1. Component:

Components

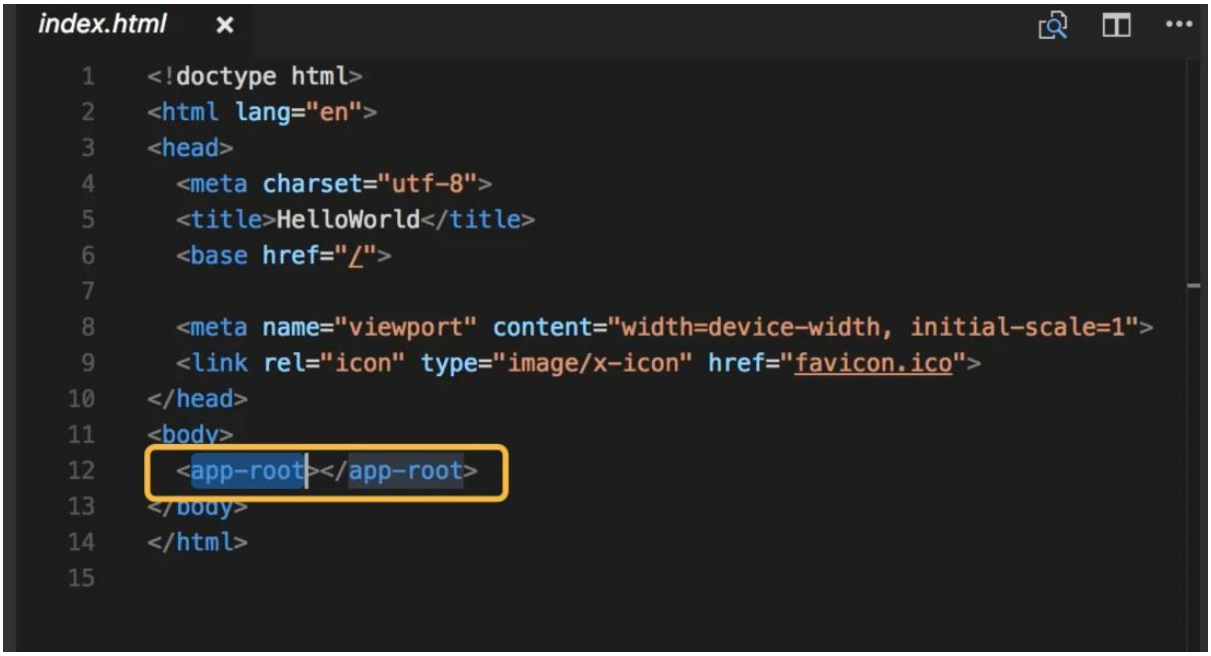
- **Create** a component
- **Register** it in a module
- Add an element in an **HTML markup**

courses.component.ts

```
1
2   import { Component } from '@angular/core';
3
4   @Component({
5       selector: 'courses',
6       template: '<h2>Courses</h2>'
7   })
8   export class CoursesComponent {
9   }
```

courses.component.ts | app.module.ts

```
1   import { CoursesComponent } from './courses.component';
2   import { BrowserModule } from '@angular/platform-browser';
3   import { NgModule } from '@angular/core';
4
5   import { AppComponent } from './app.component';
6
7   @NgModule({
8     declarations: [
9       AppComponent,
10      CoursesComponent
11    ],
12    imports: [
13      BrowserModule
14    ],
15    providers: [],
16    bootstrap: [AppComponent]
17  })
18  export class AppModule { }
```

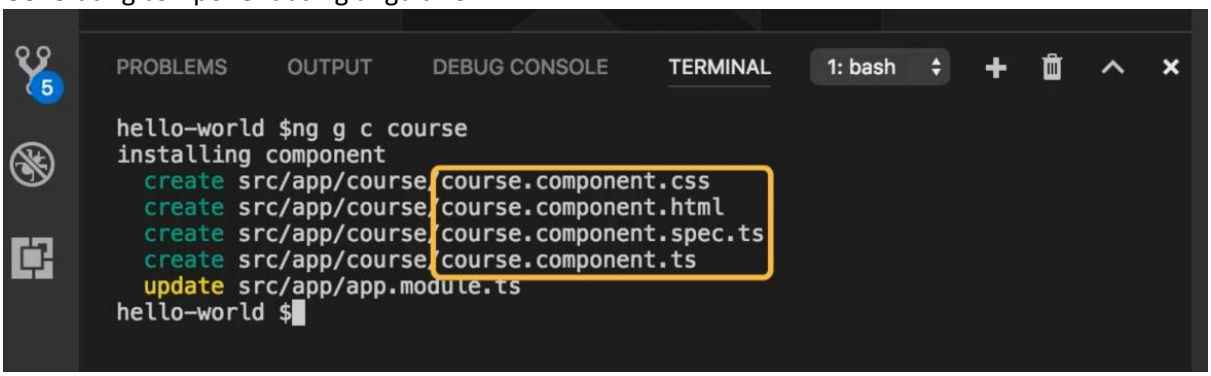


All right, Now let's see these components in action. There are basically three steps you need to follow in order to use a component. First, you need to create a component. Second, you need to register that component in a module. And third, you need to add an element in your HTML markup. Let me show you what I mean by this. So back in our first Angular project, first of all, make sure to run ng serve to serve this application. So now if you head over to localhost port 4200. This is what you see. Beautiful. Now back in VS code, I'm going to create a new component. So let's open up the file panel. Here in the source folder under app here, I'm going to add a new file. Now let's imagine I want to create a component to display the list of courses. So look how I named this file courses dot component dot ts. This is the convention that we use when building angular applications. Now, if the name of the component has multiple words, we separate them using a hyphen. For example, if you're building a component called course form, you would call this course dash form and then dot components. Okay, So courses, dot components. Now, here we start by creating a plain TypeScript class. So class. Courses component. Once again, look at the naming convention here. I'm using the Pascal naming convention. So the first letter of every word should be capital. And also by convention, we use the suffix component in the name of the class. So here's our class. In order for Angular to see this class, first, we need to export it. Now, so far we have only a plain TypeScript class. This is not a component. In order to convert this to a component, we need to add some metadata to it that Angular understands. We use a decorator to achieve this. In Angular, we have a decorator called Component that we can attach to a class to make that class a component. So first we need to import this decorator

on the top. So import curly braces. Component from now the name of the library. So at Angular core, this is the core library of Angular that you're going to see a lot in this course. So here we import the component decorator and then we need to apply it to this TypeScript class. Now look at the syntax. Use add sign component and then call this like a function. So this is what we call a decorator function. Now, this function, as you see here, takes an argument. So here we pass an object. And in this object, we add one or more properties to tell Angular how this component works. For example, one property that we use quite often is selector. So selector. And we set this to a CSS selector. So in CSS, if I want to reference an element like this, I use a selector. Like this. If I want to reference an element like a div that has a class called courses. My selector would be dot courses, right? And if I want to reference an element. With the courses. This is my selector, just basic CSS. Okay, so here we want to reference an element called courses. Why? Because with components we can extend HTML vocabulary. So we can define new elements like courses. And inside that we will have the list of courses. Or in the future we can define a custom element, a custom HTML element called rating. And wherever we use that, angular will render a rating component. Okay, so let's delete this. My selector for this component is courses. Now the second property we add here is template and that's the HTML markup. We want to be rendered for this component. Now here I want to render something very simple, so let's just add an H2 element. And call these courses in a real world application. Our templates can be several lines of code. In that case, we can put them in a separate file and I'm going to show you how that works later in the course. So this is a basic component in Angular. That was the first step. The second step is to register this component in a module. Now, currently our application has only one module, which we call app module. Let me show you where that is. So back here in the app folder, look, we have app module. Now, once again, here we have three import statements on the top. So nothing special here. On the bottom, we have a TypeScript class called App module. And note that this class is decorated with another decorator function called NG module. So with this we convert a plain TypeScript class to a module from Angular's point of view. Now don't worry about all these properties here, like declarations, import providers and bootstrap. You're going to learn about them later in the course. What I want you to focus on now is this property declarations, and this is where we add all the components that are part of this module. So by default, when we generate an application, we have one component called app component, and you can see that component is part of this module. Now I'm going to add our new component here. So courses component. So whenever I type the name of a class here and press enter. It automatically imported on top of my file. So import courses component from and here is the name of our TypeScript module. So period slash which refers to the current folder. And here's the name of the file. Courses dot component. Now note that here we don't have dot ts. In fact, if you add that TypeScript compiler doesn't like that. So the name of the module is just the name of the file without the extension. So if you want to add this plugin to your VS code, simply go here. And in the extension search box. Search for auto import. So currently I'm using auto import 1.2.2. Simply install it and then you'll have to restart vs code. All right. So here's our second step. Now the third step back to our component. So this is the selector for this component, which means anywhere we have an element like courses, Angular is going to render the template for this component inside that element. But where are we going to add that element? Let me show you. So. Back here in the app folder, we have this App.component.html. This is the external template for our app component. Let's have a look. So all this markup you see here is for rendering the home page. And this is what we get. Now, I want to simplify this. I'm going to get rid of all this markup. Add a

simple H1. Call this angular, and below that I'm going to add our custom element courses. So when Angular sees this element, it's going to render the template of our courses component. All right. Now, back in the browser, this is our new home page. So let's right click this courses element. And inspect this. So look what we have here. We have this one. And below that we have our courses element. And inside this you can see we have the template for our courses component. So this is how Angular applications work. As another example, look at this app root element inside the body element. Where is this app root? Well. Back in the source folder. Look at index.html. So this is the basic template for our application right now. Inside the body element, we have an element called app dash root. This is a custom element because we don't have an element in HTML called app root. So from Angular's point of view, we should have a component with a selector for this element and that's our app component. Let's have a look at that. So back in the app folder. Let's look at App.component.ts. So here's our root component, our app component. Look at the selector app root. So whenever Angular sees an element like that, it's going to render the template for this component inside that element. In this case, our template is external. So we have template URL and you can see the name of our template file app.component.html. So that's why inside App Root here we have the template for app component, which includes an H1 and the courses element.

2. Generating component using angular CLI



3. Templates

courses.component.ts

```ts

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

@Component({
    selector: 'courses',
    template: '<h2>{{ getTitle() }}</h2>'
})
export class CoursesComponent {
    title = "List of courses";

    getTitle() {
        return this.title;
    }
}
```

4. Directives

courses.component.ts

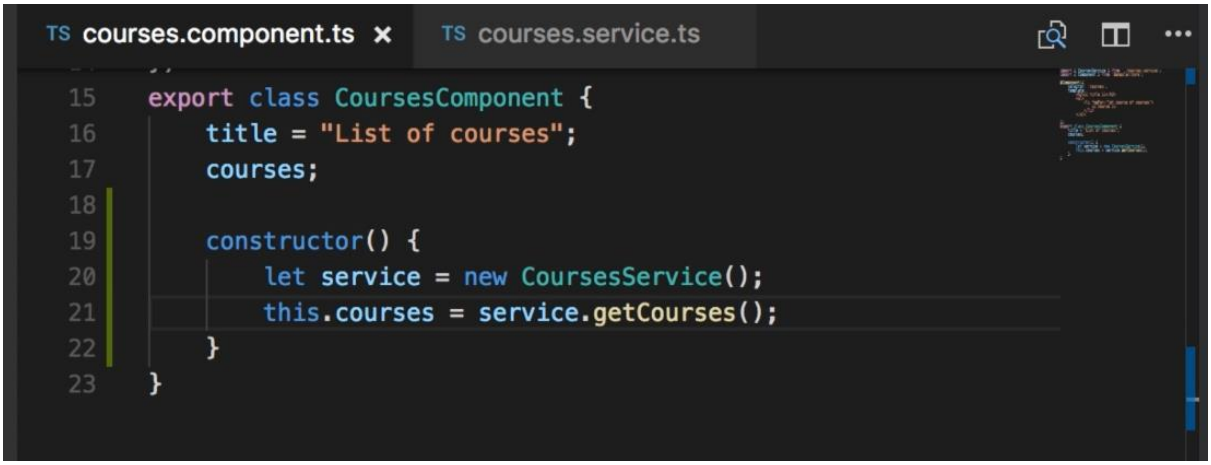
```ts

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

@Component({
    selector: 'courses',
    template: `
        <h2>{{ title }}</h2>
        <ul>
            <li *ngFor="let course of courses">
                {{ course }}
            </li>
        </ul>
    `
})
export class CoursesComponent {
    title = "List of courses";
    courses = ["course1", "course2", "course3"];
}
```

5. Services

courses.component.ts | courses.service.ts

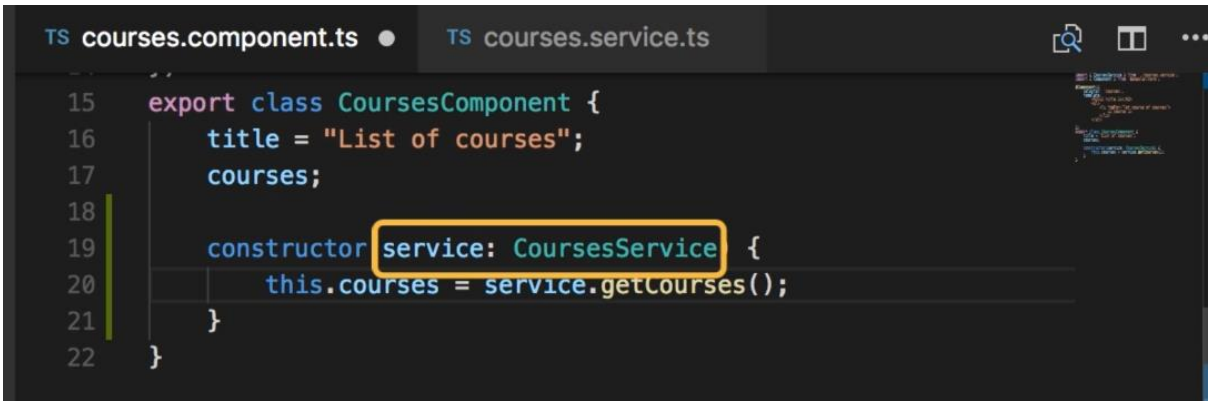
```ts

export class CoursesService {
    getCourses() {
        return ["course1", "course2", "course3"];
    }
}
```

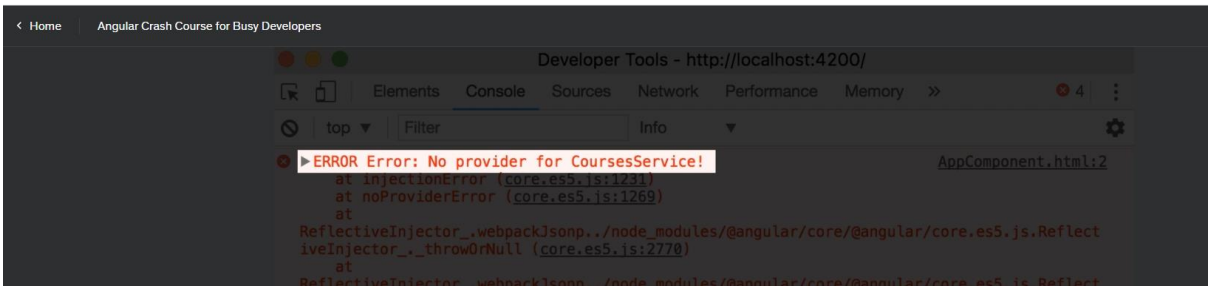
6. Dependency injection

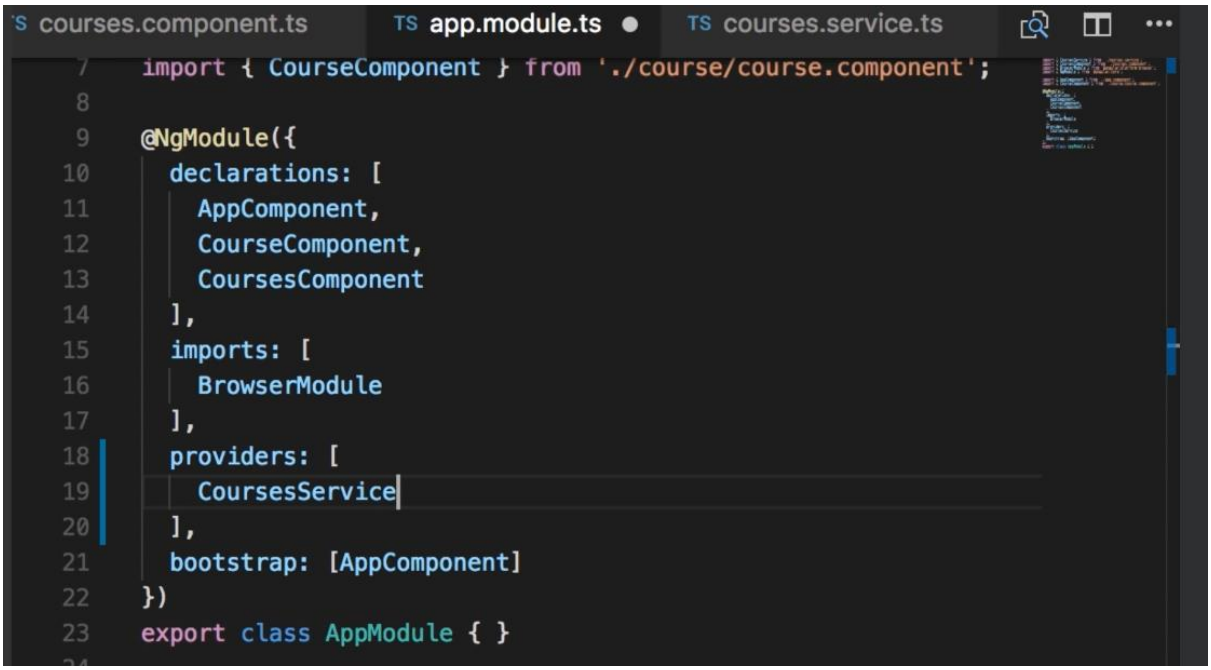


Okay, now we have a service to get the list of courses from the server. We need to use this service in this courses component. So first we need to add a constructor here. Constructor. Because a constructor is where we initialize an object. So here we need to create an instance of our courses service. Something like this. Let service be new courses service. Once again, I'm using the auto import plugin so when I press enter it automatically adds. The import statement here on the top. So import courses service from current folder courses dot service. All right. Now, back in the constructor. So we have a service and then we can initialize this courses field like this. This dot courses, we set this to service that get courses. Let's test the application and make sure everything works up to this point. Save. Okay. We get the same list. Beautiful. However, there is a problem with this implementation. The first problem is that by using this new operator here, we have tightly coupled this courses component to the courses service. So this is exactly like the problem we had earlier. If we implemented the logic for consuming an Http service inside this component, we wouldn't be able to unit test this class. Now we put this logic in a different class courses service, but because we are directly creating an instance of this class here, we're still tightly coupled to that implementation. The other issue here is that if in the future we decide to add a parameter to the constructor of courses service, we have to come back here and anywhere else in our application where we have used this courses service and add a new argument like one. So anytime we change the constructor of this service, we end up with multiple changes in our application code. So this is very fragile. So what should we do? Well, instead of recreating an instance of the courses service, we can ask Angular to do that for us. So we can delete this line here. And add a parameter in this constructor. Call it service of type courses. Service. With this. When Angular is going to create an instance of our component, it looks at this constructor. It sees that this constructor has a dependency. This dependency is of type courses service. So first it creates an instance of the courses service and then passes that to this constructor. Now with this implementation, if we change the constructor of courses service and add a new parameter, we don't have to modify 100 places in our code to reflect the change. Angular would automatically instantiate a new courses service object. The second benefit of this implementation is that when we're going to unit test this courses component instead of supplying an actual course of service to this constructor, we can create a fake implementation of this service that doesn't use that Http service on the backend. In other words, we have decoupled our courses component from courses service. So here's the lesson. When you use the new operator like this inside a

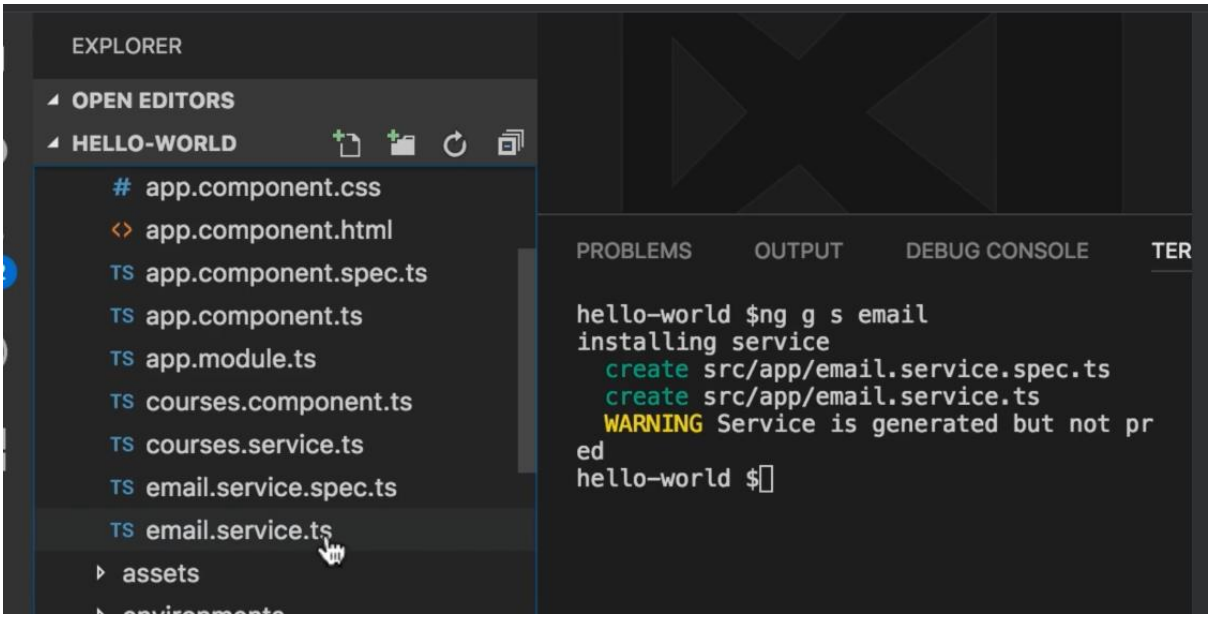
class. You have tightly coupled your class to that implementation. You cannot change this at runtime, but when you add that dependency as a parameter of a constructor, you have decoupled that class from that dependency. Now we're not done yet. We need to instruct Angular to create an instance of courses service and pass it to our courses component. This concept is called dependency injection, so we should instruct Angular to inject the dependencies of this component into its constructor. As simple as that, lets dependency injection. A lot of people think dependency injection is so complicated, but it's really a $25 term for a five cent concept. So dependency injection means injecting or providing the dependencies of a class into its constructor. Now Angular has a dependency injection framework built into it, so when it's going to create an instance of a component, it can inject the dependencies. But in order for that to work, we need to register these dependencies somewhere in our module. So let's save those changes. Now we need to go to our appmodule. So here is Appmodule. Okay. Look at this energy module decorator. Here we have a property called Providers, which is set to an empty array in this array. We need to register all the dependencies that components in this module are dependent upon. For example, our courses component is dependent upon courses service, so we need to register courses, service as a provider in this module. So. Here. I'm going to add courses. Service. Now, if you forget the step dependency injection is not going to work. Let me show you what happens. So I'm going to comment out this line, save back to the browser. We get a blank page so that means something is wrong. Let's open up Chrome developer tools with shift command and I on Mac or shift control I on Windows. Look at this error no provider for courses service. This is an error that you might see quite often when building angular apps. So the error is telling us that we have not registered courses service as a provider in our module. So back in app module, I'm going to add this courses service here. Save. Back in the browser and we have the list of courses. Now, one more thing before we finish this lecture. When you register a dependency as a provider in a module, Angular will create a single instance of that class for that entire module. So imagine in this module we have 100 components, and 50 of these components need the courses service in the memory. We're going to have only a single instance of courses, Service and Angular will pass the same instance to all these components. This is what we call the singleton pattern. So a single instance of a given object exists in the memory. So a quick roundup back in our component, we added courses service as a parameter in the constructor or in other words, as a dependency of this class and then register this. As a provider in our app module with this implementation. When Angular is going to create an instance of this component, first, it will instantiate its dependencies and then it will inject those dependencies into the constructor of this class. And this is what we call dependency injection.





