown in below screenshot
-

- Second argument will be the key property of the object
- 

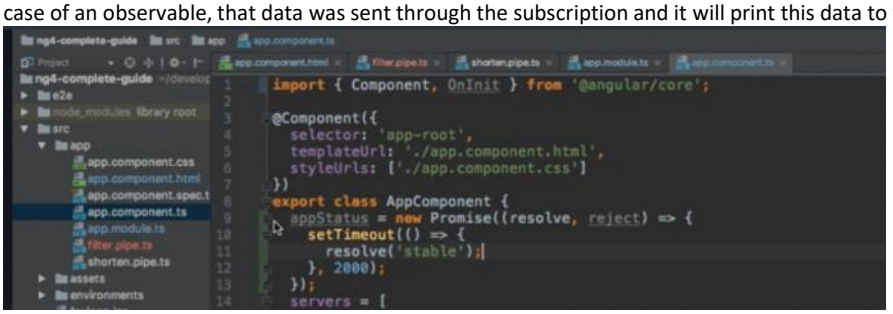
□ Pure and Impure Pipes

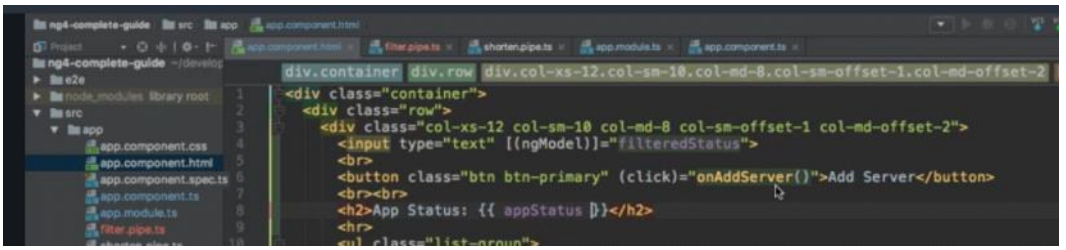
- Angular never re-runs the pipe when there is a change in the data as it would cause performance issues
- 

- But we can still update the pipe to listen to the data changes by adding another meta data as pure: false to the Pipe decorator