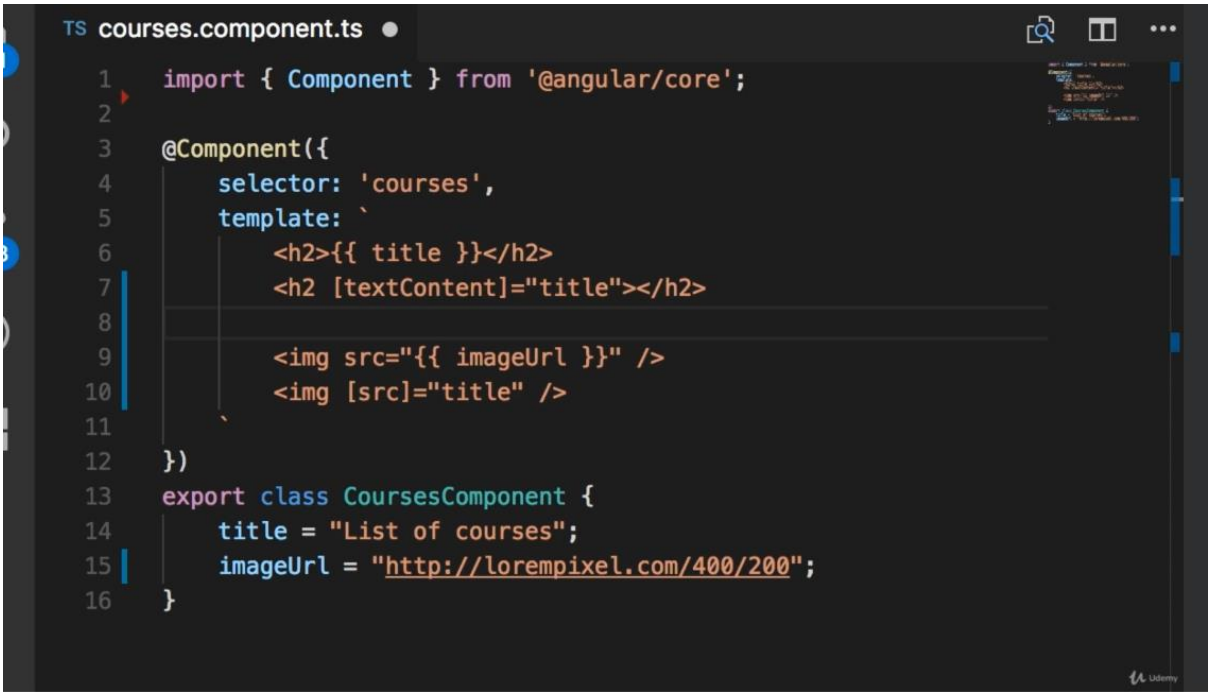


7. Generating service using CLI



All right. Now, let me show you a quick way to create a service in Angular. So we go to the terminal. Once again. You can switch to the terminal window or you can press control and backtick in vs code. So here we have a small terminal window. We're going to use angular CLI to generate a service so g g for generate. Previously we use C for components. Now we're going to use S for services. Imagine we're going to create a service for sending emails. This service is going to call an Http endpoint somewhere for sending these emails. So let's call this email and note that here I just add the service name. I don't need to add dot service. Okay, so. This generates two files for us. One is the actual service file and the other is a spec file, which includes some boilerplate code for writing unit tests for that service. So in the app folder, look, here's our new service email. That service, That's. So we have a plain TypeScript class called email service. But there is something extra here that you didn't see before. We have this injectable, which is another decorator function similar to the component decorator function. Why do we need this here? We would need this decorator only if this service had dependencies in its constructor. For example, imagine here we had a dependency like log service of type log service. In this case, we need to apply this injectable decorator function on this class. And this tells Angular that this class is an injectable class, which means Angular should be able to inject dependencies of this class into its constructor. Now, we didn't use this decorator when defining components because when we use the component decorator, that decorator internally includes this injectable decorator. So that's all about injectable. And by the way, note that this decorator function is defined in angular core library.

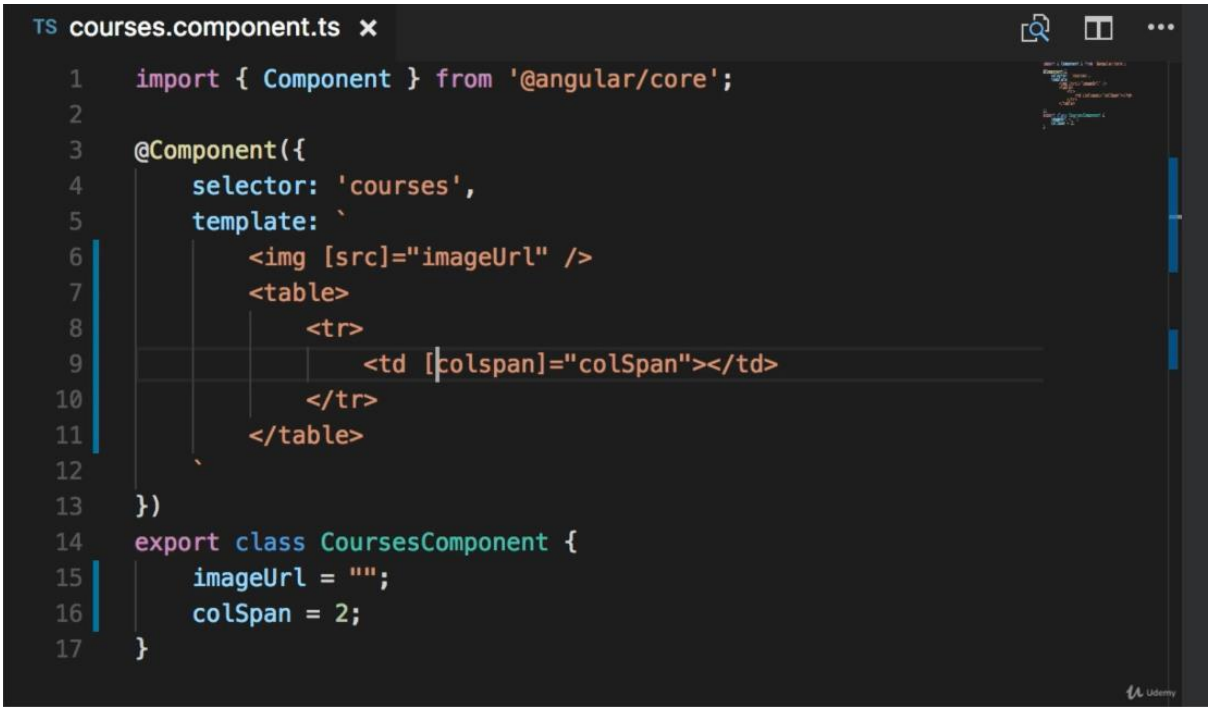
8. Property binding

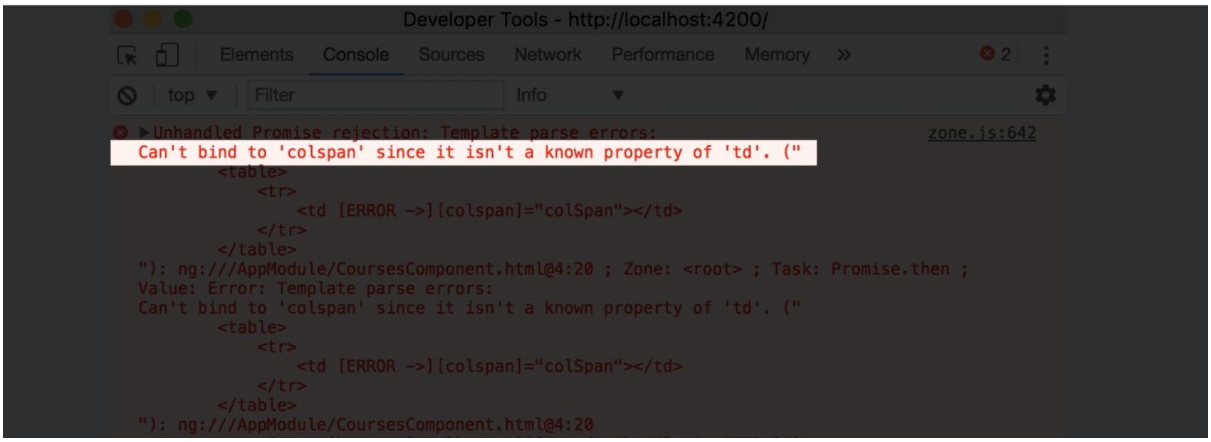


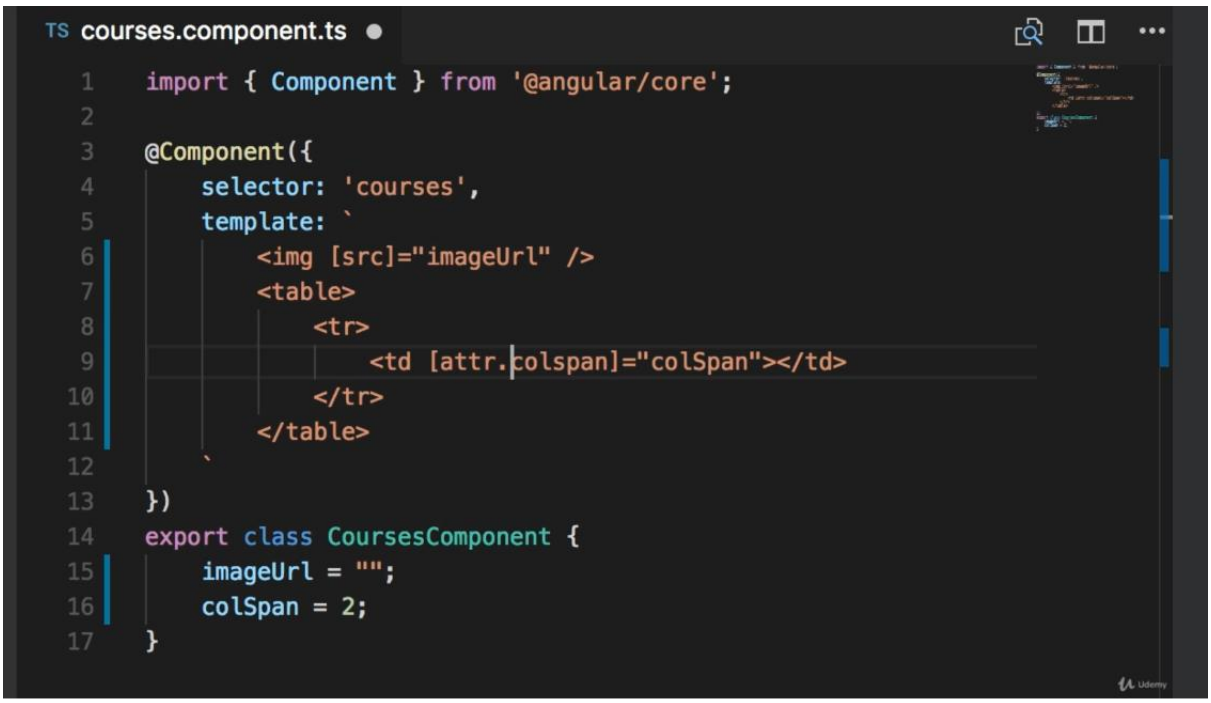
So earlier you learned about interpolation. It's the double curly braces syntax we use to display data. So here we have two examples. One is for rendering the title field and the other is for the image URL and with this template. We get something like this. Now this interpolation is just a syntactical sugar behind the scene. When Angular compiles our templates, it translates these interpolations into what we call property binding. With property binding, we bind a property of a Dom element like source here to a field or a property in our component. Now let me show you the syntax for using property binding. I'm going to rewrite the second example, but instead of using interpolation, I'm going to use property binding. So image. I want to bind the source property. So I put it between square

brackets. Source. And for the value I use the title field of courses component. Note that here we don't have double curly braces, so when we use double curly braces or string interpolation, Angular behind the scene translates that into the second syntax. And with this, whenever the value of the title field changes, the source attribute of this image element is automatically updated. Now here you might have a question. You might be wondering whether you should use string interpolation or the square bracket syntax. Well, interpolation works well for adding dynamic values between headings like here, between the H2 element, or if you're using divs spans paragraphs or wherever you want to render text, that's when you use string interpolation. In this example, if I want to use property binding with this H2 heading, look what I have to write. So. H2. Now I have to bind the text content. Property of H2 dom element have to bind this. So the title field of the courses component. Obviously you can see the second syntax is longer and more noisy, so that's why I prefer to use string interpolation wherever I want to add text between headings, paragraphs, divs and spans. In other cases, you can see the property binding syntax using the square brackets is actually shorter. Now there is nothing wrong with using string interpolation here, but the second syntax is cleaner and shorter. Now the syntax aside, one thing you need to know about property binding is that it works only one way from component to the Dom. So if these fields in the component change Angular will update the Dom, but any changes in the Dom are not reflected back in the component. So if we add an input field here and the user types something in the input field, the underlying property or field in the component will not be updated. We have two way binding for that and I will explain that later in this section.

9. Attribute binding





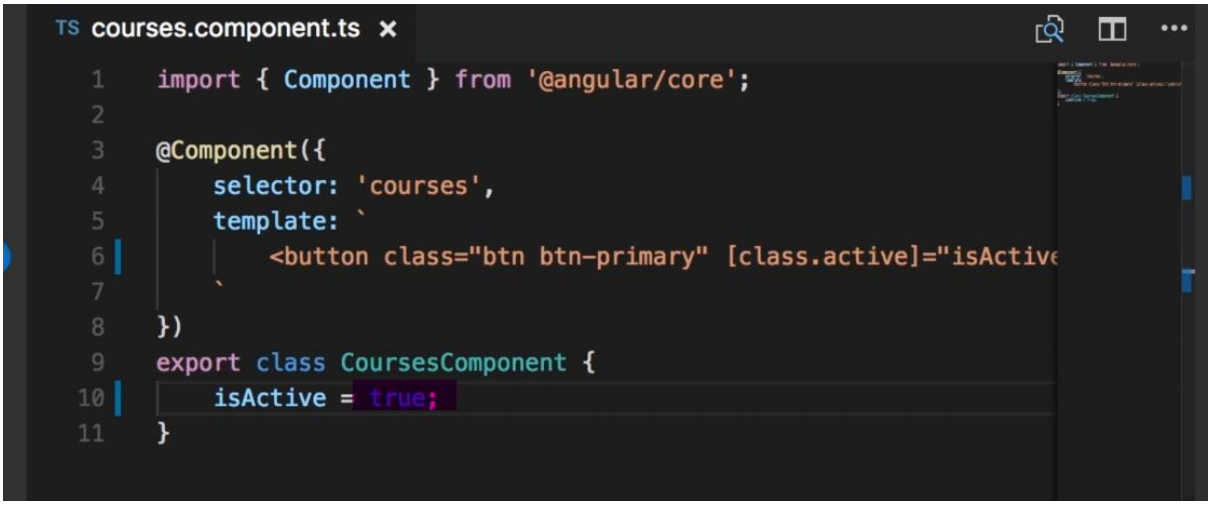


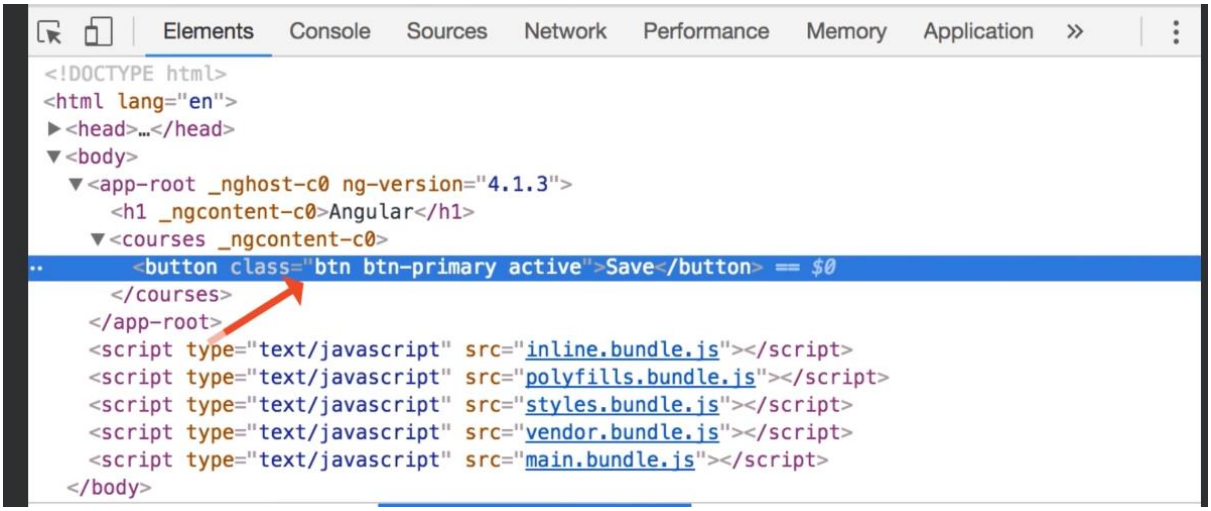
So in the last lecture you learned about property binding. So we put a property between square brackets and then bind it to a field or a property in our TypeScript class. Now, let me show you another example. This time I'm going to create a table. Inside this table. We're going to have a row. And one column like this. Now. This time I want to bind this calspan property to, let's say, Calspan field in our class. And then I'm going to set this to two. Save. Now back in the browser. We got a blank page, which means something is not right. So let's take a look at the console. Once again, the shortcut is shift command and I on Mac and Shift Ctrl I on Windows. All right. Look at the error. Can't bind to call span since it isn't a known property of TD. What's going on here? Well, in order for you to understand this error, first you need to understand the difference between Dom or document object model and HTML. Dom is a model of objects that represents the structure of a document. It's essentially a tree of objects in memory. On the other hand, is a markup language that we use to represent Dom in text. So when your browser parses an HTML document, it creates a tree of objects in memory that we refer to as the Dom. Now we can also create this tree of objects programmatically using vanilla JavaScript. So we don't necessarily need HTML, but using

HTML is far simpler. Now here's the key thing you need to know. Most of the attributes of HTML elements have a 1 to 1 mapping to properties of Dom objects. There are, however, a few exceptions. For example, we have HTML attributes that don't have a representation in the Dom. Here span is an example of that. So when we parse this HTML markup and create an actual Dom object for this TD, the Dom object does not have a property called Colspan. And that's why we get this error. So Colspan is an unknown property of TD. Also, we have properties in Dom that do not have a representation in HTML. For example, earlier I showed you that we can bind to the text content property of H1. This is a property of a Dom object and in HTML we don't have such an attribute. Now when using property binding, you should remember that you're actually binding to a property of a Dom object and not an attribute of an HTML element. Once again, in 99% of the cases, these HTML attributes and Dom properties have a 1 to 1 mapping, but we have a few exceptions here and there. So here we're dealing with the call span attribute. If you want to bind this attribute of the TD element, you need a slightly different syntax. So in square brackets you need to prefix this attribute with Attr, which is short for attribute and then dot. This way. We're telling Angular that we're targeting the call span attribute of an HTML element in this case TD. Back in the browser. We don't have any more errors.

10. Class binding

    There are times that we may want to add additional classes to an element based on some condition. For example, here we want to apply the active class on this button based on the state of the underlying component. We use a variation of property binding for that. So. We start with the property binding syntax. We add the class property here. Now, Dot and here we specify the name of the target class active. Now we bind this to a field or a property in our class in our component. So let me define a field here. Is active and set this to true. Now I'm going to use this field. Here is active. Save. Back to the browser. All right, let's right click this button. Inspect it. Look here we have three classes BTN, BTN, primary and active. If I come back here and change the value of this field to false save. Now, look, the active class is gone. So we refer to this syntax as class binding. If this condition here evaluates to true, this target class will be added to this element. Otherwise, if it's false and this class exists on the element, it will be removed. Also, I want to clarify that I separated this class binding from the actual class attribute because I want to keep these two classes, BTN and BTN Primary at all times. I only want to add the active class dynamically based on some condition.



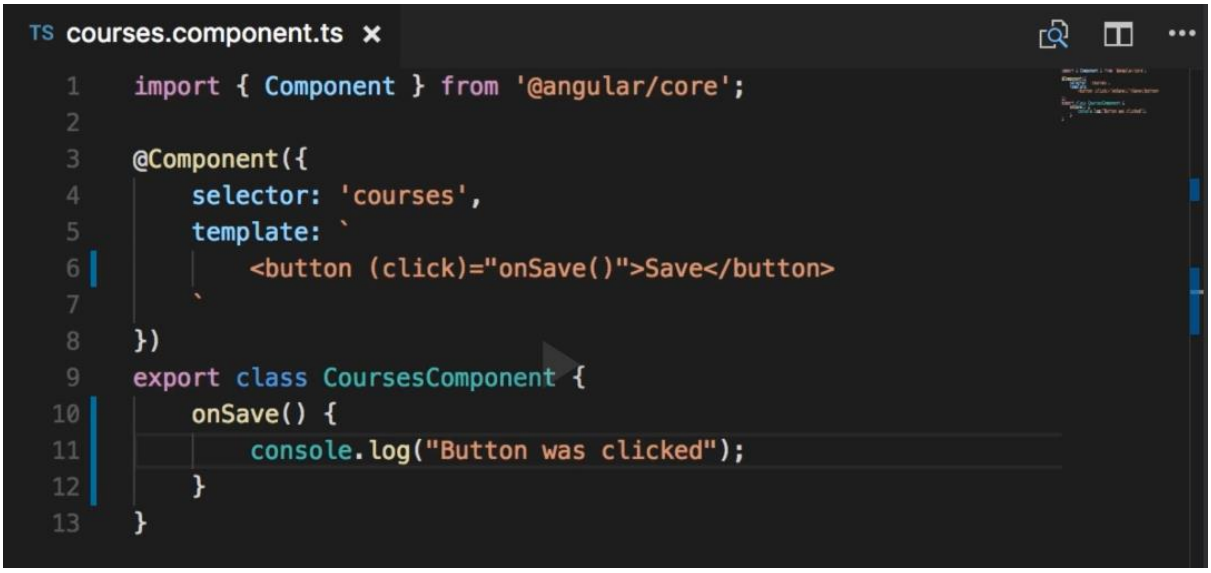


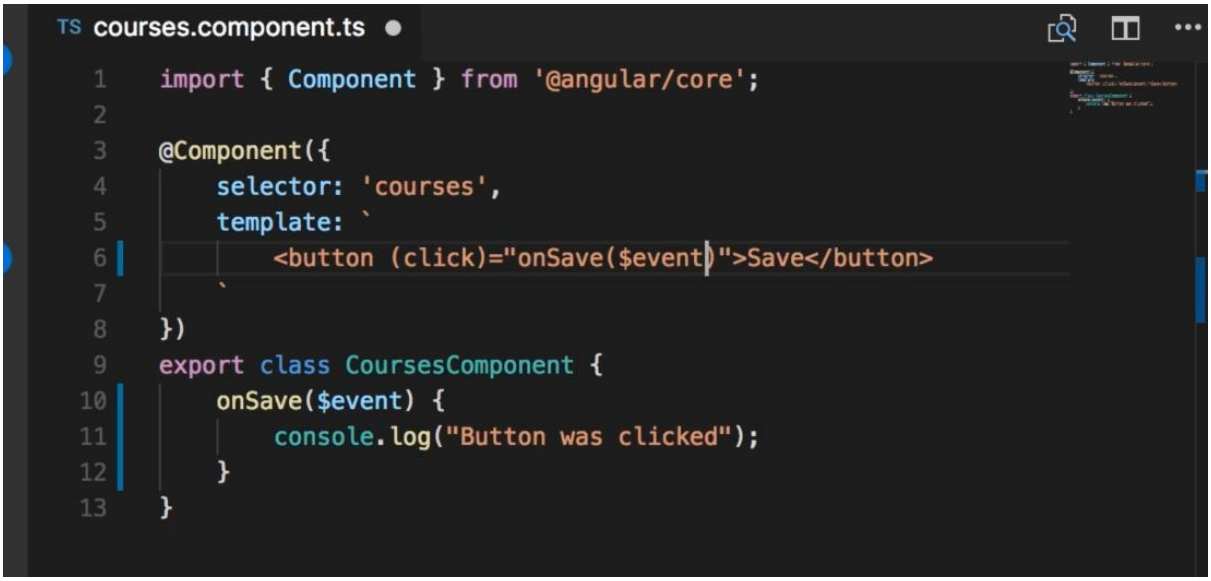
11. Style binding

12. Event binding

All right. So far you have learned about property binding and its variations like class binding, style binding and attribute binding. To add something in the Dom to display data in Angular, we also have event binding which we use to handle events raised from the Dom like keystrokes, mouse movements, clicks and so on. So here we have a simple button and we want to handle the click event of this button. So instead of square brackets, we use parentheses. And here we add the name of the event like click. Then we bind this to a method in our component. So let's call this on save. Now, here, create this method on save. And simply log a message in the console. So console dot log. Button was clicked. Let's try it. So here's our button. Click. Now let's take a look at the console. Okay, here is the message. Now, sometimes we need to get access to the event object that was raised in the event handler. For example, with mouse movements, the event object will tell us the X and Y position. If we want to get access to that event object, we need to add that as a parameter here. So dollar event. And then when calling this method, we also pass dollar event. This dollar event object is something known to Angular. Now, in this case, we're dealing with a Dom object. That's the button. So this dollar event object will represent a Dom event, a standard Dom event that you have seen in vanilla JavaScript and jQuery. Later, I will show you how to create custom components. And there this dollar event object will represent a custom event that you define in your application. So back in our unsaved method, let's log this event dollar event. Save back in the browser. So one more time, click. Okay, here's our dollar event object. All these properties you see here are part of the standard Dom event objects. So we have the movement X and y offset x and Y and many other properties. Now all the dumb events bubble up the dom tree. Unless a handler along the way prevents further bubbling. This is just a standard event propagation mechanism in Dom and is nothing specific about Angular. For example, let me wrap this button with a div. Like this. Now in this div I also want to handle the click event. So paranthesis click. I want to bind this to a different method. So let's call this on Div clicked. So. Let's add this method here on div clicked. And here I want to log a different message. Div was clicked. Now let's see what happens. So when we click this button. We get two messages in the console. The first one

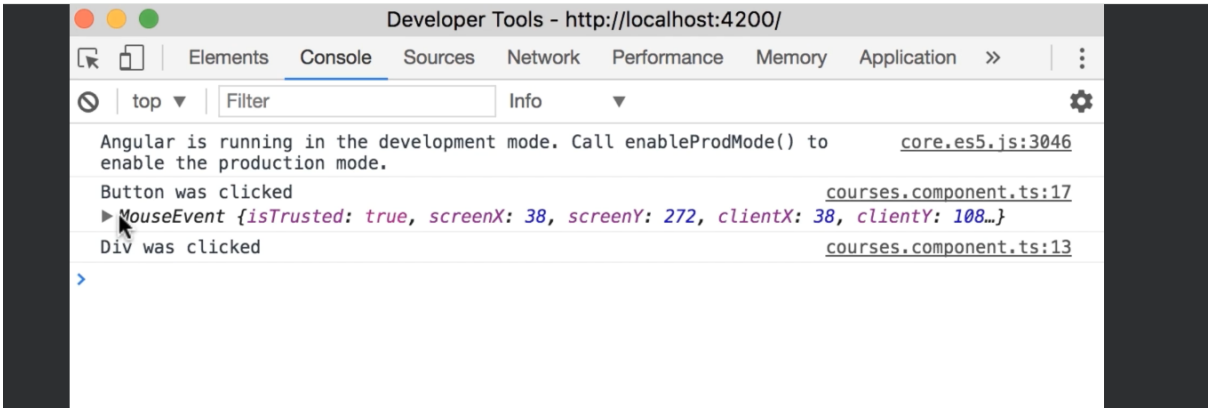
is the handler for the click event of the button, and the second is from the handler of the click event of the div. So this is what we call event bubbling. An event bubbles up the Dom tree. So to extend this example, if we had another div or another element that contained this div and we handle the click event on that element, our event object will bubble up and hit that target handler. Now how can we stop event bubbling? Well, here in Unsave we can call dollar event dot stop propagation. Again, this is a standard method you have seen before in vanilla JavaScript. So when we call this, this event will not bubble up. In other words, it's not going to hit the second handler. Let's try it. Save. Back in the browser. Click. Now look in the console. We only have one message that's coming from the handler of the click event of the button.