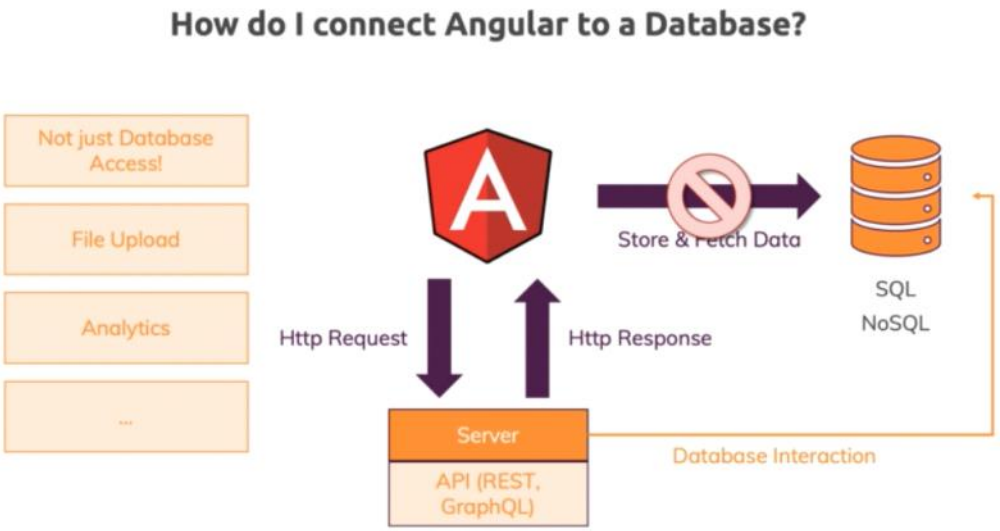
- 

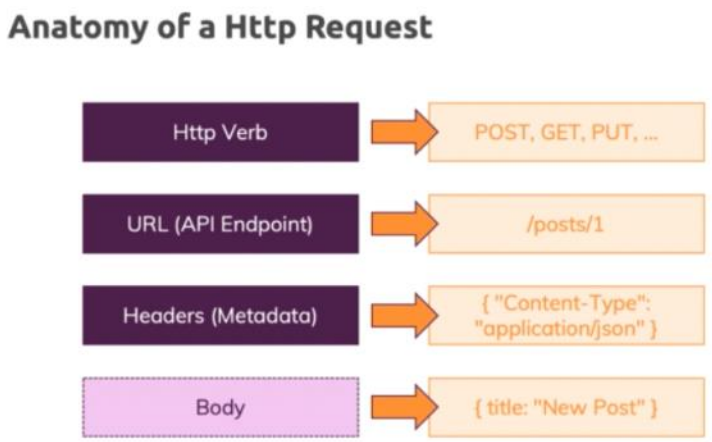
- Note: Impure pipes may lead to performance issues

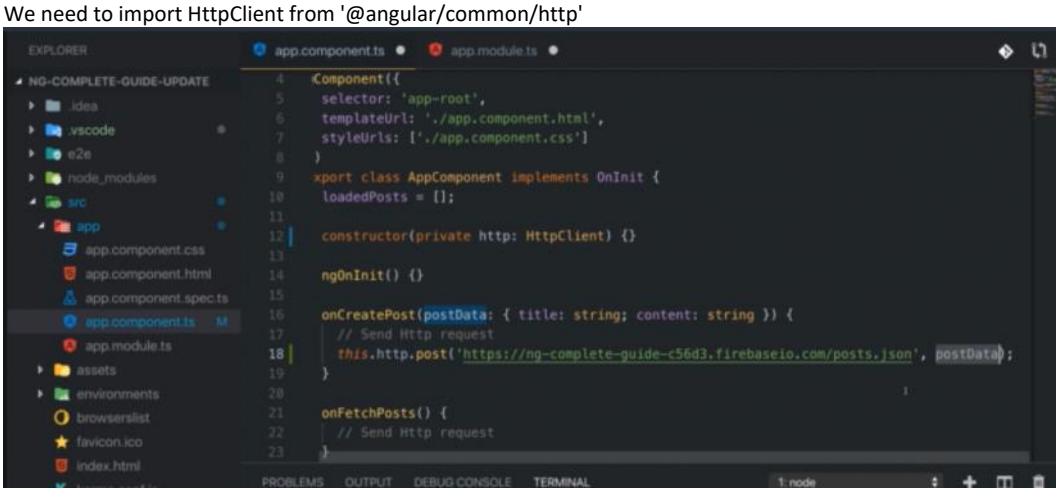
- Async pipe
  - This helps in handling asynchronous data
  - async when added as an pipe to a data which is a promise and returns a string when resolved.
  - async recognizes that this is a promise and as a side note, it would also work with observables, there it would subscribe automatically and after two seconds, it will simply recognize that something changed, that the promise resolved or in the case of an observable, that data was sent through the subscription and it will print this data to the screen.
  - 