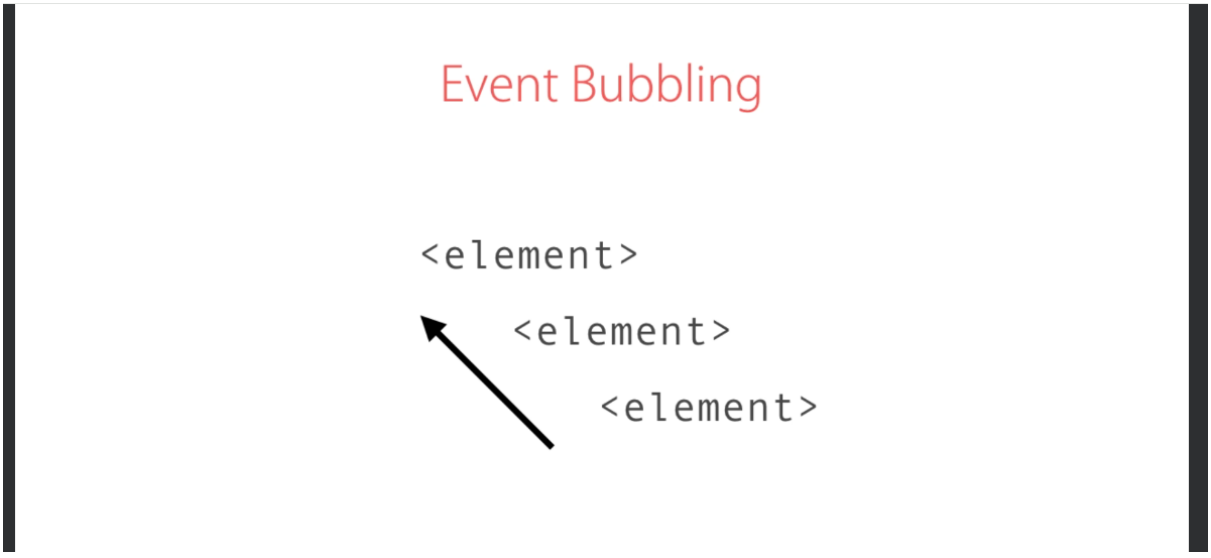




TS courses.component.ts

```
1   import { Component } from '@angular/core';
2
3   @Component({
4       selector: 'courses',
5       template: `
6           <div (click)="onDivClicked()">
7               <button (click)="onSave($event)">Save</button>
8           </div>
9       `
10  })
11  export class CoursesComponent {
12      onDivClicked() {
13          console.log("Div was clicked");
14      }
15
16      onSave($event) {
17          console.log("Button was clicked", $event);
18      }
```





```
TS courses.component.ts ●
9       `
10  })
11  export class CoursesComponent {
12      onDivClicked() {
13          console.log("Div was clicked");
14      }
15
16      onSave($event) {
17          $event.stopPropagation();
18
19          console.log("Button was clicked", $event);
20      }
21  }
```

13. Event filtering

```
TS courses.component.ts ×
1   import { Component } from '@angular/core';
2
3   @Component({
4       selector: 'courses',
5       template: `
6           <input (keyup)="onKeyUp($event)" />
7       `
8   })
9   export class CoursesComponent {
10      onKeyUp($event) {
11          if ($event.keyCode === 13) console.log("ENTER was presse
12      }
13  }
```

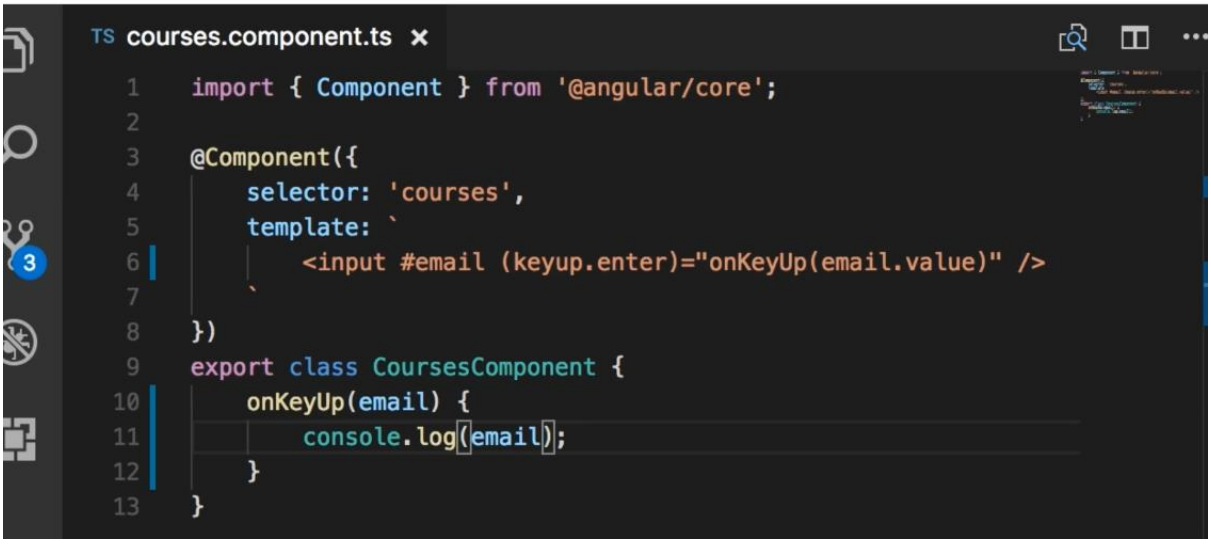
TS courses.component.ts ●

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

@Component({
    selector: 'courses',
    template: `
        <input (keyup.enter)="onKeyUp()" />
    `
})
export class CoursesComponent {
    onKeyUp($event) {
        if ($event.keyCode === 13) console.log("ENTER was presse
    }
}
```

14. Template variable

TS courses.component.ts ●

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

@Component({
    selector: 'courses',
    template: `
        <input (keyup.enter)="onKeyUp($event)" />
    `
})
export class CoursesComponent {
    onKeyUp($event) {
        console.log($event.target.value);
    }
}
```



All right. Continuing from our last example, let's imagine we want to get the value that was typed into this input field. How can we do this? There are two ways. One way is to use the event object. So once again, we pass this event object here. At this parameter to our method, and I'm going to remove this message and simply log daughter event. Now this is a standard event object in Dom, so it has a target property that references this input field. And from here we can get the value. Let's try this back in the browser. So if I type angular and press enter. Now in the console we have angular, beautiful. Now in Angular, we have another way to solve the same problem. Instead of passing this event object around, we can declare a variable in our template that references this input field. So here I declare a variable like this. Let's imagine this input field is for capturing the user's email. So I declare a variable using this hashtag or the pound sign. And this is the name of our variable. We call this a template variable, and this references this input field. Now, instead of passing this event, object around. We can pass email dot value. Okay. And then as the parameter to this method, we can simply receive the actual email address and log in on the console. Like this. Let's try it. So me at example.com enter. And here in the console, we got the email address. So in a lot of cases, you can use template variables to simplify your code.

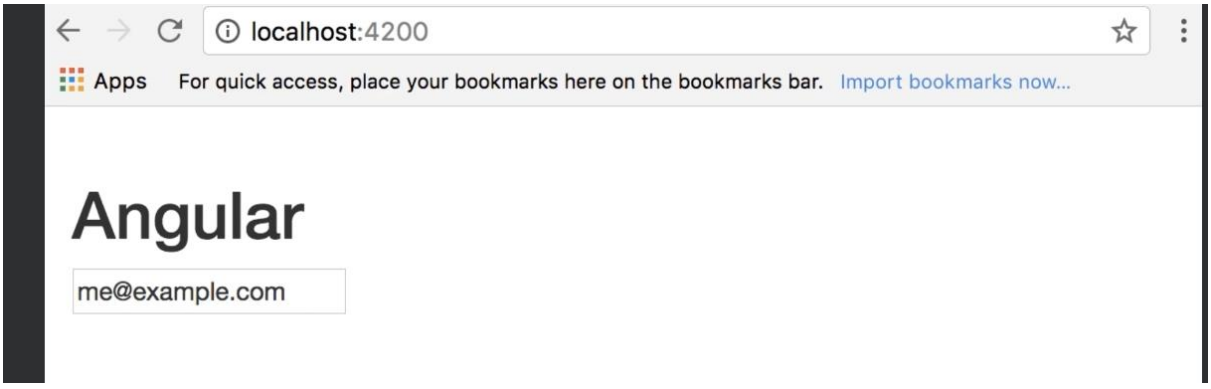
15. Two way binding

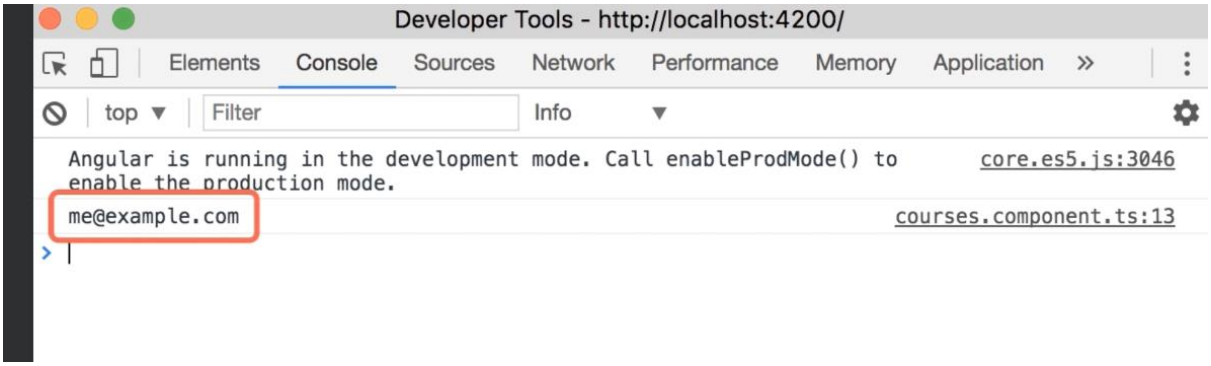
TS courses.component.ts

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

@Component({
    selector: 'courses',
    template: `
        <input #email (keyup.enter)="onKeyUp(email.value)" />
    `
})
export class CoursesComponent {
    onKeyUp(email) {
        console.log(email);
    }
}
```

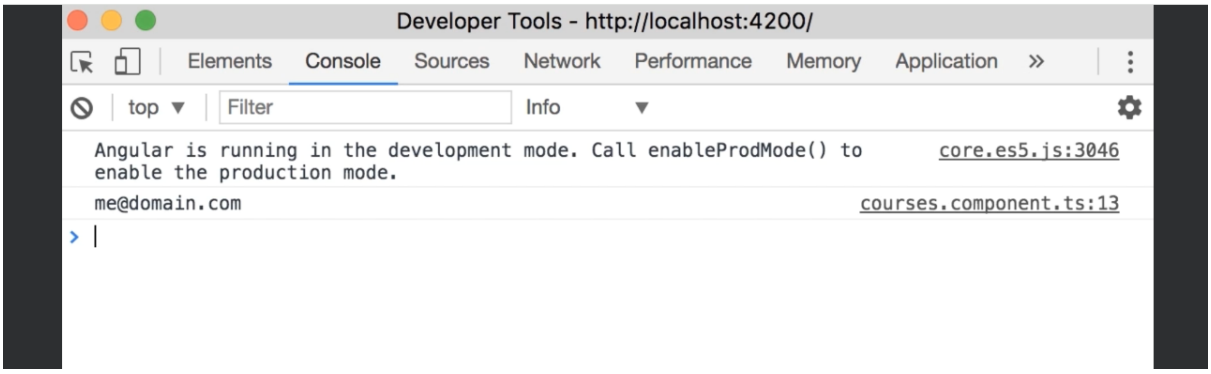
TS courses.component.ts

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

@Component({
    selector: 'courses',
    template: `
        <input [value]="email" (keyup.enter)="onKeyUp()" />
    `
})
export class CoursesComponent {
    email = "me@example.com";

    onKeyUp() {
        console.log(this.email);
    }
}
```



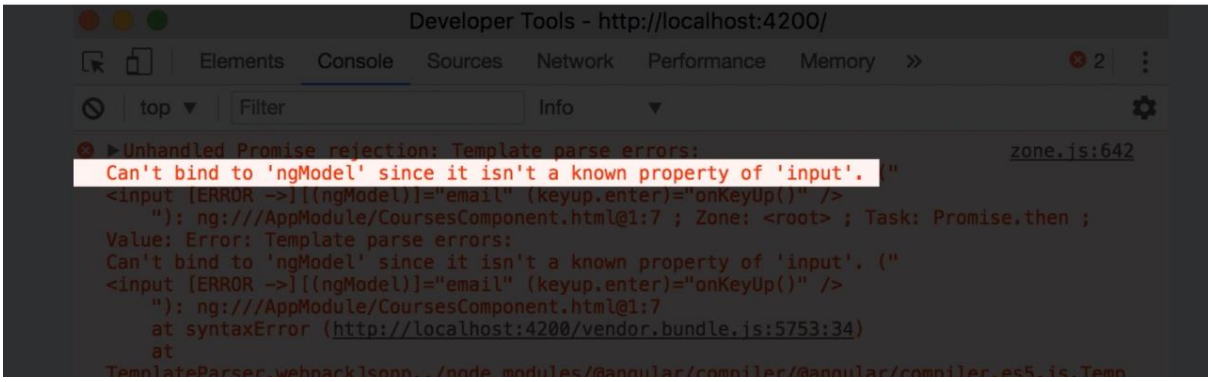

Developer Tools - http://localhost:4200/
Elements
Console
Sources
Network
Performance
Memory
Application
top
Filter
Info
Angular is running in the development mode. Call enableProdMode() to enable the production mode.
core.es5.js:3046
me@example.com
courses.component.ts:13


courses.component.ts
rom '@angular/core';
',
="email" (keyup.enter)="email = $event.target.value; onKeyUp()"
mponent {
e.com";
is.email);


Developer Tools - http://localhost:4200/
Elements
Console
Sources
Network
Performance
Memory
Application
top
Filter
Info
Angular is running in the development mode. Call enableProdMode() to enable the production mode.
core.es5.js:3046
me@domain.com
courses.component.ts:13

courses.component.ts

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

@Component({
    selector: 'courses',
    template: `
<input [(ngModel)]="email" (keyup.enter)="onKeyUp()" />
    `
})
export class CoursesComponent {
    email = "me@example.com";

    onKeyUp() {
        console.log(this.email);
    }
}
```



app.module.ts

```
import { AuthorsComponent } from './authors/authors.component';

@NgModule({
  declarations: [
    AppComponent,
    CourseComponent,
    CoursesComponent,
    AuthorsComponent
  ],
  imports: [
    BrowserModule,
    FormsModule
  ],
```

Now, this code that we wrote in the last lecture, it works, but we could do better. What I don't like about this implementation is passing this email parameter or argument around in object oriented languages where we have the concept of objects. We shouldn't really pass parameters around because an object encapsulates some data and some behavior. So if an object or a class has all the data it needs, we don't have to pass parameters around this kind

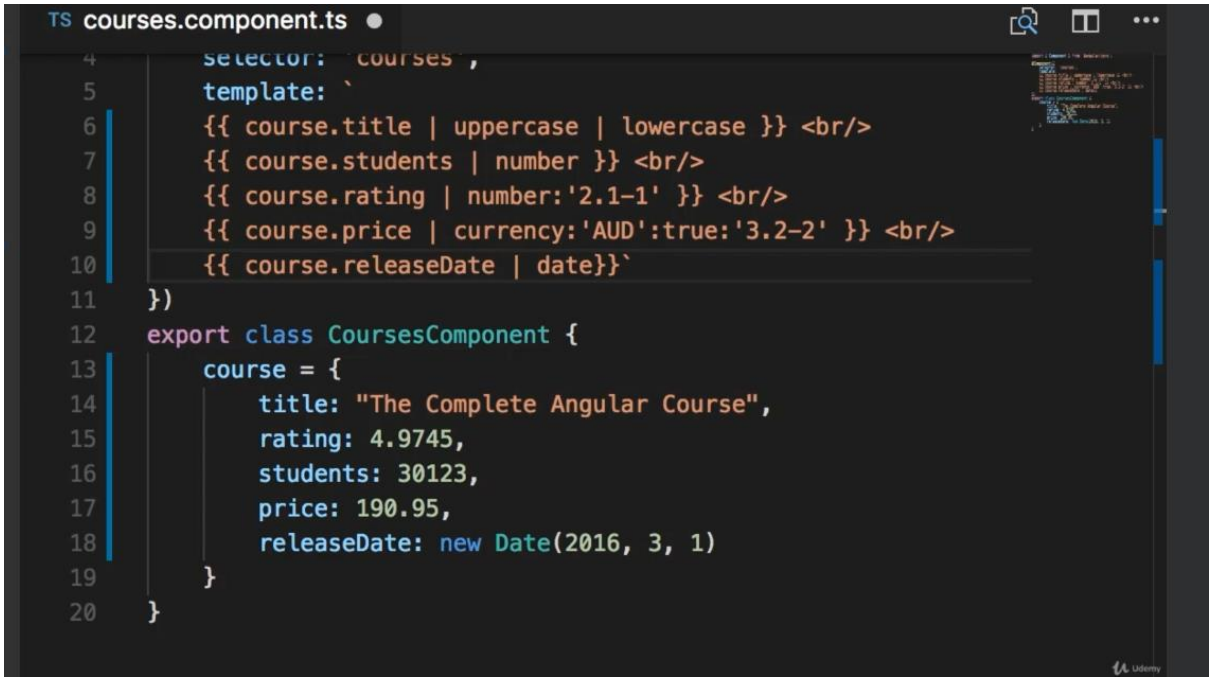
of code. We have here is what we refer to as procedural programming. That's the kind of code we use to write. 30 years ago, before we had object oriented programming languages. So imagine if we had a field here called email, then we wouldn't have to pass this email around. And here we could simply log this. The email. So you can see our method has no parameters and this is cleaner, easier to read, easier to understand, and easier to maintain. Also, going back to the definition of a component in Angular, remember a component encapsulates the data, the logic and the HTML markup behind the view. Here. The email field is used to encapsulate the data and the Onkeyup method represents the behavior or the logic behind this view. And of course, here is our HTML template. Now, how do we get here? Well, first of all, we don't need this template variable anymore. So delete. Also, we're not going to pass that here. So our code is a little bit cleaner now. Now, earlier you learned about property binding so we can bind the value property of this input object in the Dom. To this email field. Right. And now if I initialize this. So let's say me at example.com. When we load this page, the input field should be populated with this email address. So look, we've got me at example.com. However, if I change this to Domain.com and press enter. Knowing the console. Look, we got me at example.com. So how come we didn't get me at Domain.com? Because with property binding, the direction of binding is from the component to the view. So if the value of this email field changes at some point in the future, the view will be notified and this input field will be automatically updated. Now what we need here is a slightly different kind of binding. We want a binding that works in two ways from component to the view and from the view to the component. So if we type something in the input box, we want this email field to be updated. Let me show you one way to implement this. And of course, this is not the best way, but I'm going to show you a better way in just a few seconds. So with this value property binding, we have one direction from component to the view For the other direction, we can modify this expression here. So instead of directly calling the Onkeyup method. First we can set email to dollar event dot target dot value. Then semicolon and then call the on key up method. So what I want you to pay attention to here is that for the value of event binding, you can write any expression. So here we have two statements one for setting the email field and the other to call the on key up method. Now let's try this code, save back in the browser. So we've got me at example.com and I'm going to change this to Domain.com. Now enter. Look in the console. We got me at Domain.com. Beautiful. So with this implementation, we have two way binding. But as you can imagine, this is the kind of feature that we may need frequently in a lot of applications. We don't want to write all this repetitive code. Is there a better way? Of course there is. In Angular, we have a special syntax for implementing two way binding. So let me duplicate this line so you can see the difference. Instead of using property binding on the value property. We use the two way binding syntax, which includes square brackets and parentheses. Now, if the syntax is complicated or you may forget it, I give you a tip. This is called banana in a box. So this is a banana and this is a box. So banana in a box. Now instead of value, we bind to NG model. What is this? Well, our Dom objects are input. Dom object doesn't have a property called NG model. So this is something that Angular adds to this Dom object. Now, earlier you saw Ng4. Remember, Ng4 is a directive, and we use directives to manipulate the Dom. So in Angular we have another built in directive called NG model that is used for implementing two way binding. So this implementation we have here is encapsulated in a generic way inside a directive called NG model. And with this we don't have to repeat this code every time. So. Then we can simplify this expression. We can delete this statement. And in Carpenter we simply call the on key method. Now, look, the second line is obviously cleaner, shorter and easier to understand. So here's the lesson.

Whenever you want to use two way binding, use the banana in a box syntax and bind to the Ngmodel property. Now let's try this. Save. All right, we got this error. Can't bind to engine model since it isn't a known property of input. That's a familiar error, isn't it? So basically, our input object doesn't have a property called Ngmodel. It's something that Angular adds here. But why are we getting this error? Angular framework consists of several different modules. In every module we have a bunch of building blocks that are highly related. We have components, directives and pipes, and these are highly related. Now, not every application needs all the modules in Angular because when you bring all these modules, you increase the code size. So this module directive is defined in one of the modules called forms. And by default, this is not imported into your application. So if you want to use NG model or if you want to build any kind of form, you need to explicitly import this module. How do we do that? Very easy. So let's go to app module with command and P or ctrl P. We go to app dot module. Now look at this NG module decorator. Here we have a property called import that is set to an array. In this array we have browser module. Browser module is one of the built in modules of Angular, and it brings some features that almost every browser application needs. Now here we need to add another module, the forms module. So on the top. So, look, this is where. We are importing the browser module. Let's import forms module from Angular slash forms. So that's defined in this library. And then. We can import it into our main app module. So forms module save back in the browser. All right, we've got me at example.com and then I'm going to change this to Domain.com enter. Look at the console. And here's me at Domain.com. Beautiful.

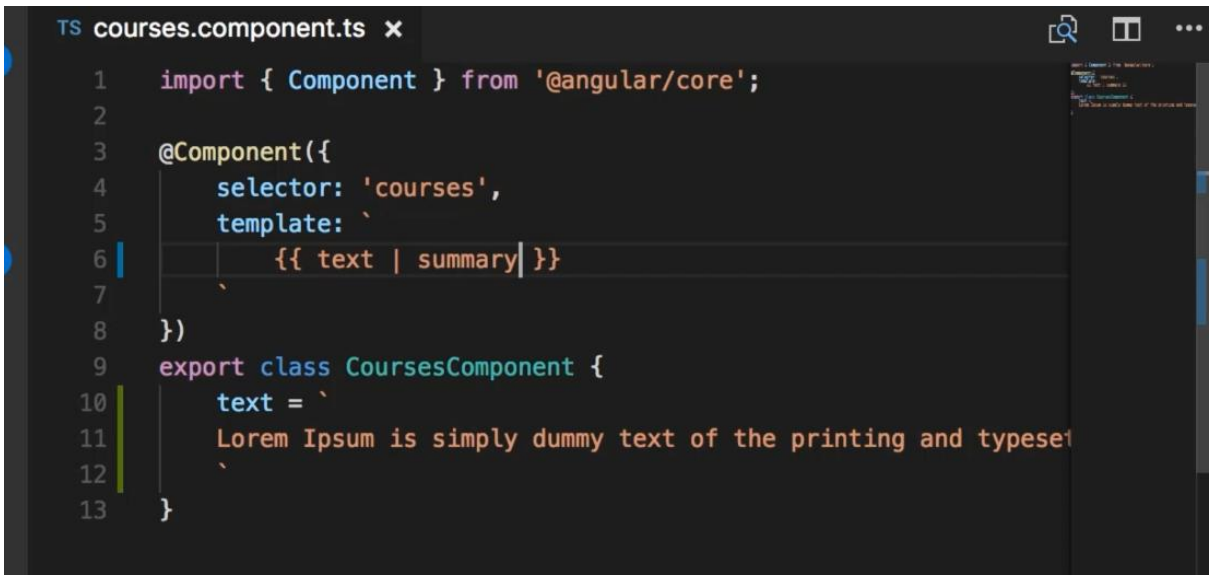
16. Pipes

Another building block in Angular is pipe. We use pipes to format data. For example, we have a bunch of built in pipes like uppercase, lowercase, decimal currency and percent. We can also create custom pipes, which I'm going to show you in the next lecture. So let's see these built in pipes in action. So here in this component, I have defined a course object with these properties title rating students price and release date. And in the template, I'm simply rendering these properties on The View. Now I want to show you how we can use pipes to format the data. So starting from the title, let's say we want to display this as uppercase. First we apply the pipe operator, which is a vertical line. And then the name of the pipe uppercase. Back in the browser. Okay, look, here's our title displayed in uppercase. We can also chain multiple pipes here so I can get the result of this expression and flow it through another pipe. In this case, lowercase. And guess what? The title is going to be lowercase. Okay, So pretty simple. Now look at the number of students. Here we have five digits. But to make it more readable, we can apply the decimal pipe to separate every three digits using a comma. So here I'm going to apply the decimal pipe. Now the keyword for this decimal pipe is number. Even though the actual class in Angular is called decimal pipe. Now let's have a look at the result. Now look at the number of students. Look, we have a comma here, which makes it more readable. Next is the rating. Now, with this number or decimal pipe, we have control over the number of integer digits as well as the number of digits after the decimal point. So here we have applied the number pipe or the decimal pipe to the rating property. Now to have more control over the number of integer and decimal point digits, we supply an argument here. So colon and here we put the argument in a string. Now the first number we supply here is the number of integer digits. So let's say one. Now, dot. And then we supply the minimum and maximum number of digits after the decimal point. So our rating look, it has four digits after the decimal point. If I want to display only

two digits after the decimal point, I would add 2-2. So two as the minimum and also as the maximum number of digits after the decimal point. Save. All right, look, we've got 4.97. Now, look what happens if I change this to 1-1. So I want to have only one digit after the decimal point. Look at what we got. So our rating is now rounded 5.0. Also, if I change the number of integer digits from 1 to 2. Look at what happens. Now we have a leading zero here. So these are the kinds of things you can do with the decimal pipe. Next we have the price. We can use the currency pipe to format this as a price. So here is the price property. Apply the pipe operator and type currency. Save. Now look, by default, we get US dollars. Now I can change this to, let's say, Australian dollars. So once again, we supply an argument. Colon. Put a string here and the name of the currency is Australian dollars aud save. And here we get Aussie dollars. Now some pipes, like the currency pipe, can take multiple arguments. So here I can supply another argument once again. Colon True to indicate that I want to display the currency symbol. Let's look at the result. So instead of aud we get a dollar sign. And finally, optionally, I can supply the third argument. So once again, colon string. And here we supply a string similar to the string we supplied to the decimal pipe. So I want to have minimum three integer digits. With two digits after the decimal point. Now, this is not going to make any difference here, but I want you to be aware of all the arguments that we can supply to the currency pipe. And finally, we've got release date which is rendered like this. Now, this is not very readable for humans, so let's apply the date pipe. Once again. Pipe Operator Date. Now we want to supply a format. So colon string. Let's format it as a short date. Save back to the browser. This is what we get. Now, if you want to see all the formats available to you, head over to angular.io and in the search box, search for date pipe. So here's the date pipe class. Note that this is defined in angular slash common library that you can install using NPM. Our project has this by default and is defined in this module. Common module. You haven't seen common module before. You have only seen the browser module, but when you import browser module it also brings in the common module. Common module has a bunch of artifacts that all applications, whether there are browser applications or not, require. Example of that is this date pipe. So if you scroll down on this page. Here, you can see all the formats available to you, like full date, long date, medium date and so on. And below that, there is more detail in this table. Now in Angular one, we had two pipes or two filters more accurately for sorting and filtering data. In Angular two, we don't have this, and this is not an oversight. It's for deliberate reasons because this filters or this pipes in angular terms are expensive operations. So Angular team decided to drop this filters in Angular two and onwards. Next, I'm going to show you how to create a custom pipe.



17. Custom pipe



courses.component.ts | summary.pipe.ts | app.module.ts

```ts
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
    name: 'summary'
})
export class SummaryPipe implements PipeTransform {
    transform(value: string, limit?: number) {
        if (!value)
            return null;

        let actualLimit = (limit) ? limit : 50;
        return value.substr(0, actualLimit) + '...';
    }
}
```

18. Component api

app.component.html

```html

<img [src]="imageUrl" />
<button (click)="onClick()"></button>

```

app.component.html | app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  post = {
    title: "Title",
    isFavorite: true
  }
}

```

app.component.html ● | app.component.ts

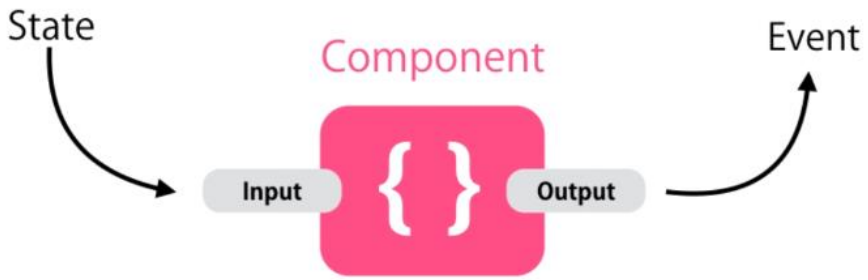
```
1
2   <img [src]="imageUrl" />
3   <button (click)="onClick()"></button>
4
5   <favorite [isFavorite]="post.isFavorite"></favorite>
6
```

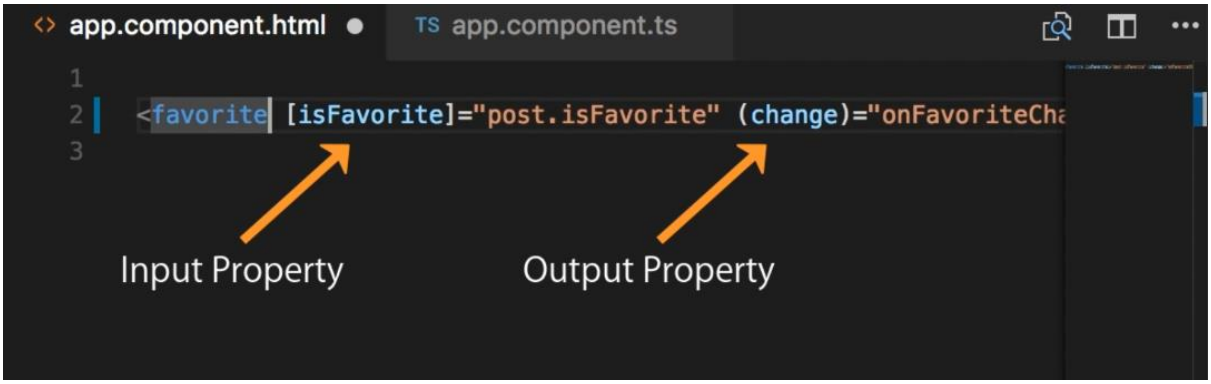
Developer Tools - http://localhost:4200/

Elements | Console | Sources | Network | Performance | Memory | 2

top ▼ | Filter | Info ▼

```
▶Unhandled Promise rejection: Template parse errors:                    zone.js:642
Can't bind to 'isFavorite' since it isn't a known property of 'favorite'. ("
<favorite [ERROR ->][isFavorite]="post.isFavorite"></favorite>
"): ng:///AppModule/AppComponent.html@1:10 ; Zone: <root> ; Task: Promise.then ; Value:
Error: Template parse errors:
Can't bind to 'isFavorite' since it isn't a known property of 'favorite'. ("
<favorite [ERROR ->][isFavorite]="post.isFavorite"></favorite>
"): ng:///AppModule/AppComponent.html@1:10
    at syntaxError (http://localhost:4200/vendor.bundle.js:5753:34)
    at
```

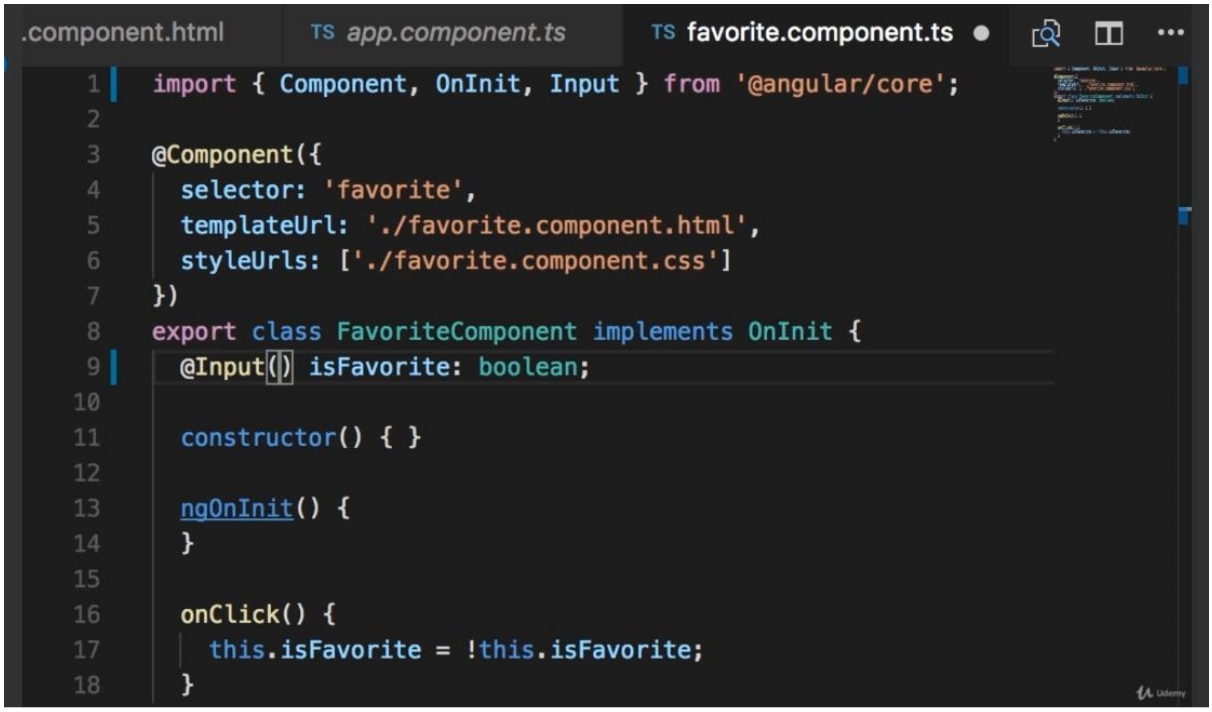




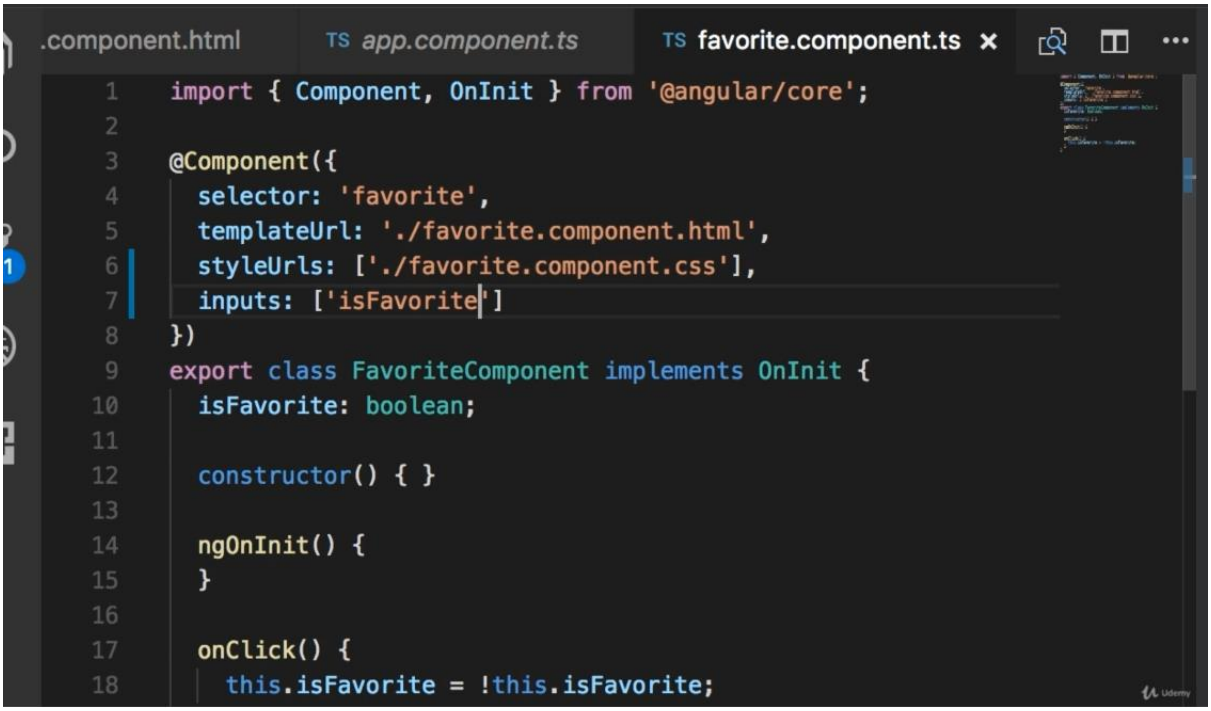
So in the last section you learned about property and event binding. We use property binding with the square bracket syntax to bind properties of Dom objects to fields or properties in our host component. That is the component that is using that Dom object in this case, this image object. Another way to think of this is that the source property is an input into this Dom object. We use this to supply data for this object to supply some state. Now, similarly, we use event binding to respond to the events raised from a Dom object, in this case the click event of a button. But this favorite component we implemented in the last section. It doesn't have any property and even binding, it's not very reusable. Ideally, I want to set the initial state of this favorite component using some object that we have in the host component. For example, here in app component, let's say we get a post object from the server. So let's go to app dot components. Okay, so here I want to define a post object. This post object has a title. And also this is favorite property. And I'm going to set this to true. Now, for now, let's not worry about the complexity of getting this object from the server. Let's just imagine we have this object in memory. Now, in the template for app component, we want to display this post and if it's set as favorite, we want to render this favorite icon as a full star. So back in the template. Here. Unfortunately, we cannot use property binding to bind. This is favorite field of the favorite component to. This post object that we have in this component Dot is favorite. This is not going to work. Let me show you. First, let me delete these two lines so we focus only on one thing. Now back in the browser. Look at this error. Can't bind to is favorite since it isn't a known property of favorite. So this property binding doesn't work even though in favorite component we have this field called is favorite. And this is a public field because in Angular templates, in order for you to use property binding, you need to define that property or that field as an input property. And that's what I'm going to show you in the next lecture. So to make this favorite component more reusable, we want to add support for property and event binding. For example, in the future we want to be notified whenever the user clicks on this favorite component. For example, we want this favorite component to raise a change event like this. And we want to be notified and call a method in the host component in this case app component. So unfavorite change. And inside this method we can call the server or do something else. Once again to add support for event binding, we need to define a special property in this favorite component that we refer to as an output property. In other words, in order to make a component more reusable, we want to add a bunch of input and output properties. We use input properties to pass input or state to a component, and we use output properties to raise events from these custom components. The combination of input and output properties for a component make up what we call the public API of that component. Public

API, which stands for Application Programming interface. So now our favorite component doesn't currently have a public API because it doesn't have any input or output properties. And in other words, we cannot use property and event binding here. So over the next few lectures I'm going to show you how to add support for property and event binding to this favorite component and make it more reusable.

19. Input properties



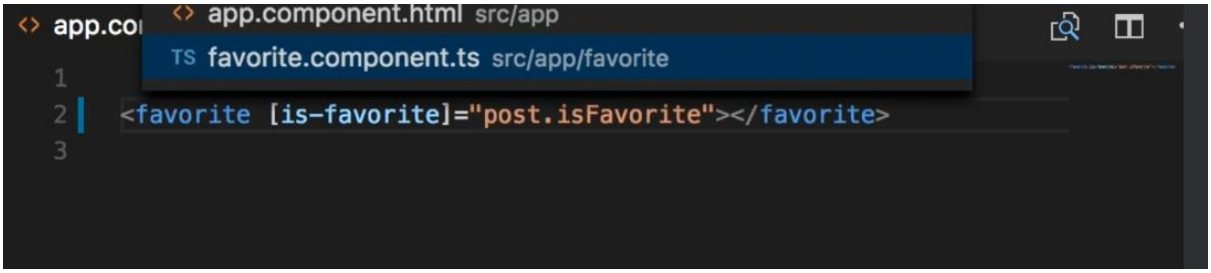
Or



So here in favorite component we want to mark this field is favorite as an input property so we can use it in property binding expressions like here. In App.component.html. Now, there are two ways to mark this field as an input property. Here's the first approach. We need to import a decorator function here on the top from Angular core library. This is called input. And then we can use this to annotate this field. So add sign input and we call it like a function exactly like how we use this component decorator here. Now, input is another decorator in Angular for marking fields and properties as input properties. Now this field is exposed to the outside, and in our templates we can bind it to a property somewhere else. So let's go back in the browser. You can see our star icon is initially full because an app component I set the is favorite property of the post object to true. So here in the template look we're binding is favorite to post that is favorite. So our property binding works perfectly. If I click this, it goes empty. Beautiful. Now, let me show you the second approach to mark this field as an input property. So back in the favorite component, we don't need this input decorator so we can delete it from the top. Instead, we can declare this field as an input property in the component metadata. So here in our component decorator, we can add another property input. We set this to an array of strings. In this array, we list all the fields and properties that should be input properties. So in this case is favorite. Now, you may think this approach is better because there is one less step. We don't have to import the input decorator and our code is a little bit cleaner. But actually there is a problem with this approach and I'm going to show you that in a second. Now, before going any further, let's make sure the application works up to this point. So save back in the browser. So our star icon is full click. It's empty. Beautiful. Now, what is the problem with this approach? The problem is the use of this magic string here. So this code works as long as we have a field or a property in our class called is favorite. If tomorrow I decide to rename this field for refactoring reasons, let's say I want to change this. Two is selected. So I use F2 to rename this field and you can see all the occurrences of this field are also updated. However, this magic string here is not updated, so our input property is broken. Let me show you save

back in the browser. So, look, the star is full and you may think this is actually working, but no, look what happens when I click on this icon. It doesn't go empty. What's going on? Well, when you declare a field as an input property using the component metadata, this actually creates a field under the hood like this. Is favorite boolean. And of course, in our template we can bind this to another property like here in app component, we're binding this to another property, right? So in our component, we'll end up with two fields. This is favorite as a completely separate field and it's got nothing to do with is selected. Note that here in onClick method we have used is selected field. Not is favorite. That's why when we click this icon, nothing happens. So I personally think this is really redundant. I don't like that Angular team have multiple ways to do things in Angular. It just makes the framework harder to learn and confuses new learners. Plus, you can see the issue with this magic string here. So I hope in the future versions of Angular, they just drop this and make component metadata simpler. So back to the first approach. I'm going to import the input decorator and let's delete this. Apply at input here. So this is how we define input properties.

20. Aliasing input properties



So in the last lecture you learned how to define an input property. In this lecture, I'm going to show you how to give this input property an alias or a nickname. Here is a use case. In this component we have defined our is favorite field using camel casing notation because that's what we use in JavaScript or in TypeScript. So the first letter of the first word is lowercase, and the first letter of every word after is uppercase. Now, maybe in your applications you don't want to use camel casing notation in your HTML markup. So back in our app component, let's say in our property binding, we want to use something like this is dash favorite. Now in JavaScript or in TypeScript, we cannot have a field with that name. That's not an acceptable identifier. So the solution is to use an alias or a nickname for an input property. So back in our favorite component. Here. When using the input decorator, we can optionally supply a string to set an alias for this property. So I'm going to use is dash favorite as simple as that. Now back in the browser. Our favorite component is initially filled. And when we click it, it toggles beautiful. Now using an alias also has another benefit. It keeps the contract of this API stable. Let me show you what I mean by that. So I'm going to remove this. Let's go back to our app component. And revert this back to is favorite like this. Okay now back in the component. Let's say tomorrow, for refactoring reasons, we decide to change the name of this field. So we press F2 and change this to is selected. Now you can see all the occurrences are updated here in onClick. Okay. But our application is now broken because app component is expecting an input property called is favorite. So when we go to the browser we see the same error can't bind to is favorite. Now to minimize the impact of these changes, we can use an alias to keep the contract of a component stable. So back in our favorite component. I can apply an alias here is favorite. Okay, now this magic string is going to stay here. If I rename this field in the future, our magic string is not going to be impacted. So app component will continue to use this component exactly like before. Let's try it. So we don't have any errors in the browser. And here's our favorite icon. But note that it's initially empty, whereas we expect it to be full. Why? Let's go to the template for our favorite component. So favorite dot component dot HTML. Note that here we're still using the is favorite field so we rename this field to is selected, but we forgot to come back here and make this change. So if I quickly rename this field here to is selected. Now our component is working. What I want to point out here is that, yes, when you rename fields, your templates are not updated, so you need to manually make these changes. However. Because we use an alias on this input property. We don't have to go and change another 100 places in our code where we have used this component like in our app component. Also, if you're building a reusable component that we want to distribute, we don't want to tell all our clients, all the consumers of our components that, Hey, now this field is renamed from is favorite to is selected. So we are minimizing the impact of these changes. We can use rename refactoring in vs code. So by pressing F2 we can change this field and all the occurrences in our TypeScript code. But then we have to manually go and change the template as well, which is one extra step. Okay. So here is the lesson. If you're building reusable components, give your input properties an alias to keep the contract of your components stable. Next, we're going to look at output properties.

21. Output properties

```
pp.component.html   TS app.component.ts   TS favorite.component.ts ×

8   export class FavoriteComponent implements OnInit {
9     @Input('isFavorite') isSelected: boolean;
10    @Output() change = new EventEmitter();
11
12    constructor() { }
13
14    ngOnInit() {
15    }
16
17    onClick() {
18      this.isSelected = !this.isSelected;
19      this.change.emit();
20    }
21  }
22
```

22. Passing event data

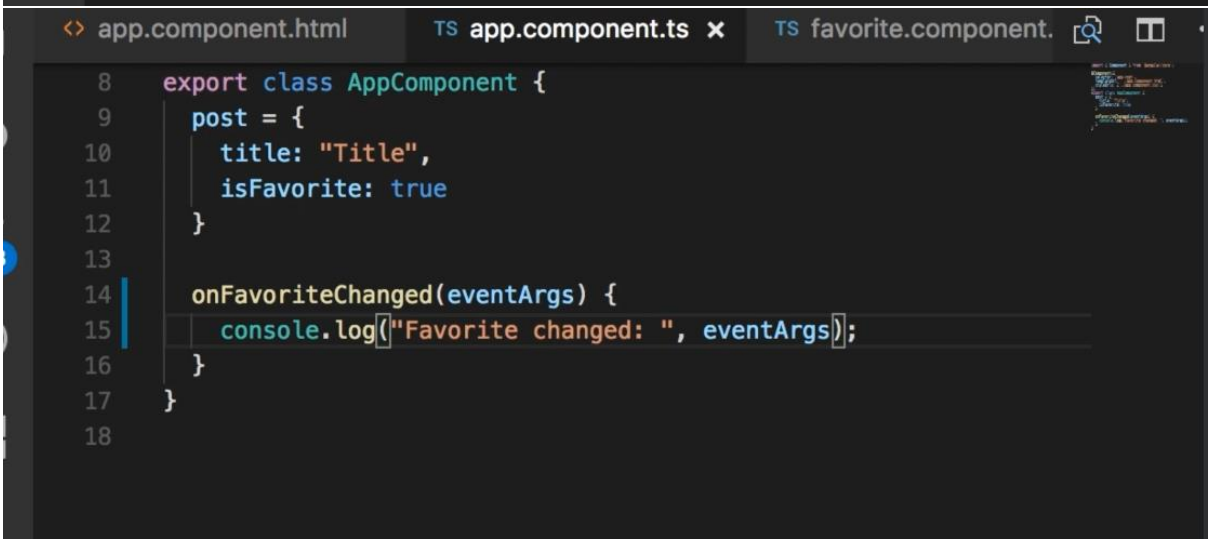
```
pp.component.html   TS app.component.ts ×   TS favorite.component.ts

8   export class AppComponent {
9     post = {
10      title: "Title",
11      isFavorite: true
12    }
13
14    onFavoriteChanged(isFavorite) {
15      console.log("Favorite changed: ", isFavorite);
16    }
17  }
18
```

```
  styleUrls: ['./favorite.component.css']
})
export class FavoriteComponent implements OnInit {
  @Input('isFavorite') isSelected: boolean;
  @Output() change = new EventEmitter();

  constructor() { }

  ngOnInit() {
  }

  onClick() {
    this.isSelected = !this.isSelected;
    this.change.emit(this.isSelected);
  }
}
```