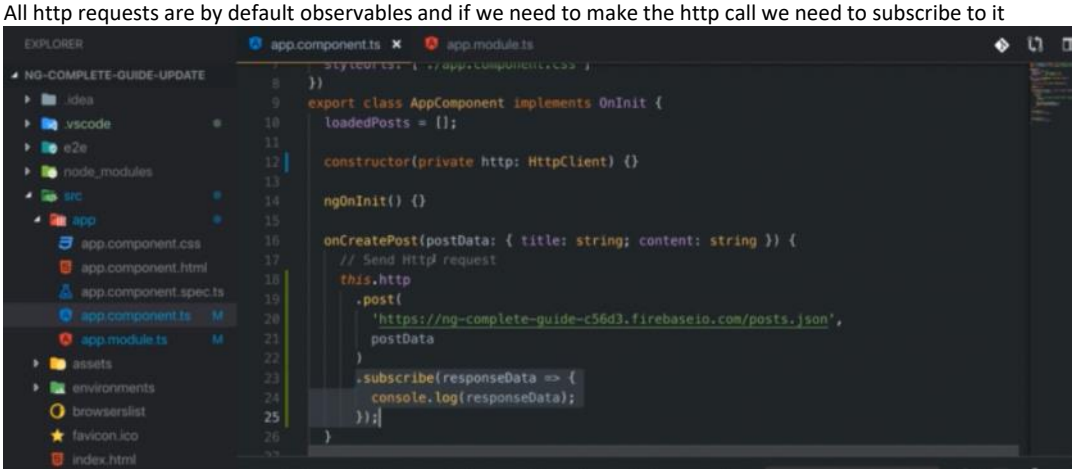
  - 

- Making Http requests

- 

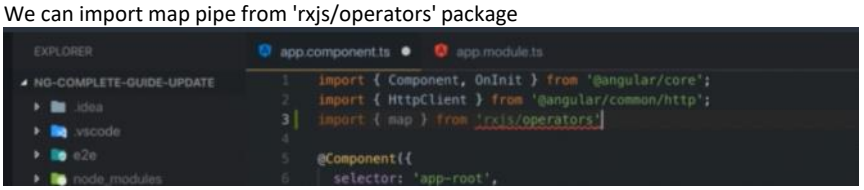
- 

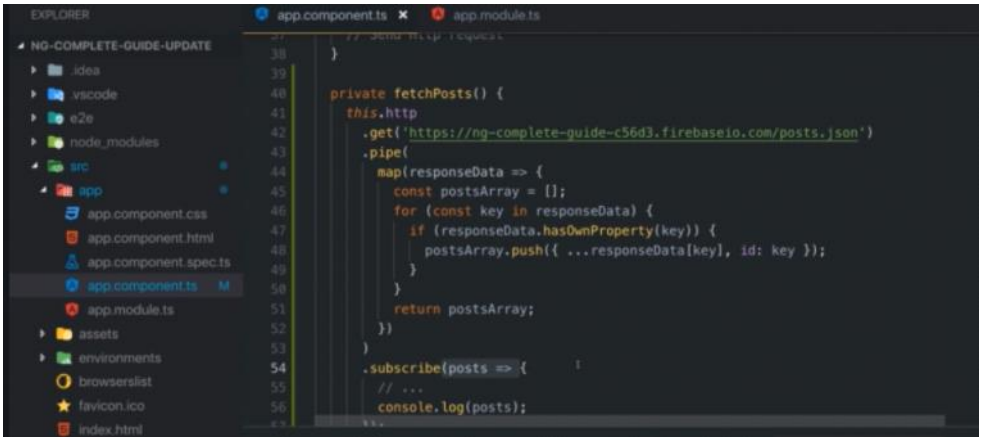
- We need to import HttpClientModule from '@angular/common/http' package to use http requests
- We need to add HttpClientModule in app.module.ts imports array
- Now to use http requests
  - We need to import HttpClient from '@angular/common/http'
  - 

  - All http requests are by default observables and if we need to make the http call we need to subscribe to it
  - 

- Get call using http client

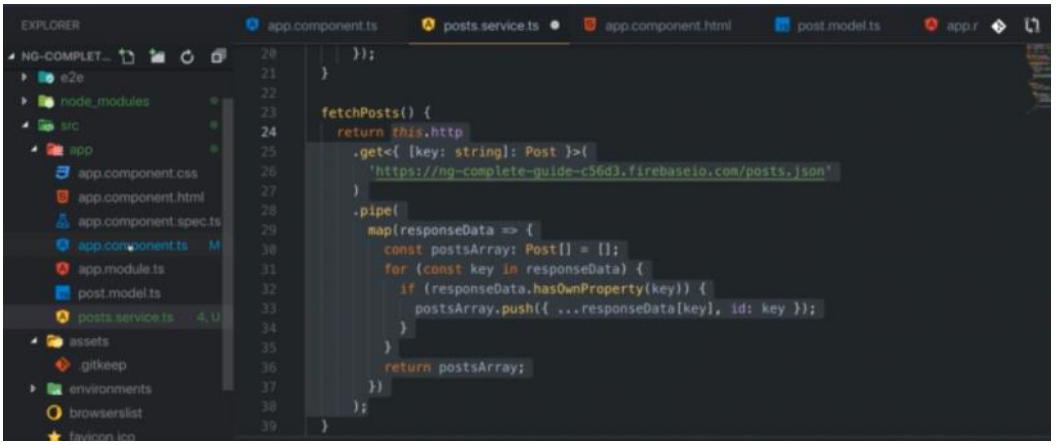
-

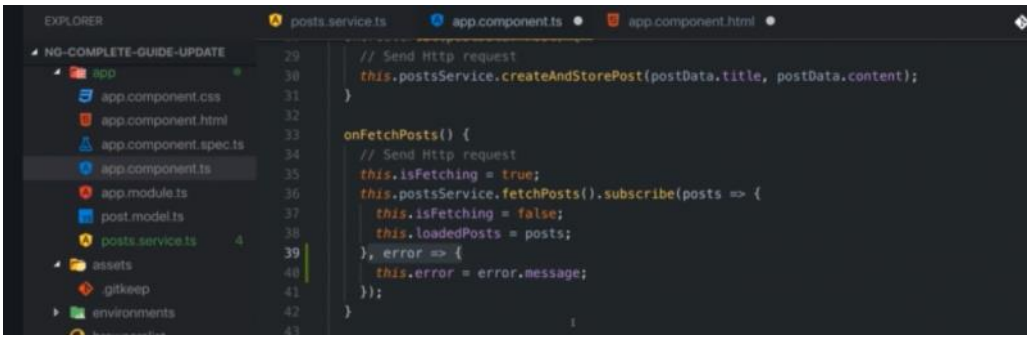
- Using Rxjs operators to transform the response data
  - We can import map pipe from 'rxjs/operators' package
  - 

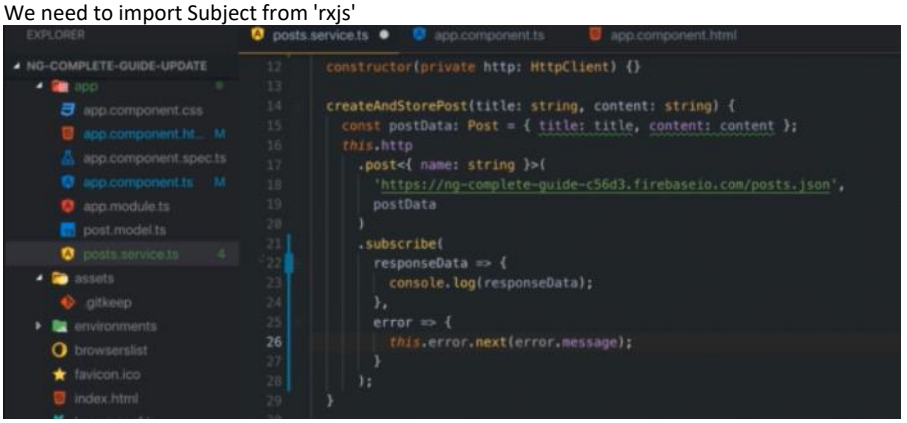
  - 

- We can also use types for our Post and get calls
  - this.httpRequest.get<return type>
  - this.httpRequest.post<return type>

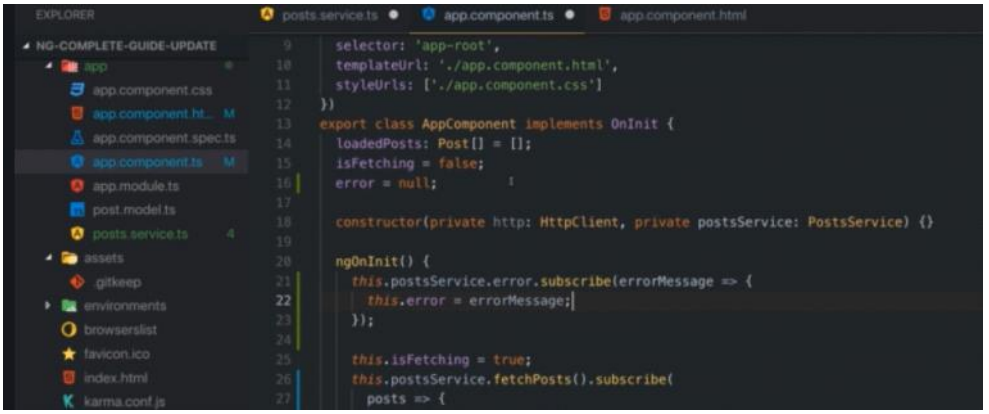
- If we are using service for our http requests we can return the observable and subscribe to the service in the component
  -