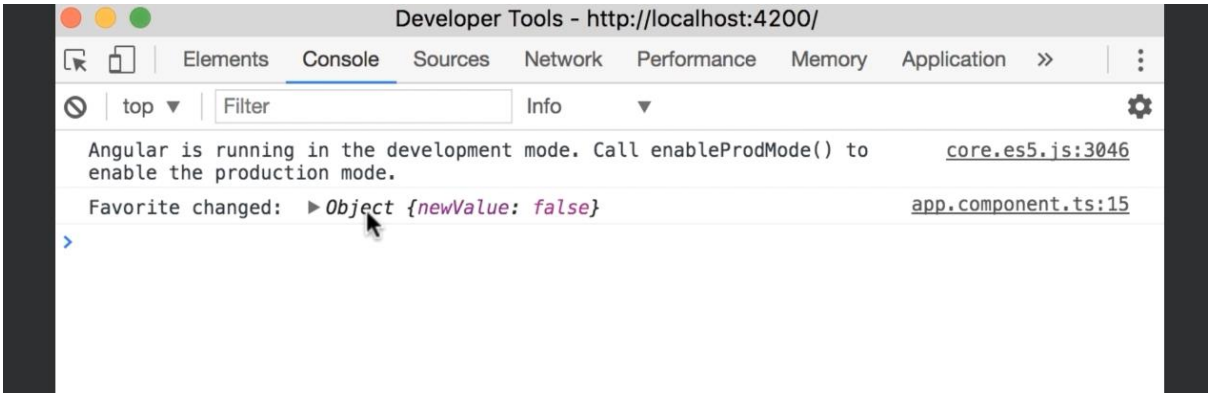
So in our app component here in this event, handler on favorite changed. Currently we're displaying only a simple message in the console. We don't know anything about this event. We don't know if the user has marked an object as favorite or not. So we need to change our implementation and pass some data when raising this event. So back in our favorite component here, when we are emitting this event, we can optionally pass some value and this value will be available to all subscribers of this event. In this case, the subscriber of the Change event is our app component because that's where you're handling this event. That's where we are getting notified when the state of the favorite component changes. Okay. So back in the favorite component, we can pass an argument to this emit method. In this case, I'm going to pass this dot is selected. That's the new state of this favorite component. Okay, So this is a simple boolean, and in our event handler, we're going to get a simple boolean value. So save. Back in app component. Okay, I'm going to add a parameter. Let's call This is favorite. Okay. And then I'm going to simply display this in the console is favorite. And finally, the last step. Let's go to our App.component.html. So here when we're handling the change event, we want to pass a dollar event object. To our event handler. You have seen this before when handling the click event of buttons. We use this dollar event object, which is a built in object in Angular. Okay. Now, previously, when handling the click event of

buttons, this dollar event represented a standard Dom event object. Here, because we're dealing with a custom component, this dollar event could be anything that we pass when raising an event. In this case, it's going to be a simple boolean value. Let's try this. So back in the browser, I'm going to click this icon. Now look in the console. His favorite is false. Okay. Now I'm going to click one more time. And this time his favorite is true. So this is how we pass data along with our events. Now back in the favorite component. Here we're dealing with a very simple component. Maybe in your applications, you're building something more complex. So instead of a simple Boolean value, you may want to consider passing an object. So let's change this implementation and instead pass an object. This object can have a property like new value. And we set this to this. That is selected. Okay. Now back in the template for app component. Here are our event object now represents an actual JavaScript object that has a property called New value. And also in our app component, which is the subscriber of the change event of our favorite component, instead of a simple Boolean value, we're going to receive an actual object. We can call this event args event arguments or the arguments passed with the event. And then. Displayed here. Now let's try this. So back in the browser, I'm going to click this icon. And then in the console, look at the event data. We have an object with a property called new value, which is set to false. Now, one last tip before we finish this lecture. If you're dealing with a complex implementation, perhaps you may want to apply a type annotation here so you get compile time checking as well as IntelliSense. In this method. Now, one way to do this is to use inline annotation so we can say this event args is an object that has a property called new value. Which is a Boolean. It's a little bit messy. It's a little bit verbose and noisy. Potentially we can define an interface somewhere else that defines the type of arguments of this change event. For example. Here I can define an interface called favorite changed event args. So this interface represents the shape of the object that is passed along with the change event of our favorite component in this object we're going to have. New value property, which is a boolean. And then. Instead of this inline annotation, we can use favorite change event arcs. And with this we get compile time checking and IntelliSense, that's better. Now, if you're building a reusable component, you want to declare this interface in your implementation and export it from your module. And then any consumers would import this from your module. So instead of declaring this interface here. Ideally, we should add it. Here after our component, we're going to export this interface. So export. Like this. And then in app component. We need to import this type from the other module. So with auto import plugin, I can simply put my cursor here and press command and period. And import this type on the top. So this is all about passing event data.

```
styleUrls: ['./favorite.component.css']
})
export class FavoriteComponent implements OnInit {
  @Input('isFavorite') isSelected: boolean;
  @Output() change = new EventEmitter();

  constructor() { }

  ngOnInit() {
  }

  onClick() {
    this.isSelected = !this.isSelected;
    this.change.emit({ newValue: this.isSelected });
  }
}
```



```
export class AppComponent {
  post = {
    title: "Title",
    isFavorite: true
  }

  onFavoriteChanged(eventArgs) {
    console.log("Favorite changed: ", eventArgs);
  }
}
```



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

interface FavoriteChangedEventArgs {
  newValue: boolean
}

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  post = {
    title: "Title",
    isFavorite: true
  }

  onFavoriteChanged(eventArgs: { newValue: boolean }) {
```

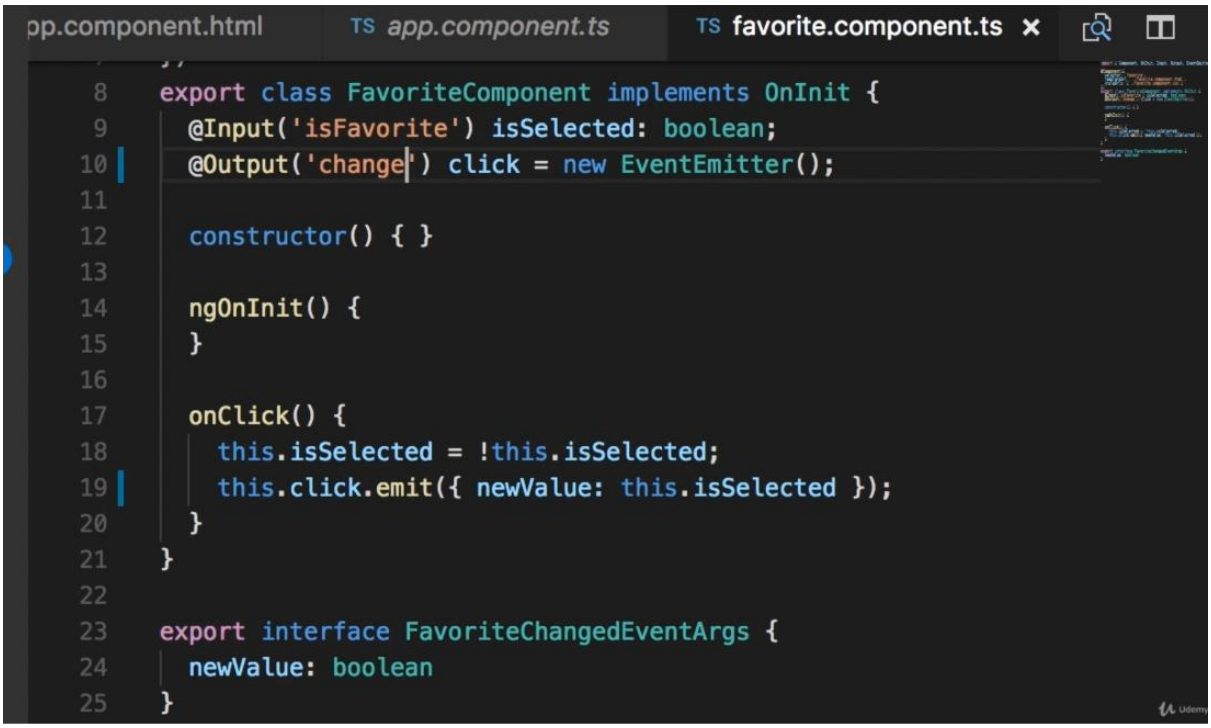
app.component.html | app.component.ts | favorite.component.

```
  post = {
    title: "Title",
    isFavorite: true
  }

  onFavoriteChanged(eventArgs: FavoriteChangedEventArgs) {
    console.log("Favorite changed: ", eventArgs);
  }
}
```

23. Aliasing output

So earlier you learned how we can use an alias to keep the contract of a component stable. Previously, we used an alias on an input property, but we can also use an alias on an output property. So here. If tomorrow I decide to change the name of this event from change to something else, my application is going to break. Let's try this. So I'm going to rename this to click. Now back in App.component.html, you can see this component, which is the client or the consumer of our favorite component is expecting a change event, but that doesn't exist anymore. Now back in the browser, I'm going to click this. So our component is working and you may think nothing is broken, but if you look at the console. We no longer

have our log messages. So I'm talking about this message that we log in on favorite changed method of app component. In other words. In App.component.html. We're handling the change event, but there is no event by that name is because our app component is expecting this event, but it's never happening. So it's assuming maybe sometime in the future this event is going to get raised. That's why it doesn't give us an error in the browser console. But our code is broken. This event handler is not called. So back in our favorite component, we can apply an alias here to make sure that if in the future we rename this field, the client of this component are not going to break. So. Let's call this change. Save. Now back in the browser, I'm going to click this. Now look at the console. We get all these messages that are coming from our event handler. So once again, you can use an alias on the input and output properties of your



24. View encapsulation

favorite.component.ts ● | favorite.component.css | styles.css

```ts
import { Component, OnInit, Input, Output, EventEmitter, ViewEnca

@Component({
  selector: 'favorite',
  templateUrl: './favorite.component.html',
  styleUrls: ['./favorite.component.css'],
  encapsulation: ViewEncapsulation.Emulated

})
export class FavoriteComponent {
  @Input('isFavorite') isSelected: boolean;
  @Output('change') click = new EventEmitter();

  onClick() {
    this.isSelected = !this.isSelected;
    this.click.emit({ newValue: this.isSelected });
  }
}
```

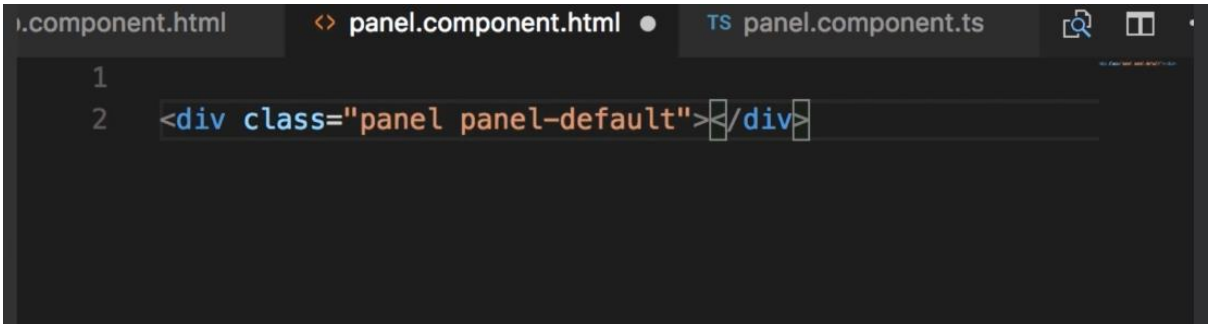
25. Ng content

.component.html | panel.component.html | panel.component.ts ×

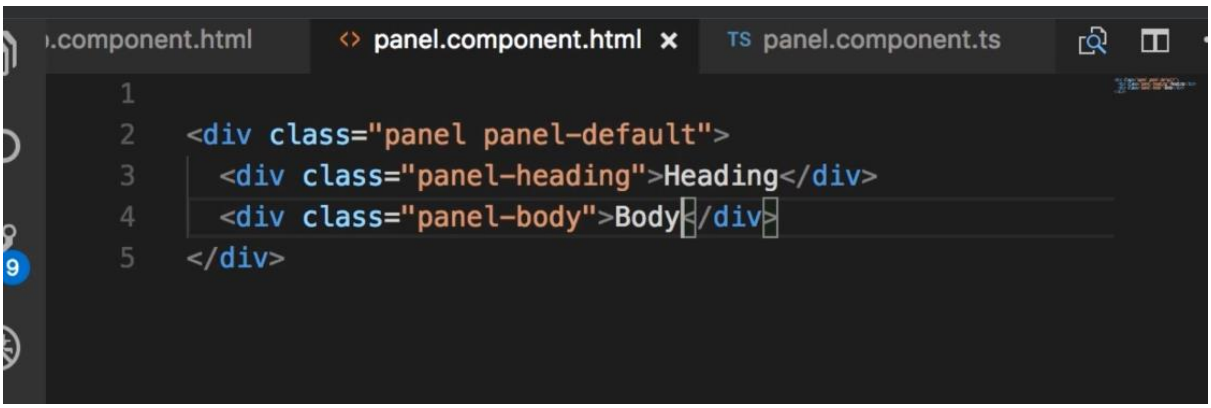
```ts
import { Component } from '@angular/core';

@Component({
  selector: 'bootstrap-panel',
  templateUrl: './panel.component.html',
  styleUrls: ['./panel.component.css']
})
export class PanelComponent {
  constructor() { }
}

```

.component.html | panel.component.html ● | panel.component.ts

```
1
2  div.panel.panel-default
```


.component.html
panel.component.html
panel.component.ts


.component.html
panel.component.html
panel.component.ts
iv.panel.panel-default>div.panel-heading+div.panel-body


.component.html
panel.component.html
panel.component.ts

Heading
Body


app.component.html
panel.component.html
panel.componer

app.component.html ● panel.component.html panel.componer

```
1
2  <bootstrap-panel [body]="body"></bootstrap-panel>
```

opers

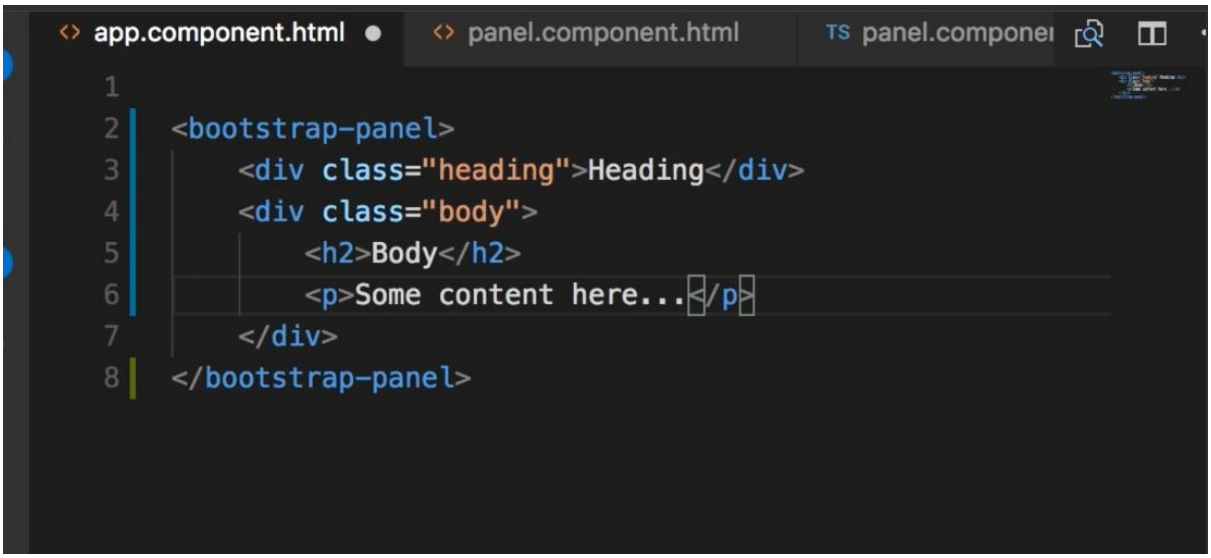
app.component.html ● panel.component.html panel.componer