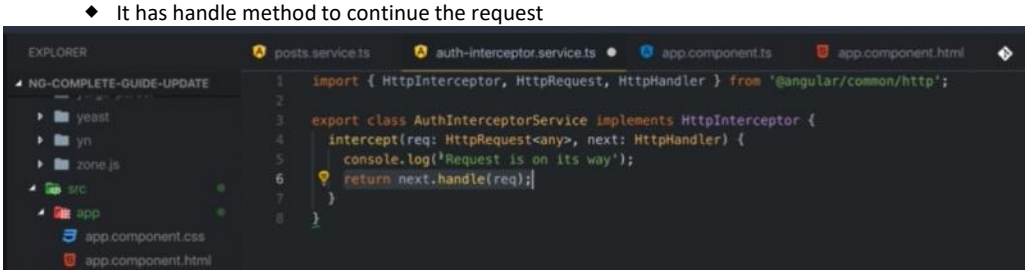
- 

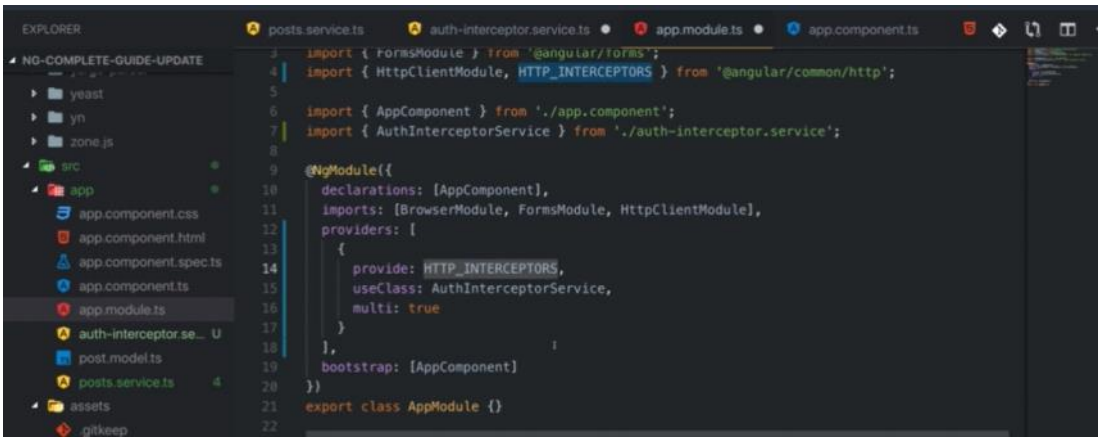
- Handling errors in http requests
  - We have an second argument for the error handling while subscribing to the observable
  - 

- Using Subject for error handling
  - We need to import Subject from 'rxjs'
  - 

  - We need to subscribe to the subject to display the error message
  - Note: We also need to unsubscribe on destroy of the component
  -

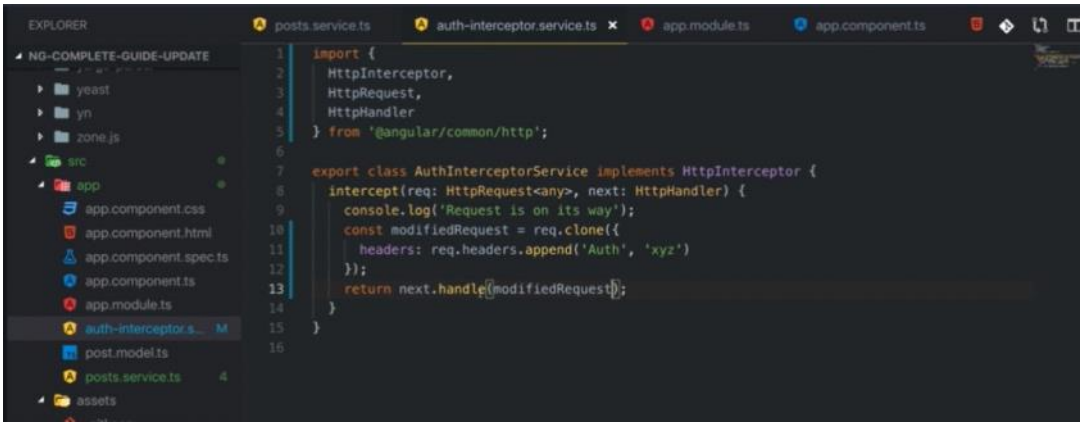
- 

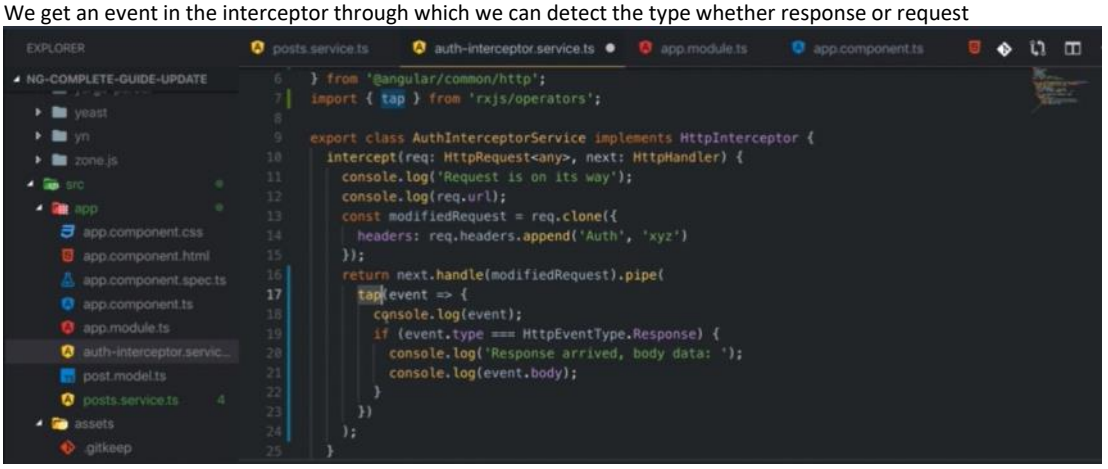
- Interceptors
  - We need to import HttpInterceptor from '@angular/common/http'
  - We have an intercept method through which we can set the common headers for all the requests
  - This intercept code will run before request leaves the app
  - intercept accepts 2 arguments
    - req of type HttpRequest which needs to be imported from '@angular/common/http' - request obj of any data it might yield
    - next of type HttpHandler which also needs to be imported from '@angular/common/http' - obj which forwards the request to continue the journey
      - It has handle method to continue the request
  - 

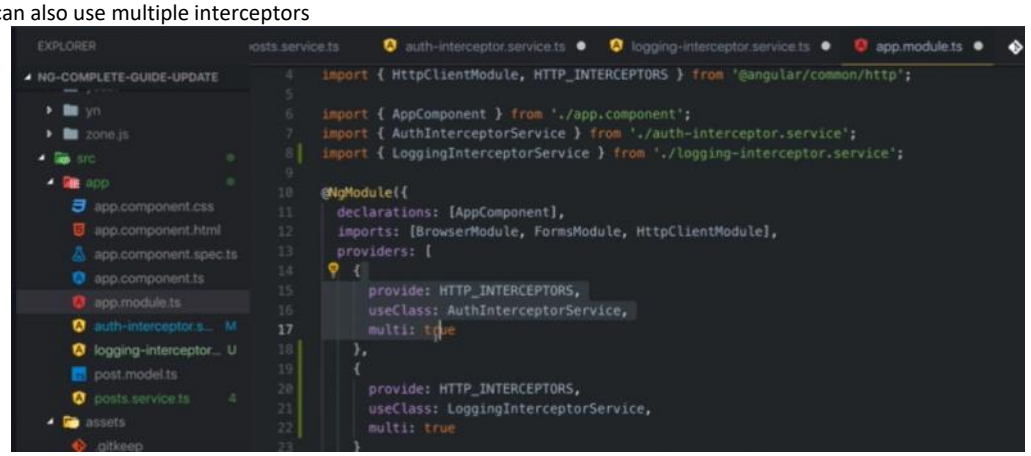
  - 

  - In app.module.ts, we need to mention the HTTP_INTERCEPTORS in the providers array though an object to make sure all the requests go through this interceptor
    - provide: HTTP_INTERCEPTORS -> Needs to be imported from '@angular/http/common'
    - useClass: Class name of our interceptor
    - multi: true -> To tell we have more than one interceptors

  - We can modify the request obj before it has to be sent
    - We cannot directly do that, but we can clone the request object and modify it
    -

- 

- We can also manipulate the data received through the request using pipe operator through the tap method
  - tap can be imported from 'rxjs/operators' to look into the response.
  - We get an event in the interceptor through which we can detect the type whether response or request
  - 

  - We can also use map to transform the response data

- We can also use multiple interceptors
  -