```
1
2  <bootstrap-panel></bootstrap-panel>
```

panel.component.html ●

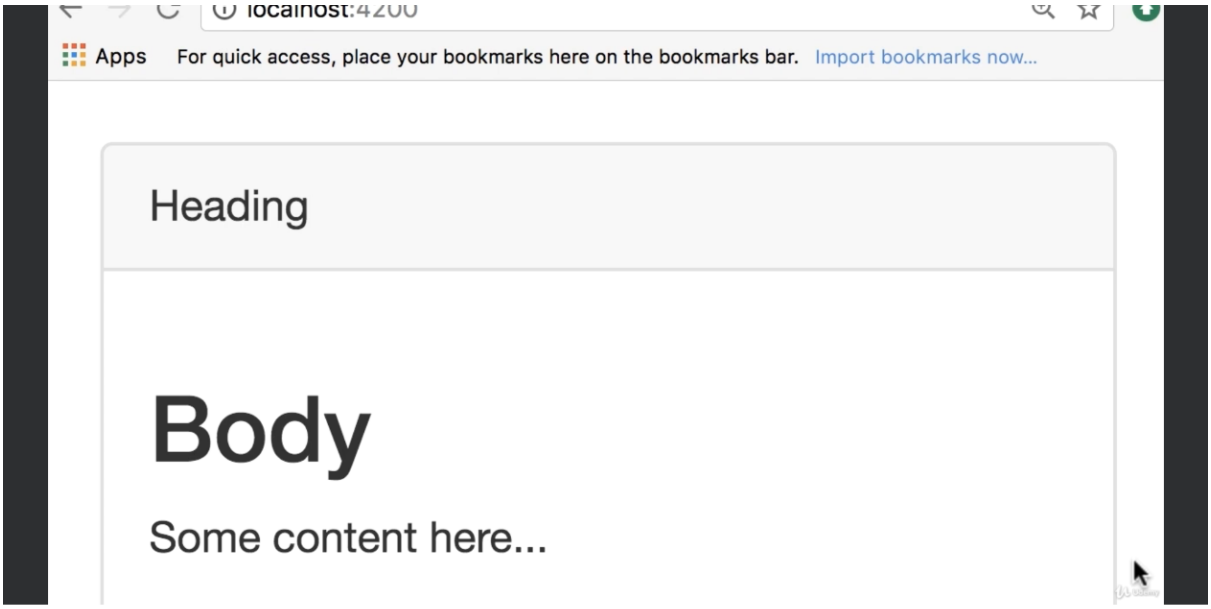
```
1
2  <div class="panel panel-default">
3    <div class="panel-heading">
4      <ng-content select=".heading"></ng-content>
5    </div>
6    <div class="panel-body">
7      <ng-content select=".body"></ng-content>
8    </div>
9  </div>
```


app.component.html
panel.component.html
panel.componer
<bootstrap-panel>
Heading
Body
</bootstrap-panel>


app.component.html
panel.component.html
panel.componer
<bootstrap-panel>
Heading

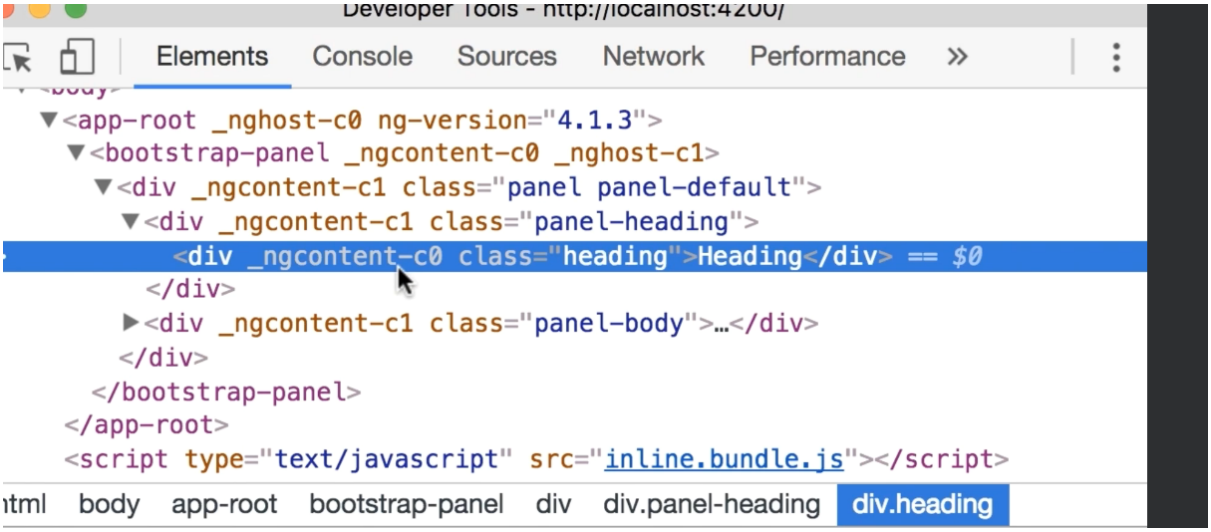
<h2>Body</h2>
Some content here...

</bootstrap-panel>


localhost:4200
Apps
For quick access, place your bookmarks here on the bookmarks bar.
Import bookmarks now...
Heading
Body
Some content here...

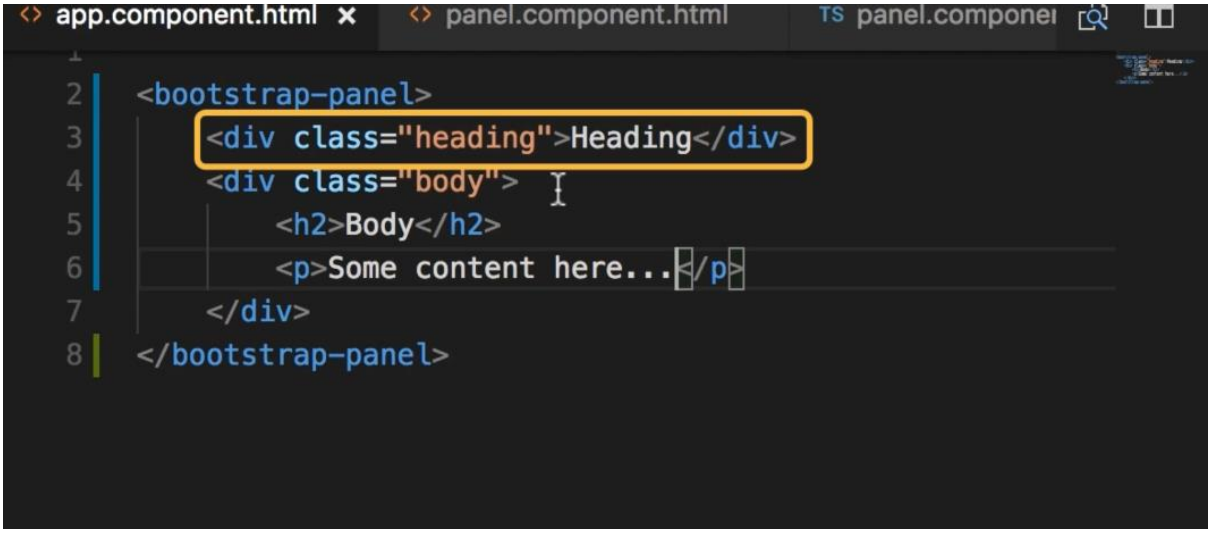
Imagine you want to build a bootstrap panel component. So I've created a new component called panel component. You can see the selector for this component is bootstrap panel. So here I have used bootstrap to prefix this selector because we don't want to use panel because chances are this name is going to clash with another component. So as a best practice, if you're building reusable components, always prefix them. So here, bootstrap panel. Now let's go to the template for this panel. In order to render a bootstrap panel when it a div with two classes. So dot panel and dot panel dash default. So if you press tab here, we get this markup right now let's undo this. Want to add more markup here so we generate it in one go. Now inside this div we want to have another div. With the class panel dash heading and next to this div we want to have another div with the class panel dash body. Now, tab. This is our markup. So let's just put some label here. Heading and body. Now back in the browser. This is a bootstrap panel. It's a nice user interface component. Now let's make this dynamic so I don't want to hard code this heading and body labels here. I want the consumer of this panel component to be able to inject text or markup into this component. So let's go to App.component.html. This is where we have used the bootstrap panel. Now, in order to set the heading and the body, one way is to use property binding so we can define input properties. Like heading and body. And set them to the body property of App.component. But this syntax is a little bit weird here because I don't want to go in app.component define a property called body and then add my markup. I want to be able to write my markup right here. So let me show you a better way. Instead of using property binding with input properties, we're going to use the content element. So let's go to the template for panel component. So here we want to add two injection points so the consumer of this panel component can provide content into those injection points. So here inside panel dash heading, I'm going to add an element called NG Dash content. This is a custom element defined in Angular. And similarly, I'm going to add this inside panel dash body because I want the consumer of this component to be able to provide content that should be placed here at runtime. Now here we have two NG content elements. Somehow we need to distinguish them. We need to give them some kind of identifier. So we use the select attribute, select and set this to a CSS selector. We can reference a class, an ID or an element. So here I'm going to use a CSS class like dot heading. So if in the consumer of this panel component, we have an element that matches this selector, which means an element with the heading class, that element is going to be placed right here instead of NG content. In other words, NG content is going to be replaced with that element. Okay. Now, similarly, I'm going to set the select attribute here. And set this to dot body. Now let's go back to our app component. Here inside the bootstrap panel element. I'm going to add two divs, one with the class heading, and next to that I'm going to add another div with the class body. Okay, now let's set our custom content. So here is the heading. And here's the body. Or we can add complex HTML markup here. So we can have, let's say, an H2. Here's the body below that. We're going to have a paragraph. With some content here. Okay, Now let's preview this in the browser. So look this way we can provide custom content to our reusable components. So if you're building reusable components, like in this case, a bootstrap panel and you want the consumer of this component to be able to provide custom content, use the content element.

26. Ng container


Developer Tools - http://localhost:4200/
Elements
Console
Sources
Network
Performance
<app-root _nghost-c0 ng-version="4.1.3">
<bootstrap-panel _ngcontent-c0 _nghost-c1>
<div _ngcontent-c1 class="panel panel-default">
<div _ngcontent-c1 class="panel-heading">
<div _ngcontent-c0 class="heading">Heading == $0

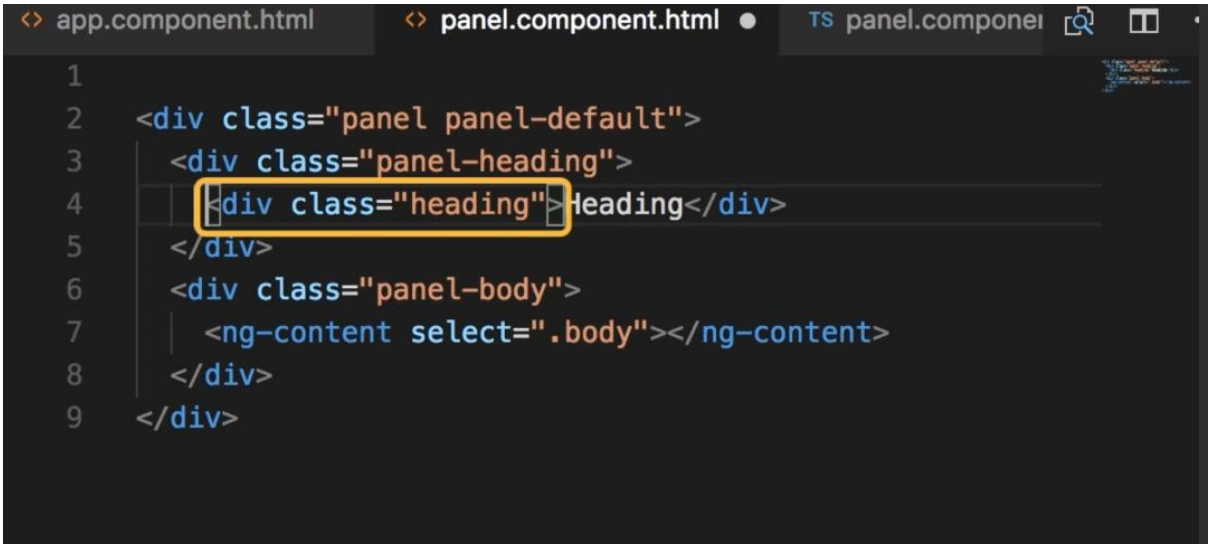
<div _ngcontent-c1 class="panel-body">…

</bootstrap-panel>
</app-root>
<script type="text/javascript" src="inline.bundle.js"></script>
html
body
app-root
bootstrap-panel
div
div.panel-heading
div.heading


app.component.html
panel.component.html
<bootstrap-panel>
Heading

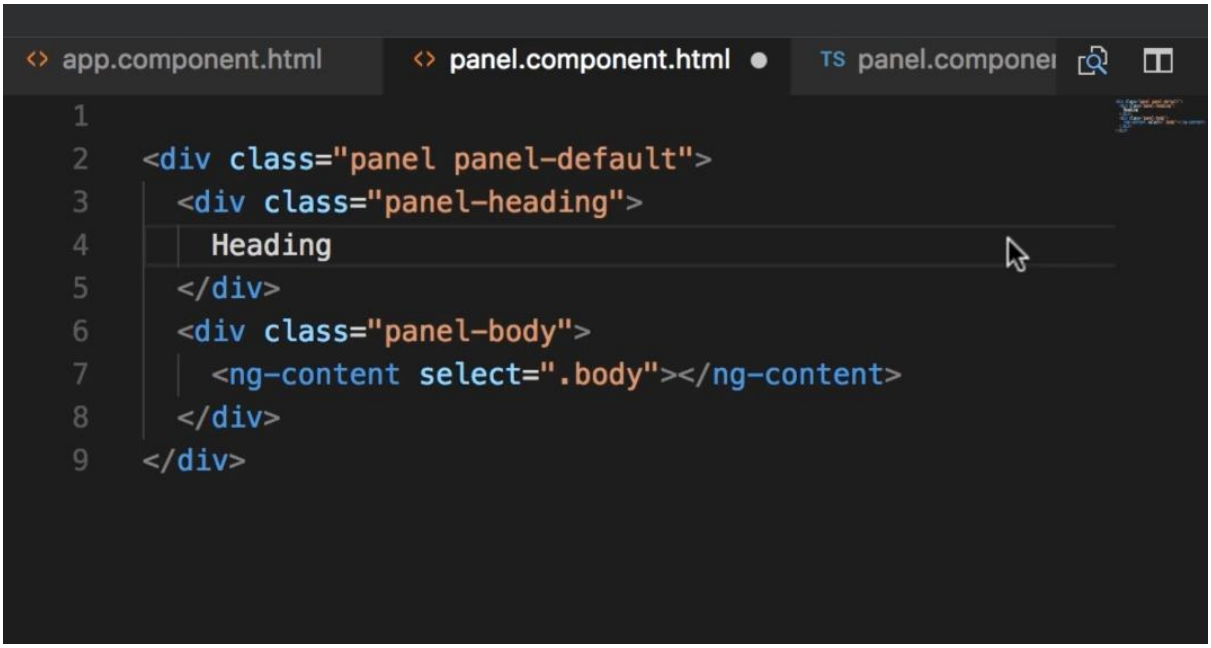
<h2>Body</h2>
Some content here...

</bootstrap-panel>


app.component.html
panel.component.html

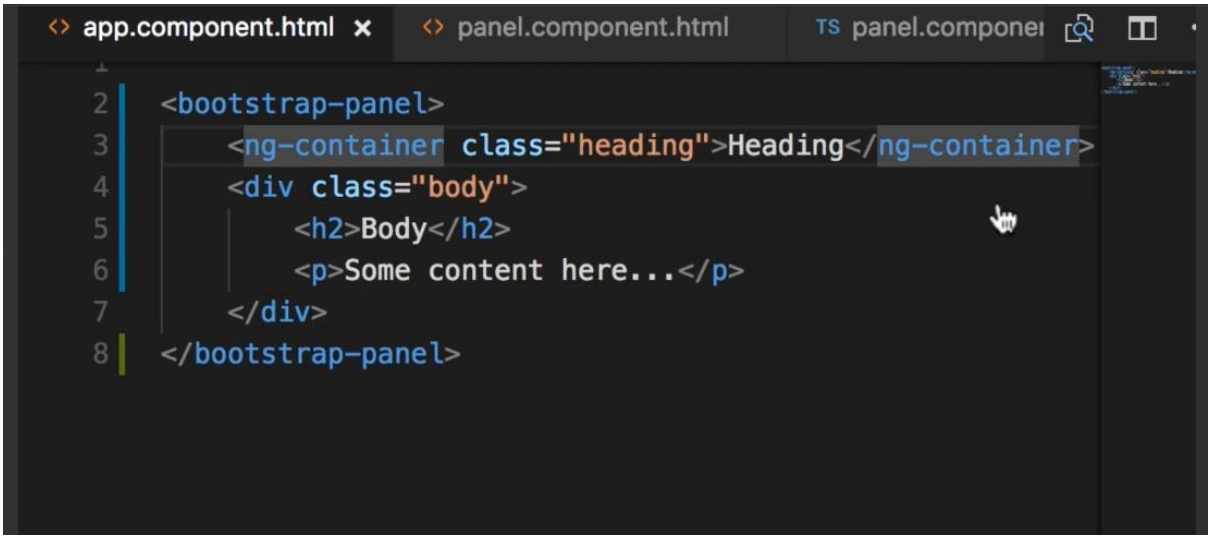
Heading

<ng-content select=".body"></ng-content>


app.component.html
panel.component.html
panel.componer

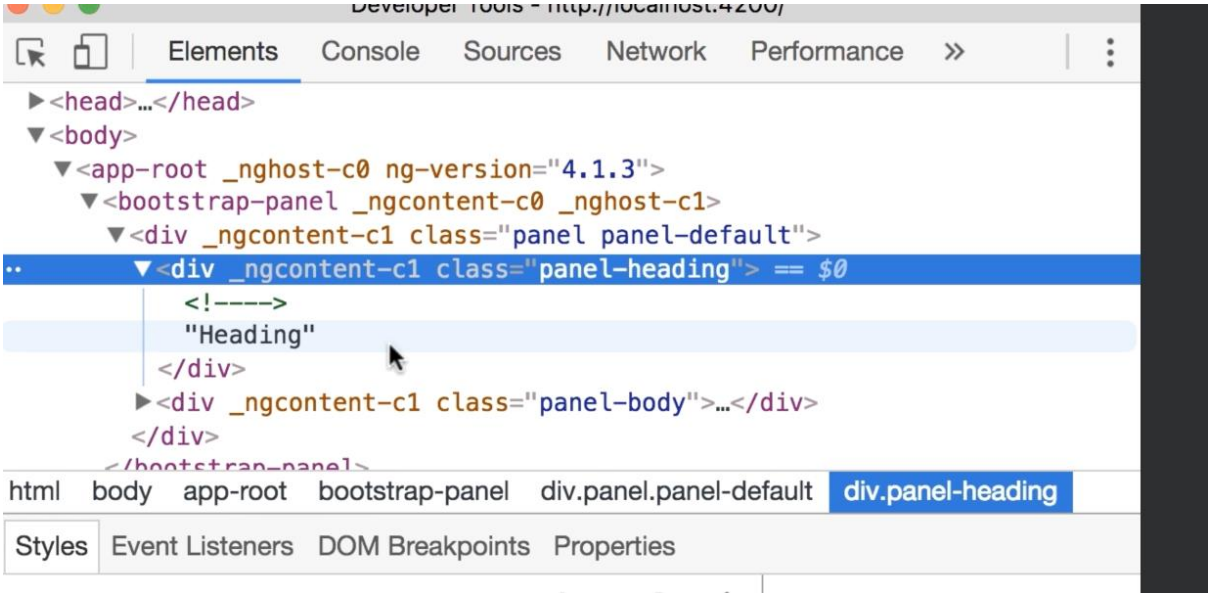
Heading

<ng-content select=".body"></ng-content>


app.component.html
panel.component.html
panel.componer
<bootstrap-panel>
<ng-container class="heading">Heading</ng-container>

<h2>Body</h2>
Some content here...

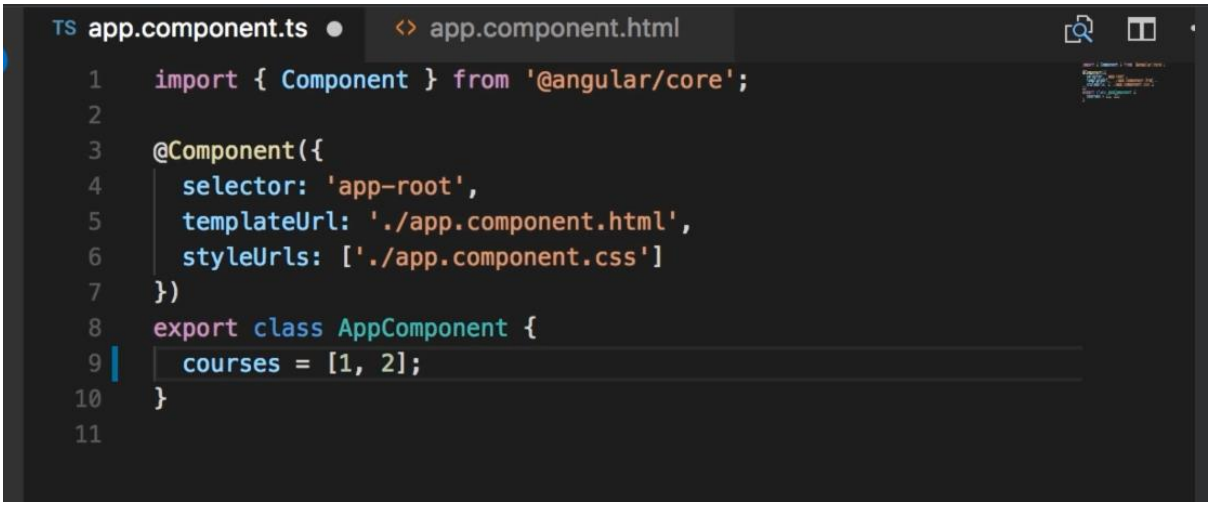
</bootstrap-panel>



Falling from our last example, let's inspect this heading element. So, look. This is our panel that's heading Element and inside that, we have this debate with the class heading and the actual heading label. So this DiBiase here is what we put in AB component. Let me show you one more time. So hearing app component template inside bootstrap this panel element, we have this div with a class heading now we use this class as a selector for the energy content element in panel component. Let's have a look. So here's a panel component. And here we have an energy content with this selector dot heading. So with this implementation, when Angular finds an element with the selector. In this case, DOT heading, it's going to replace this energy content with that element. And that element, as you saw in AB component, is a div with the class heading. So I'm going to copy this. I can panel component at runtime, this energy content is going to be replaced entirely with this device. Now, this markup worked perfectly fine. We had a beautiful bootstrap panel, but sometimes we don't want this additional div here. It's just extra noise in the markup. It will be nicer to have just this heading here. Right. So let me show you how you can achieve this. Now back to this energy content. OK, let's go to our app component. So I'm going to replace this div with engy dash container. This is another custom element in angular, so at runtime, Angular is going to take only the content of this engie container. It's not going to render this energy container element in the dump. It's not going to be a Dave or something else. Let's look at the result. So save back in the browser. Let's inspect this heading one more time. So here's our panel that's heading, which is in our bootstrap panel component, let's inspect this and now we can see we have the actual heading without any additional markup around it. So if you want to render something without putting it inside a div or another kind of HTML element, using that container element in angular. Have limited time to learn Angular 4 (Angular 2+)? Take this course and learn Angular in 10 hours! Rating: 4.5 out of 5 4.5 18,299 ratings 77,454 Students 10.5 hours

27. Ng if

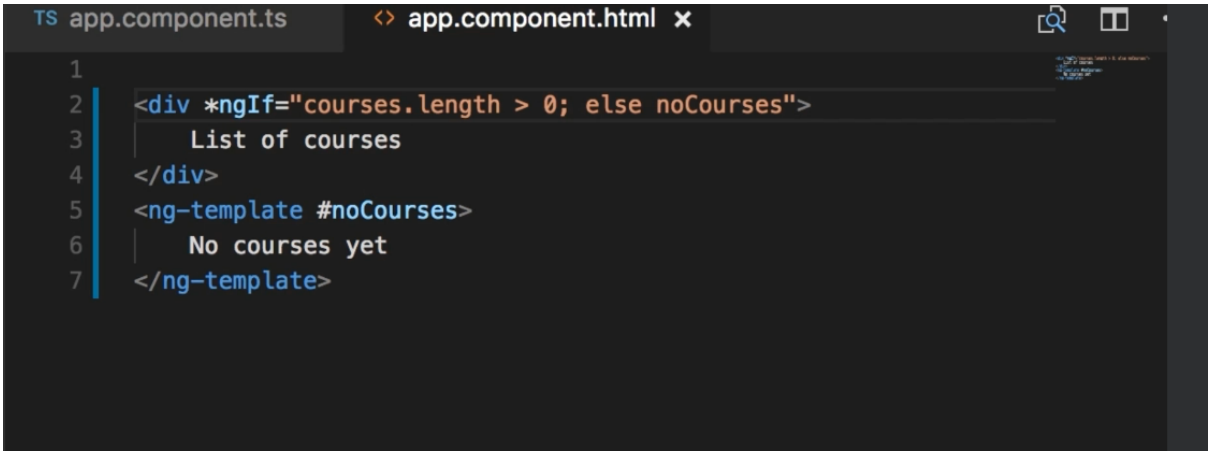
There are times that you may want to show or hide part of a page depending on some condition. So here in app component, I'm going to define a field courses, set it to an array with two items. Now for now, we're just going to use simple numbers. In a real world application, you're going to have the actual course objects.



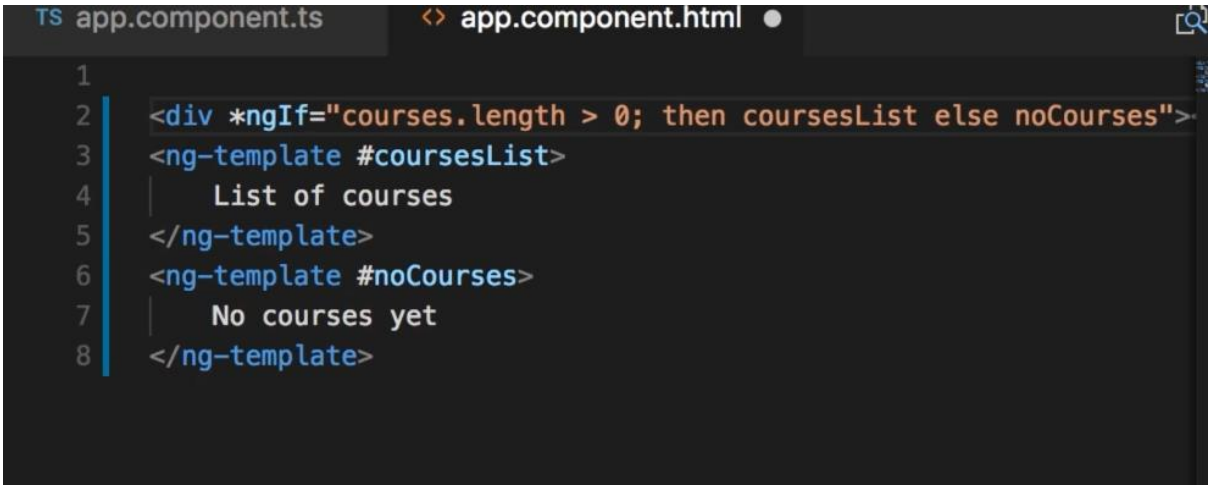
Now in our App.component.html file. If you have any courses in this area, we want to render them in a list. Otherwise, we want to display a message to the user like there are no courses yet. So first I'm going to create a div. This is where we're going to display the list of courses. Now, here again, I'm simplifying this example. If you want to render the list of courses, you would use a Ul and Li with the NG for directive. So here's one div for displaying the list of courses and here's another for displaying a message like no courses yet. We want to render one div or the other, depending on the number of items in our courses array. For that we use the If directive. So to refresh your memory, we use directives to modify the Dom. There are two types of directives. Structural directives modify the structure of the Dom by adding or removing Dom elements. Attribute directives Modify the attributes of Dom elements. So here we want to change or modify the structure of our Dom. We want to add or remove one dom element. So whenever you're using structural directives, you need to prefix them with an asterisk. And I'm going to explain the reason later in this section. So asterisk ng if. Now here we add the condition. If this condition evaluates to true, see this div will be rendered in the Dom. So we can add something like courses that length greater than zero. Now, you don't necessarily have to write the condition here. You may have a method in your class like this. Do we have any courses? And we call this method, we get some value. If that value evaluates to true, then this div will be rendered. Now let's revert this back. Now we can use the same technique with the other div and this is the old approach before Angular four. So let's see how that works. Now if I want to render this div if we don't have any courses, so courses dot length. Equals zero. Now back in the browser. So because we have two items in our array, we see the list of courses message.

Now back in our component. I'm going to empty this array. Save. Back in the browser. Now we get no courses yet. Message. Let's inspect this. So here under app root look, we have only one div, which is the div with the message. No courses yet. So the other div is not in the Dom and that's what I want you to pay attention to. So when you use NG if if this condition evaluates to true see this element will be added to the dom. Otherwise it will be removed from the dom.

Now in Angular four now we have a slightly different syntax to implement the exact same feature. So instead of repeating NG twice with this condition reversed, we can implement it using an if and else kind of approach. Just like how we have if and else in a lot of programming languages.

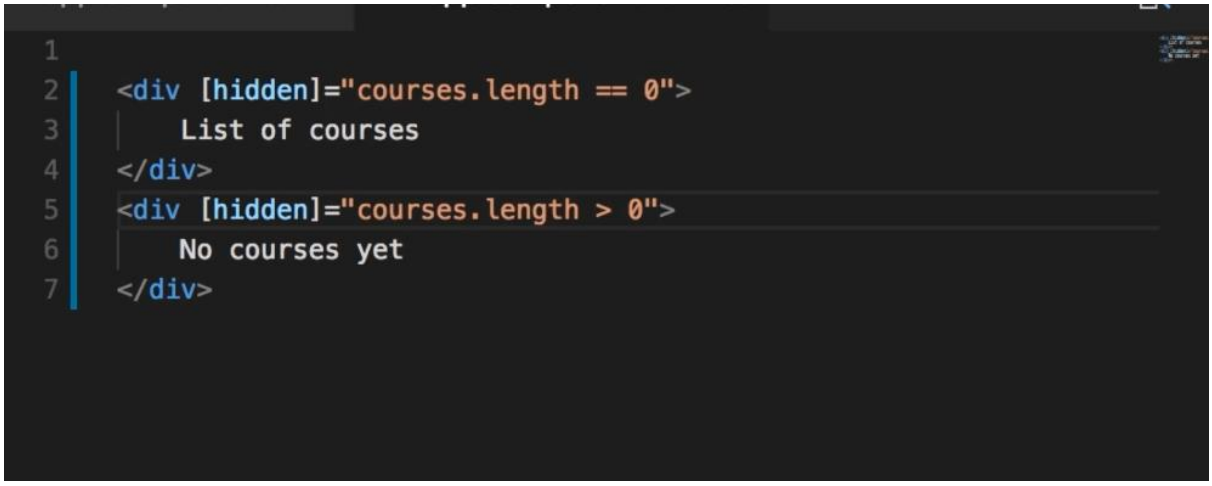


So first we need to change the type of this second element from div to NG template. So NG dash template. Then we remove this ng if make the code cleaner. But we need to assign this template an identifier. Earlier you learned that you can use this hashtag to define a template variable. Let's call this block No courses. Then we modify our NGF. So after the condition, we add a semicolon and we type LS and then the name of our template variable no courses. Note that I didn't add this hashtag here, just the name of our template variable. So if this condition is true, see this block will be rendered otherwise the other block.



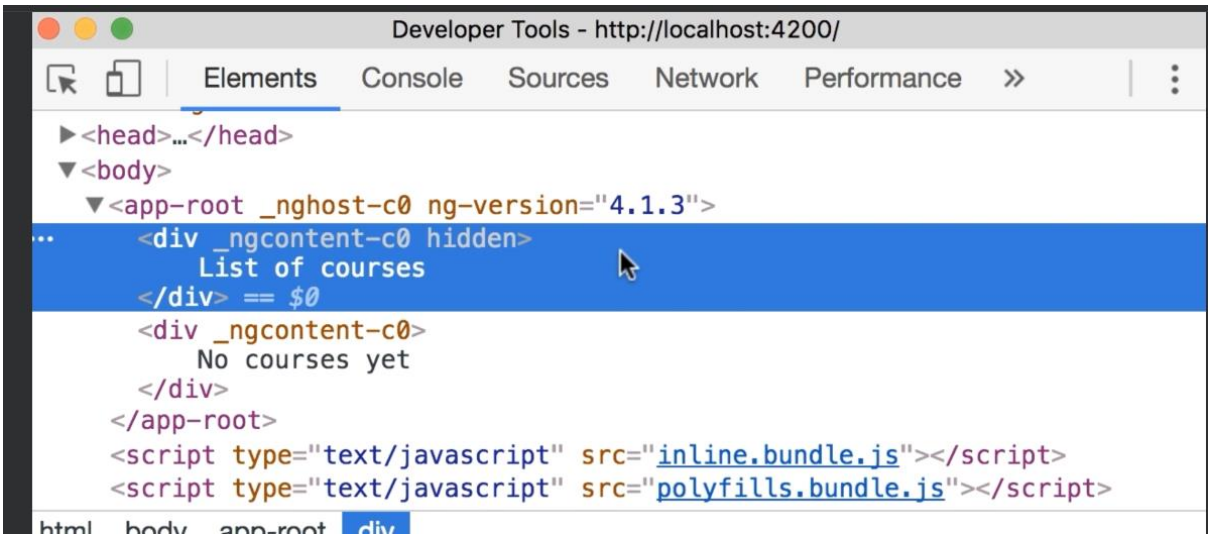
Now, you may not quite like this syntax because in the first block we're using a div. In the second block we're using NG template. They're kind of inconsistent. There is another approach, so we can define another NG template here. NG template. And this is for displaying the list of courses. So we assign it an identifier like courses list. And here we add that message list of courses. Okay, now we have one div with two templates. One for the condition that is truthy and the other for condition that is falsy. Okay. Now we need to change our if. So first we add our condition. If this condition is truthy then and here we specify the name of our template. So courses list. And then we have LS And here here's the other block. So note that we have a semicolon after the condition. Then we have then as a keyword here is the name of our target block or target template. After that we have LS as a keyword and finally the name of the other template. I personally prefer this approach because it's more clear and more consistent. Now there is another approach to show or hide part of a page, and that's what I'm going to show you in the next lecture.

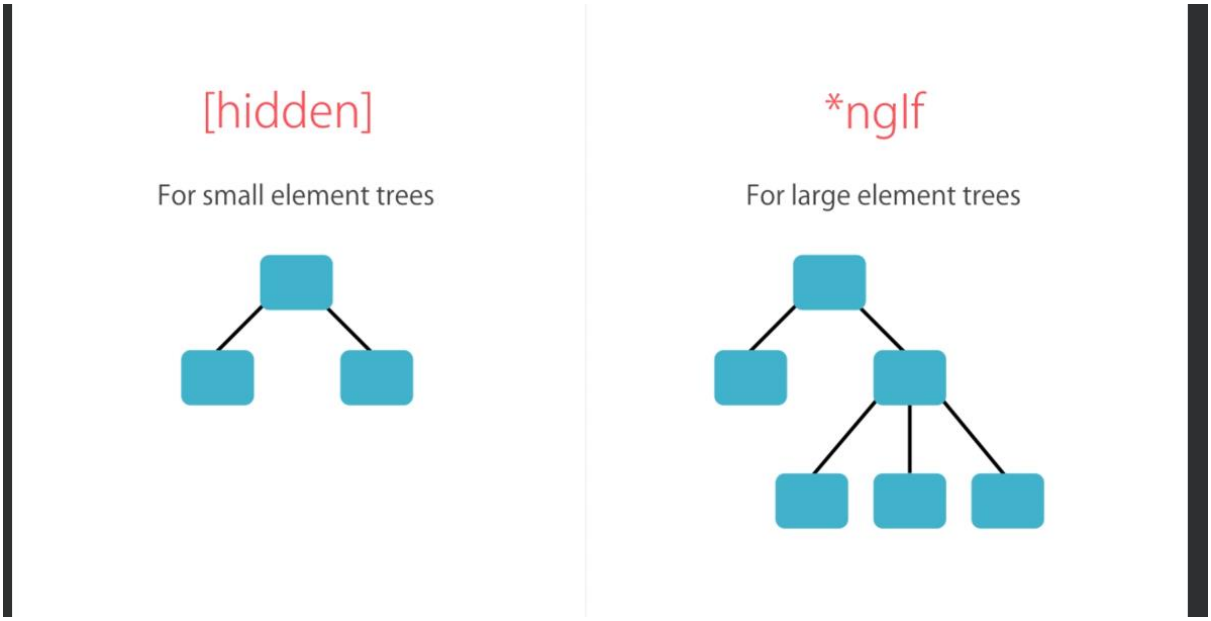
28. Hidden property



Now, back in the same example, let me show you another way to show or hide part of a page. Instead of using the If directive, we can use the hidden attribute. So in HTML, you know, we can apply this hidden attribute to our divs to hide them. Now, if you go back in the browser, we should see no courses yet. Message. Here is the message. Beautiful. Now this attribute in HTML also exists as a property in our Dom objects so we can use property binding and bind this property of the underlying Dom object. To some expression. For example, here I want to hide this first div if we don't have any courses. So courses dot length equals zero. Now similarly, we can hide the second div if we have at least one course. So courses dot length greater than zero. Now back in app component. Currently we have two items here. Let's see if application is working properly. So we get the list of courses. Beautiful. Now back in app component, I'm going to empty this array. Save and back in the browser. No courses yet. Beautiful. Now let me show you something. Let's inspect this element. Note that both Divs exist in our dom, but the first div has the hidden attribute and that's why it's hidden. So that's the main difference between using NGF and the hidden property. When we use NGF on an element, if the condition evaluates to false, i.e. that element is removed from the Dom, whereas when we use the hidden attribute, the element is in the Dom, it's just hidden. So you might ask which approach is better. Well, if you're working with a large tree with a lot of children, it's better to use NGF because these elements can take substantial memory and computing resources. You don't want to put them in the Dom if you're not going to show them to the user. Plus, Angular may continue to check for changes even to invisible properties. So the change detection mechanism in Angular, that's the mechanism that keeps your views in sync with your components, that's running in the background. And when these elements are in the Dom Angular continues to perform the change tracking on these elements. So in these situations, if you're dealing with a large tree with a lot of Dom objects, it's better to use NGF to free up resources. There is just one exception. In some situations, building a large element tree in the right state may be costly. So if you have a page with an element subtree. In that case, NGF may have a negative impact on the performance of that page. So if the user is going to click a button to toggle something to show or hide that part of the page, if building that elementary is costly,

you shouldn't use NGF. It's better to keep it in the Dom and hide it using the hidden attribute. So in summary, if you're dealing with a small tree of objects, it doesn't matter which approach you choose, it's purely your personal preference. If you're working with a large tree, first check to see if building that tree is going to be costly or not. If it's costly, use the hidden property to keep it in the Dom, but hide it. Otherwise it's better to use NGF to remove it from the Dom and free up the resources. Mosh posted an announcement · 5 years ago This language gives you 2x-4x more job opportunities Hi guys, Python is super-hot right now and I've had a ton of requests to create a Python

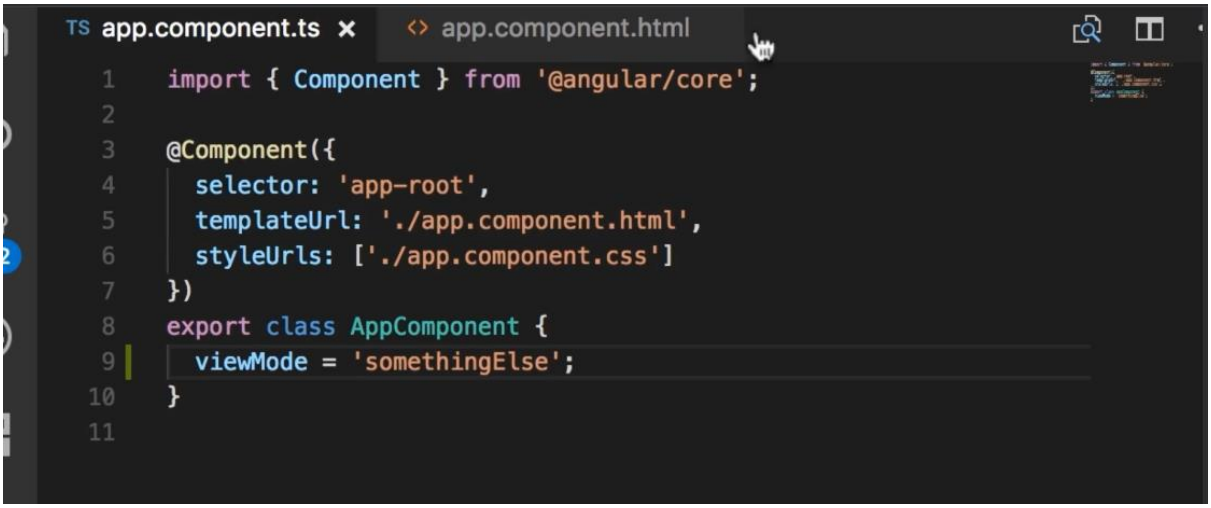


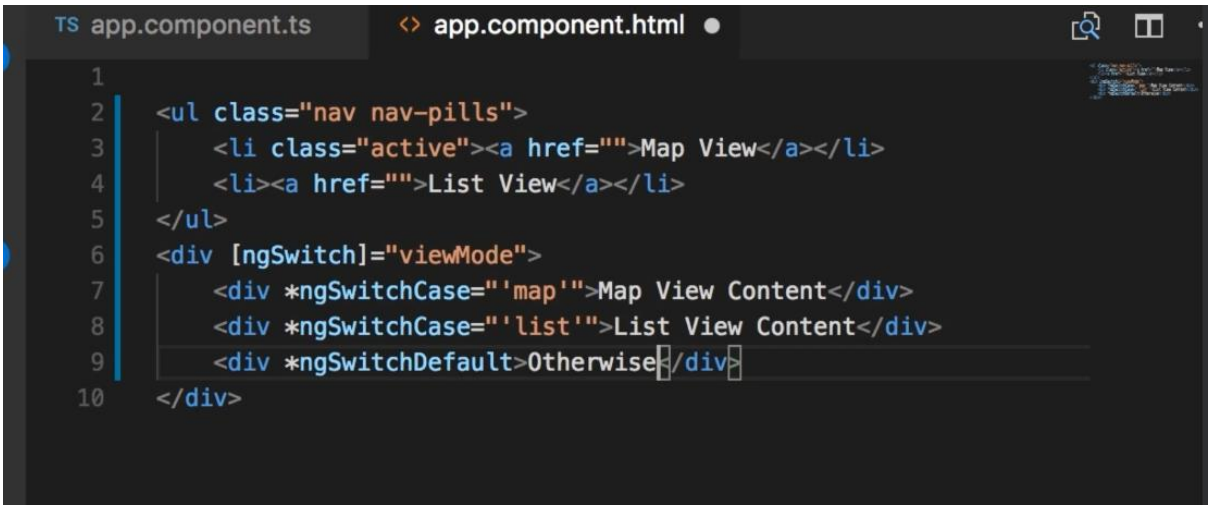


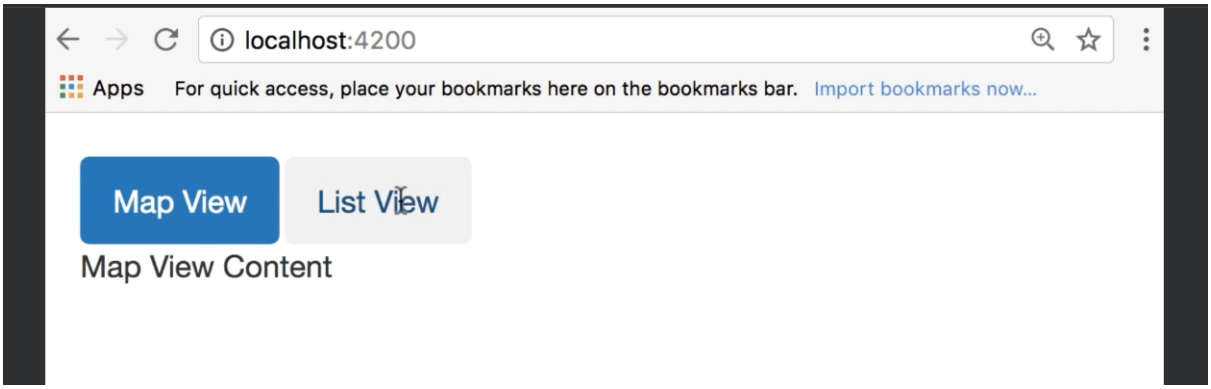
29. NgSwitchcase

In Angular, we have another directive called switch case, which is very similar to the concept of switch case we have in a lot of programming languages. So imagine we want to build a page to display properties on a map or in a list. So here we have two tabs, map, view and list

view. When we select the list view, we see the list view content. And when we select the map view, we see map view content. So let me show you how to implement this with switch case. And by the way, you can implement the exact same thing with NGF, but NGF works only if you have a truthy and a falsy condition. In this case we might have multiple tabs, so we cannot implement that using NGF and that's why we need NG switch case.



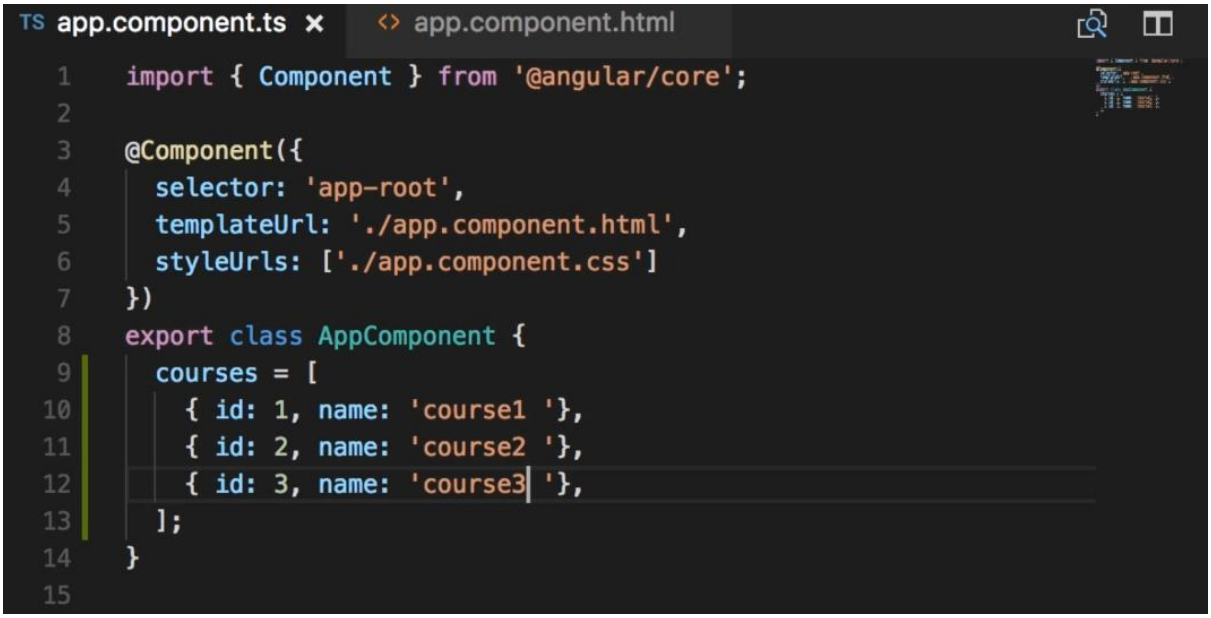


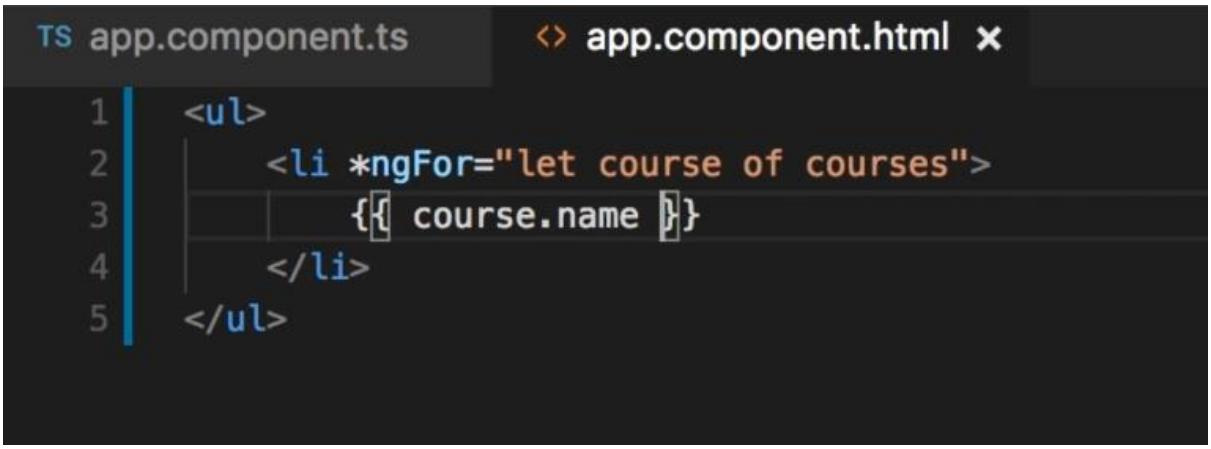


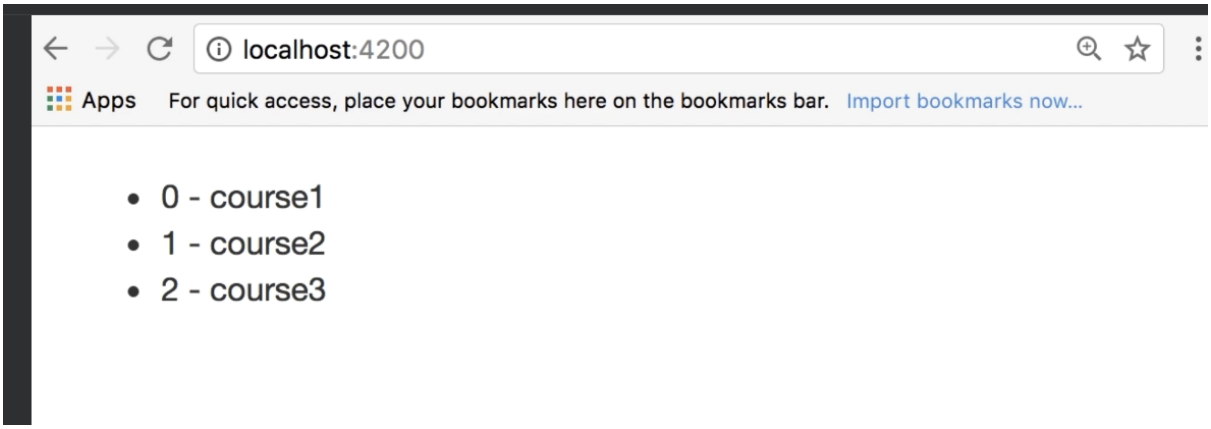
So in App.component.html, I want to add some basic markup to render bootstrap tabs. So here we want a Ul and this Ul should have two classes, so we add dot nav and also dot nav dash pills. This syntax you see here is what we call Xen coding. And with this we can quickly generate HTML markup. So tab. Now we have a Ul with these two classes. Now inside this Ul, we want an li and inside this li we want an anchor. Now we want two of these elements. So I put them in parentheses times two tab and this is what we get. Now let me temporarily make the first li active. So we apply the active class here. And name this map view. The second lie the anchor should be called. List View. Now below that we will have a div. And here we'll have either the content of the map view, so map view content or. List. View content. Now, I want to temporarily comment this out because one div one content div will be rendered dynamically. Now let's just make sure that our markup is working. Back in the browser. All right. It looks good. Beautiful. Now let's move on and make this dynamic. So let's go to our app component. Here. We need to define a field to keep track of the currently selected tab. We can call this view mode and we can set this initially to map and then we can change this to list so it can hold one of these two values map or list. Now back in our template. Let's bring this div back. We want to render one of these divs dynamically based on the value of view mode field. So on this top parent div, we're going to apply property binding. So Angular adds a new property to our Dom element that is not part of the standard dom. This property is called NG switch. We can use property binding and just switch and bind this. To have filled in our class like view mode. Now on each div we'll apply a switch case directive and because with this directive we add or remove a Dom element, this is a structural directive and we should prefix it with an asterisk. So ng switch case. Now if the value of view mode is map, so map as a string. Note that I put map in single quotes. Let me zoom in. So we have double quotes, which is for the value of this directive and we have single quotes, which is for the map as a string. Okay. Now. Similarly for the other div I'm going to apply. Engine switch case. We want to render this Dave if the value of view mode is list. Now. Optionally we can have another div. So if the value of view mode is something else, I want to render this div. So here we add Angie switch default and it doesn't have a value. We apply it like an attribute. Okay, so let's add some label here. Otherwise. Right. So this is how we use the engine switch directive. Now, finally, we want to make this class active dynamic so we can use class binding. We put it in square brackets. Class dot active. And we want to render this if view mode is set to map so we can write a simple conditional statement here. View mode equals map. Okay. And similarly for the other lie, I'm going to add another class binding class that active. We set this if view mode equals list. Now, finally, when we click these links here, we want to change the view mode. So let's remove this attribute. We use event binding and bind the click event to an expression. Now

here we can simply call a method in our class like change view mode and simply pass a string like map. But that method is going to be a one liner, so we can simply write that expression here. So we simply set view mode to map. Okay. And I'm going to copy this and apply it to the other link as well. But this time I'm going to change this value to list. Right now. Let's test the application. So back in the browser. Initially the map view is visible because we set the view mode to map in our component. If we click list, we go to the list view and back to the map view. Now back in the component. I'm going to change this value to something else. With this. This other div that we applied in switch default directive to this one should be rendered. So look, we get the other wise message here. So here's the lesson. If you want to compare the value of a field or a property against multiple values, use the switch case

30. NgFor

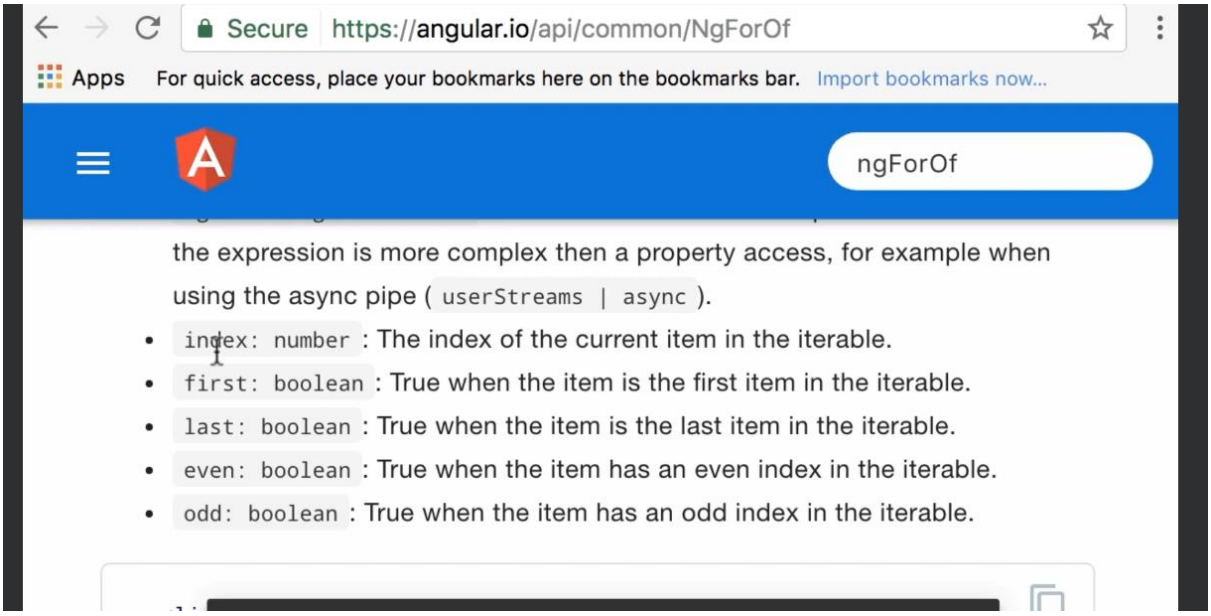


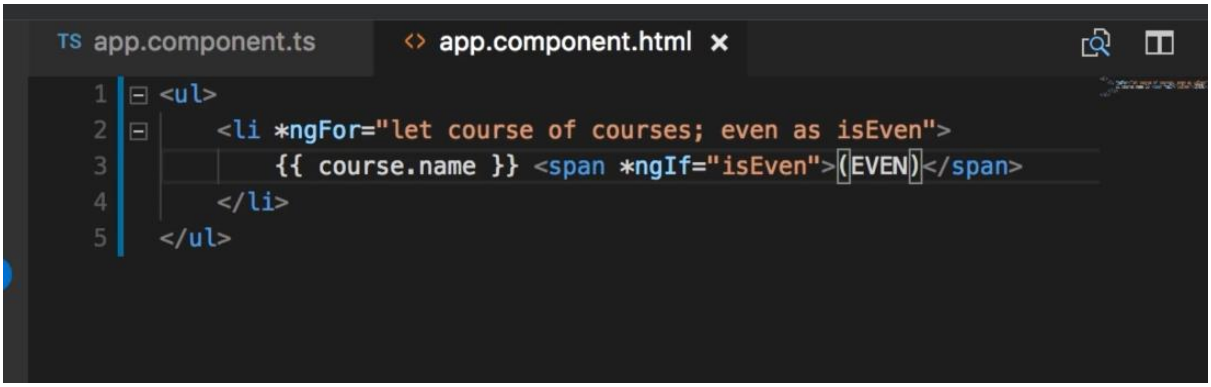




All right. You have seen the Ng4 directive before. We use this directive to render a list of objects. So let's review this one more time. And this time I'm going to show you more details about this directive. So here in App.component, I'm going to define a new field called courses. And set this to an array of objects. In this array, we're going to have three course objects, and for each course we're going to have two properties. One is ID and the other is the name. So course one. Now I'm going to duplicate this. So two and three and obviously, of course, two and course three. Now let's use the NG for directive to render these on The View. So app.component.html, we want to have a Ul and inside this ul we need an li tab. That's the markup. Now we add ng for. And here as the value we type, let course of courses. And then we can use interpolation to render the name of each course. So course dot name. Let's make sure the application works up to this point. So back in the browser. Okay. We got three courses on the screen. Beautiful. Now this for DirecTV exports a bunch of values that might help you build certain features. For example, imagine you want to render a table and you want to highlight the first row, or maybe you want to highlight the last row or you want to highlight the odd or even rows, or maybe you want to display an index next to each object. We can use the exported values from Ngfor directive. So here after this expression, we add semi-colon. One of these exported values is index. So we type index. Now we need to alias this to a local variable. So we type as and then the name of our local variable like I. So with this expression, we get the value of the index and put it in a local variable called I and then we can use interpolation to render I on the screen. So like this. I. All right, Now let's test this back in the browser. Look, before each course, we have its index in our courses field. So as you can see, the index of the first item or the first course is zero. Now index is one of these exported values. There are several other let me show you how you can find the

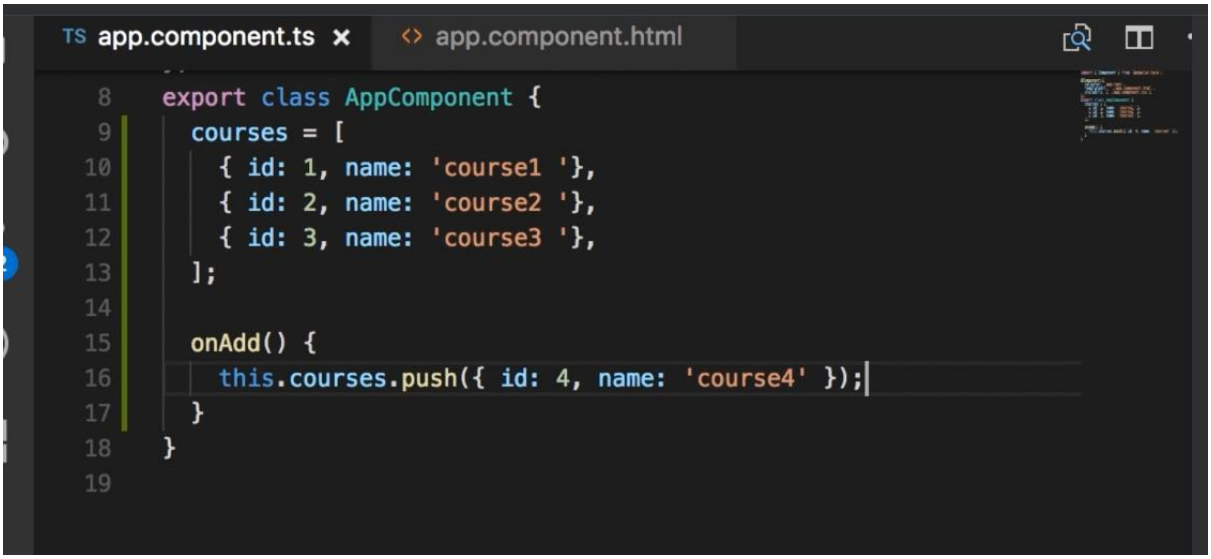
list of all these exported values. So if you head over to Angular.io and the search box, you can search for ng4 of directive. That's the name of our directive. Even though in HTML we use ng4. But the actual name of this directive is ng4. Of. And you can see that here it has a D icon which indicates a directive. Now on this page if you scroll down. Under local variables. You can see all these exported values. So you saw index. We also have first, last even and odd. So let me show you one more example. Let's say we want to render a table and we want to highlight all the even rows so we can get access to the even value. And aliases as a local variable is even because this even is a boolean, unlike the index, which is a number even odd, first and last values, they are all boolean now here. Next to each course I want to render a span. And I want to render this only if this is an even row. So if is even for now, just add a simple label. Now, in your applications, you might want to render a table so you can use this variable is even and apply some style or some class to even rows in your table. Now let's try this back in the browser. So you can see in front of the first and the third rows we have this label even.

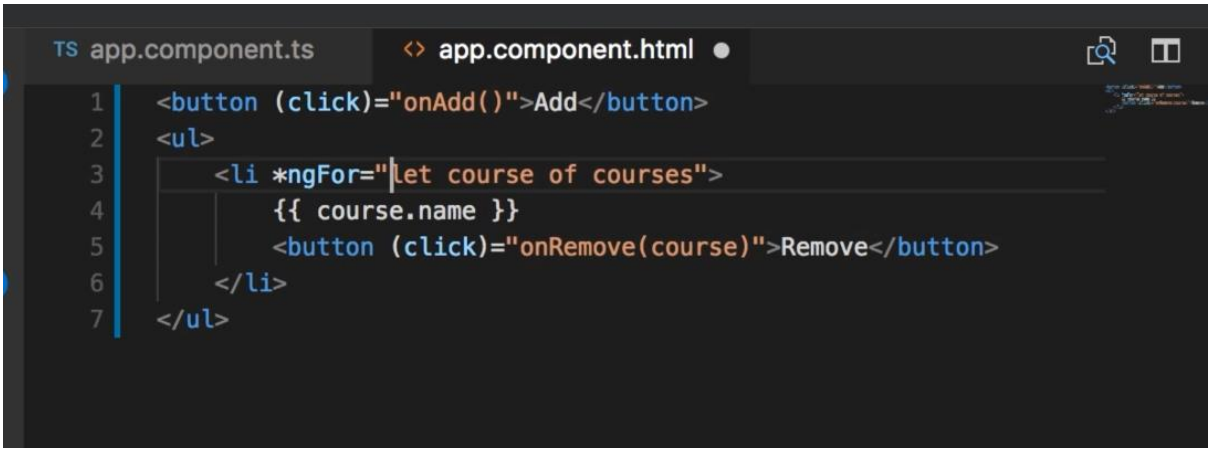




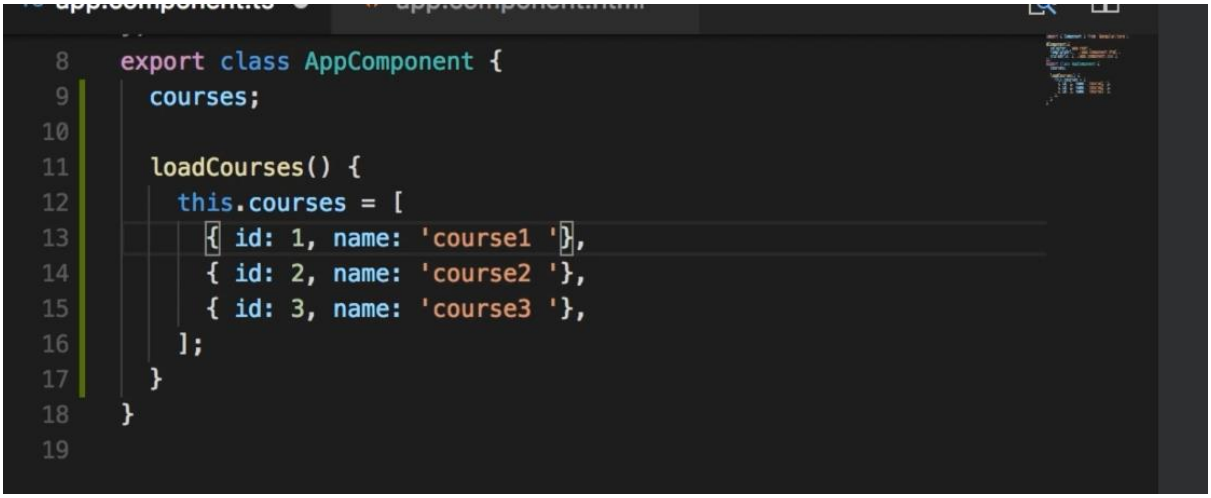
31. NgFor and change detection

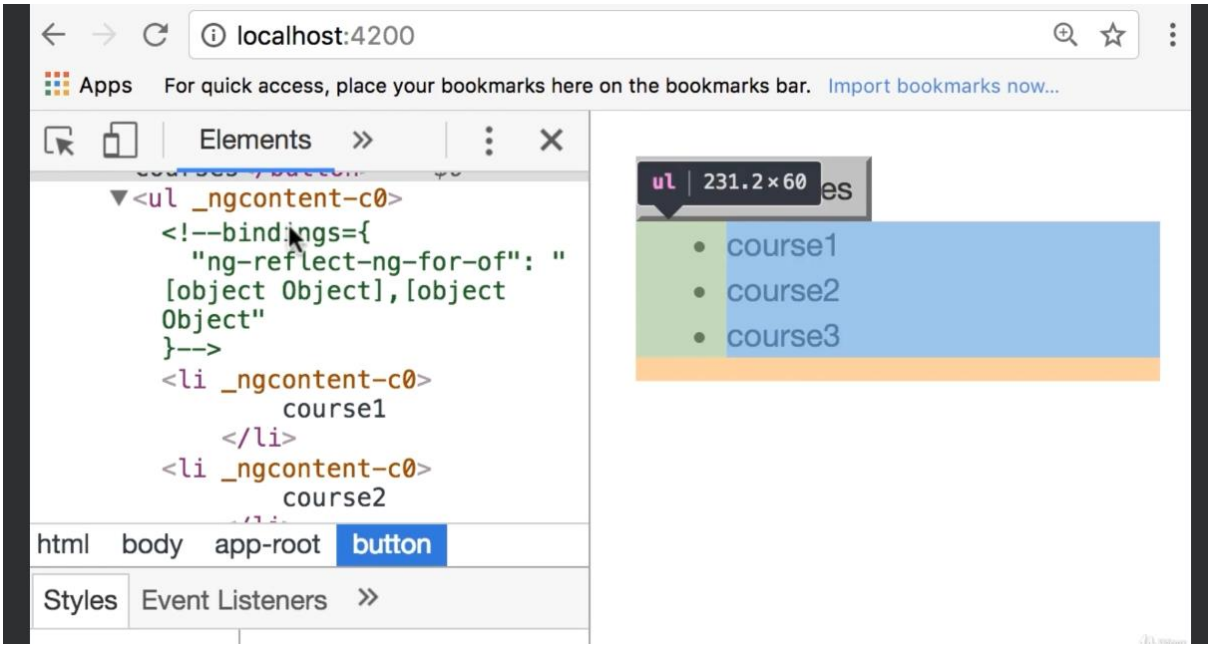
All right, Now let me show you how this Ng4 directive responds to the changes in the component state. So on the top, I want to add a button. With this button, we want to add a new course in our list. So here we can handle the click event and bind this to a method called on Add. Okay. Now back in the component. Let's add this new method here. So on Add. We call this dot courses that push. And we push a new object here. ID for a name is going to be course for. Save back in the browser. So here we have three courses. When I click the add button, we get a new course here. So Angular has this change detection mechanism. So whenever you click a button or whenever an Ajax request or a timer function completes. Angular performs its change detection. It looks at our component. It knows that this courses field now has a new object. That's the course with ID four. So then it will render that course using this template. Now. Similarly, we can add a button next to each course to remove it. So button. Remove. And here we can handle the click event. And bind this to this expression. We're going to call on remove method in our class. And as an argument, we can pass this course object that we have in each iteration. That is the course object here. Okay, now back in the component. So similarly, I'm going to add another method on remove. Here we get a course object. Now to remove this from an array. First, we need to find the index of this course in the array. So this dot courses dot index of. This course object. We get the index? And then we call this dot courses dot splice. So we go to that index and delete one object. Let's try it. So back in the browser. Now if I click this remove button, of course number two will be removed. There you go. So once again, after the execution of this on remove method Angular performed its change detection, it realized that one of the objects in our courses array is no longer there, so it removed the corresponding list item in the Dom. Now. Similarly, if we modify an existing object again, Angular will refresh the Dom automatically. So let's change the name of this method from on remove to on change. Back in the component. Rename it here now instead of removing this course from the array. I'm simply going to change its name. Update it like that. Let's test the application. You forgot to change the label. It doesn't matter. So when we click this, the name of the course changes Again, this is change detection in action. In the next lecture, we're going to look at this change detection from a performance point of view.





32. NgFor and trackBy





So once again in app component, we have this field courses with three objects. Now let's simulate something. Let's simulate a scenario where we get these courses from the server. So in our template I want to add a button on top. Let's call this load courses. And when we click this button. We're going to execute load courses method. Okay. Now back in the component. I'm going to add this method load courses. And then I'm going to set this courses field in that method. So delete. And we said this that courses to this array. So when we call this function, we're going to initialize this courses field. Now, in this particular demo, we're going to explicitly load the courses, but in a lot of real world applications, this happens automatically when you land on a page. Now don't worry about that aspect. What we're focusing on here is that every time we click this button, we are resetting this courses field to a new array. Okay, so save back in the browser. Now I'm going to open up Chrome developer tools and put it on the left side of the screen. So inspect. It's here. Let's expand the body element. Here is the app root. This is our button. Now I'm going to click this. All right, look, we have a Ul. And this is where we have our courses. So a Ul with three ls right. Now look what happens every time I click this button, you will see a quick purple highlight on the left side of the screen. Look. Do you know what that means? That means every time we click this button, Angular is reconstructing this Ul element. Look, now, this is so fast. It doesn't have any performance overhead. But if you're working with a large list or a list with a complex markup and sometime during the life cycle of your page, you're going to call the backend to download these objects. Angular is going to reconstruct this entire Dom object tree. And this can be costly if you're dealing with a large, complex list. Now let's see how we can optimize this.

Memory Location: 100

```
{
  id: 1,
  name: 'course1'
}
```

Memory Location: 200

```
{
  id: 1,
  name: 'course1'
}
```

TrackBy

course.id

app.component.ts | app.component.html

```
<button (click)="loadCourses()">Load Courses</button>
<ul>
    <li *ngFor="let course of courses; trackBy: trackCourse">
        {{ course.name }}
    </li>
</ul>
```

```
trackCourse(index, course) {
  return course ? course.id : undefined;
}
}
```

Angular by default tracks objects by their identity. So here we have three objects. These objects have three different references in the memory. When we reset this courses field, even though we're dealing with the exact same content as in course with ID, one, course with ID two and three. But these objects will be different from the previous ones in the memory. So Angular sees this as new content and that's why it reconstructs that Dom tree now back in our template. Here in the G4 directive, we have this ability to change how angular tracks. Objects. So as I said, by default it tracks them based on the object identity, which means the reference of that object in the memory. So if we redownload course with ID one every time we redownload that course, that course is going to be a different object in the memory. Even though the content of those objects are going to be equal. Now we want to instruct Angular to use a different mechanism to track objects. So instead of tracking them by their identity or the reference in the memory, we want to track them by their ID. So course with ID one is always course with ID one. So if we redownload the exact same course from the server and none of the properties are changed, Angular will not rerender that Dom element. So here we add track by and we set this to a method in our class. Let's say track course. Note that here I'm not calling this method. I'm simply adding the name of the method as a reference. Okay, now let's go ahead and implement this method. So back in our app component, I'm going to add track course. This method should take two parameters. Index and course. Now here we want to check if we have a course object. We want to return its ID. Otherwise we return undefined. Now, with this method, we change how Angular tracks course objects. So instead of tracking them by the object identity, it tracks them by their IDs. Let's try this back in the browser. So I'm going to click load courses. Now, here, look. This is where we have the list of courses. So a Ul and a bunch of lies. Now I'm going to click load courses several times, and you will no longer see that purple highlight on the left side of the screen. See nothing is happening. So Angular is not rerendering this URL and its children. Because every time we reset our courses field, we're using the same course objects. These course objects have the same ID. So here's a lesson. If you're dealing with a simple list, don't worry about the track by feature. You don't need this. Angular performs well out of the box. However, if you're dealing with a large list with complex markup and you do observe performance problems on a given page, you can try using track by to improve the performance of that page. So don't use it by default in every page because you have to write more code and you won't gain any performance benefits.

33. The leading asterisk

app.component.html

```
<div *ngIf="courses.length > 0; else noCourses">
    List of courses
</div>
<ng-template [ngIf]="courses.length > 0">
    <div>
        List of courses
    </div>
</ng-template>
<ng-template [ngIf]="!(courses.length > 0)">
    No courses
</ng-template>

```

Well, earlier you saw this gif with the LS syntax. So here, if we have any courses, we display them on the screen. Otherwise, we use this template to render the node courses message. Now, when we use this leading asterisk on Ngif, we are telling Angular to rewrite this markup. We write this block using an NG template element. So when Angular sees this leading asterisk. It's going to rewrite our div like this. It's going to create an engine template. It's going to put our div inside this template and the content of this div which is list of courses. And then it's going to apply if as property binding on this template. So property binding NGF is going to bind this to an expression courses dot length greater than zero. And then for the else part, it's going to apply similar property binding on the other template. So. And if. It's going to take the same expression, which is courses dot length greater than zero and then apply. The not operator to reverse this condition. So here is the lesson. When we use the leading asterisk with our structural directives like NG, if. Ng4 and ng switch case, Angular is going to rewrite that block using an NG template. Now obviously we can do this ourselves, but it's much easier to use the leading asterisk and let Angular do the hard work.

34. NgClass

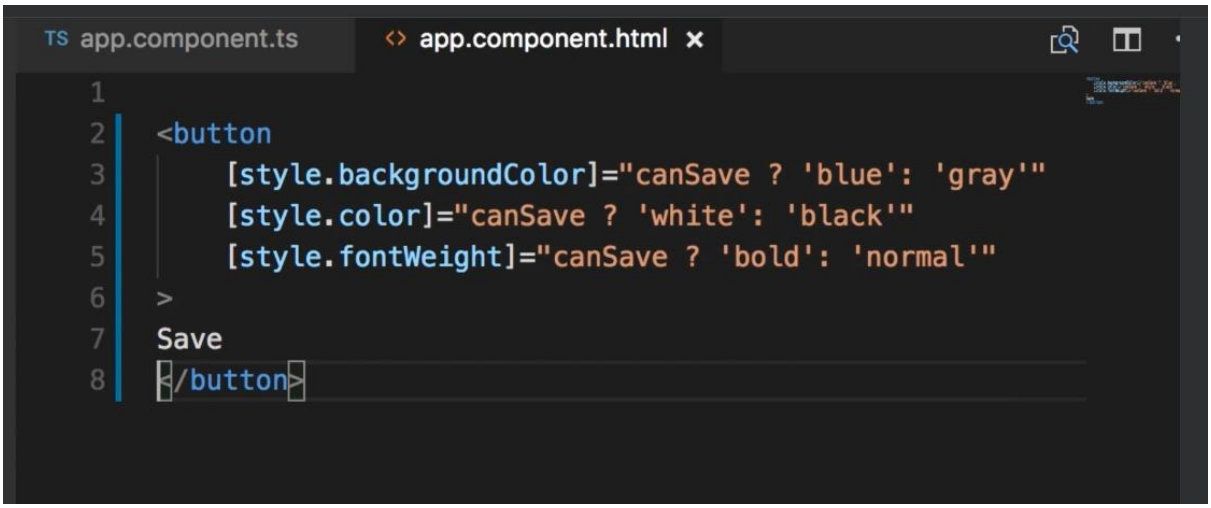
```
<span
  class="glyphicon"
  [class.glyphicon-star]="isSelected"
  [class.glyphicon-star-empty]="!isSelected"
  [
  (click)="onClick()"
  ></span>
```

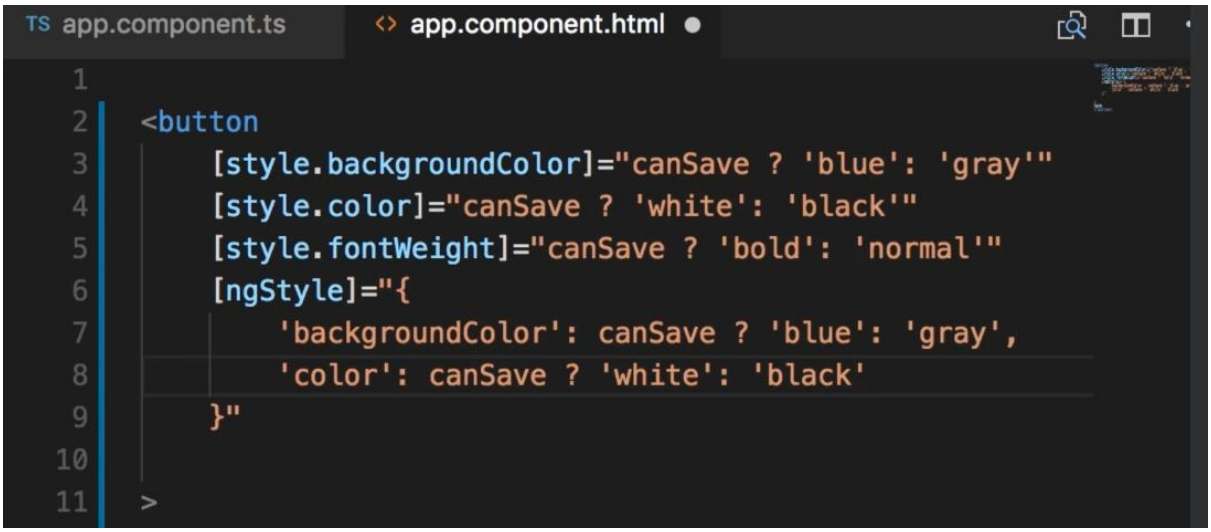
favorite.component.html

```
<span
  class="glyphicon"
  [ngClass]="{
    'glyphicon-star': isSelected,
    'glyphicon-star-empty': !isSelected
  }"
  (click)="onClick()"
  ></span>
```

So earlier we built this favorite component, and here we use class binding twice to dynamically render glyph icon star or glyph icon dash star dash empty classes. While this approach works perfectly fine, there is also another way to deal with multiple classes and you might find this other approach a little bit cleaner. So instead of using class binding twice here. We can use the class directive so we bind. Inter-class. An expression. And here we're going to have an object. And this object, we're going to have one or more key value pairs. Each key represents a CSS class, and the value for that key determines if that class should be rendered or not. So here we're dealing with two classes, so we need two keys. One is glyph icon dash Star. And the value for this key is is selected. So if this evaluates to true, then this class will be rendered in the Dom. Note that I've put the key in quotes and this is a requirement. Otherwise this is not going to work. Now let's add the other key. So once again, quotes Glyph icon. Dash star dash empty and we set this to not is selected. So with the class directive, we don't have to repeat class binding twice and we can simply delete these two lines. Now let's look at the result. So here's our icon. Click, click. Beautiful. So this class is an example of an attribute directive. We use it to modify attributes of an existing Dom element.

35. NgStyle

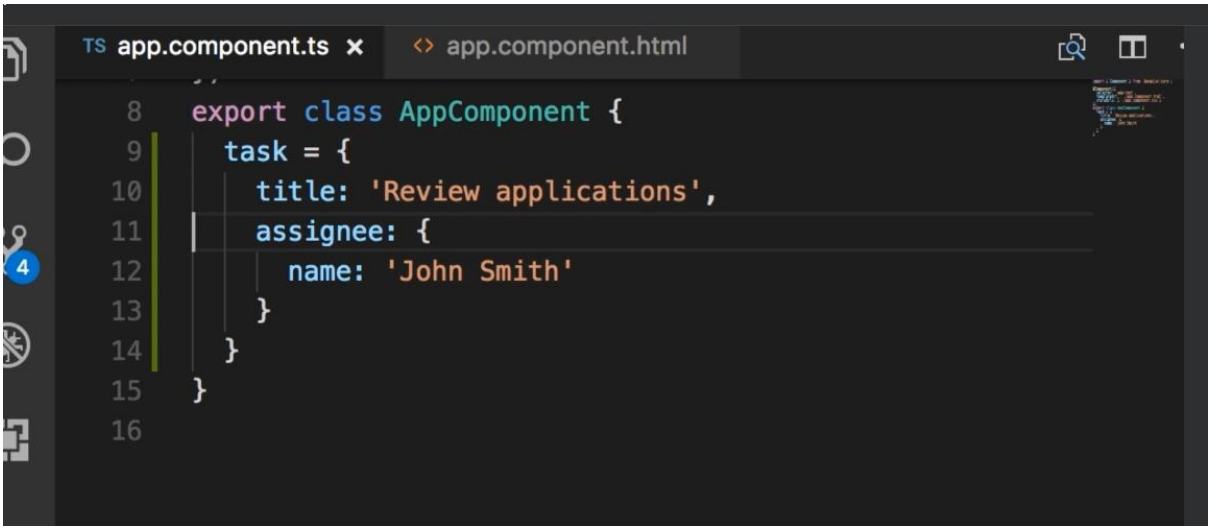


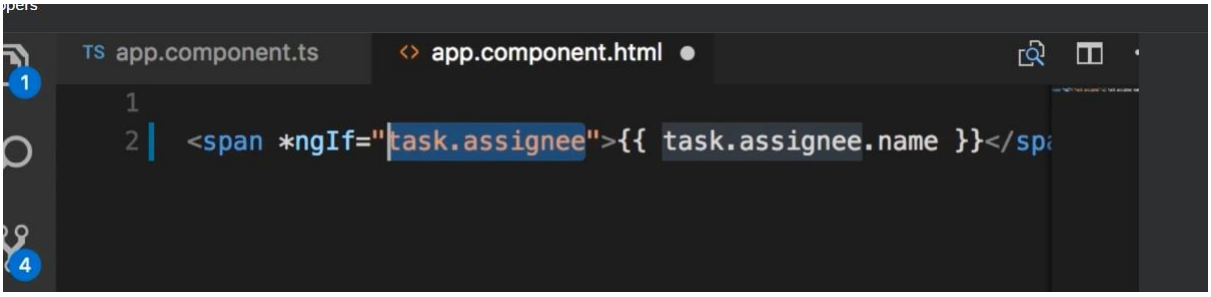


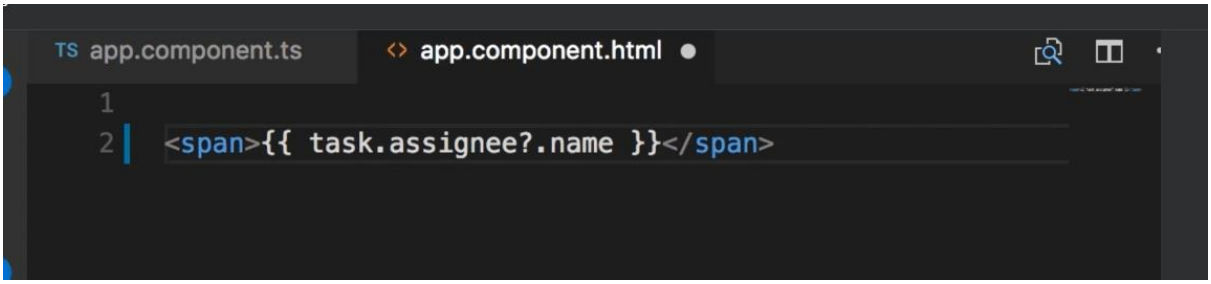
All right. Now in app component I have defined this field can save and set this to true. Now let's go to the template for this file. Here we have a button element with three style bindings. We have style, background, color, style, dot color and style dot font weight. And for each style binding you can see we have an expression that uses the value of can save to determine the value for that style. So if can save is true, background color is going to be blue. Otherwise it's going to be gray. And similarly, we have these two other styles here color and font weight. If you look in the browser. So if can save is true, this is what we're going to get. Otherwise, if you go to app component and change this to false, this is what we're going to get. Now, this template, as you can see, is a little bit noisy. In the last lecture you learned how we can use the class directive to clean up multiple class bindings. Now, similarly, we have another directive, another attribute directive called NG style. So when you're dealing with multiple style bindings, you may prefer to clean up your code by using the NG style directive. So let's see what we can do here. Once again, we use property binding syntax and bind NG style to an expression, and here we're going to have an object. In this object, we're going to have one or more key value pairs. Each key represents a CSS style, so we put it in quotes just like before. Background color. And then for the value we

use the same value we had in the style binding. So if can save is true, we're going to use blue. Otherwise we're going to use gray. Now. Similarly, we can have another key value pair here. So color is going to be white or black. Now, obviously, this is not the best way to implement this feature. If you're dealing with multiple styles, it's better to encapsulate them in a CSS class and then depending on the value of can save your render one of these classes instead of adding multiple styles here. But sometimes in certain situations you may want to add styles explicitly here. So if that's the case, use either style binding or the NG style directive to add

36. Safe traversal operator







All right. Now in app component, I have defined a field called task. Here we have an object with two properties title and assignee, which is another object itself. This assignee has a

name property which is set to John Smith. Now in the template for this component. You can see I've used a span with interpolation to render the name of the assignee for this task. So when we go to the browser, we see John Smith on the screen. Now, sometimes when you're dealing with complex objects, it is possible that the value of a property may be null or undefined for a certain period of time. For example, you might want to call different endpoints to get these objects from the server. So then assignee might be null or maybe a fraction of a second. Let me simulate this scenario. So I'm going to change the value of this property to null. Now back in the browser. We have a blank screen. So let's take a look at the console. Look at this error. Cannot read property name of null. So because assigning is null, we cannot access the name property here. Now, there are two solutions to solve this problem. One way is to use the NG if directive. So ng if we want to render this span only if this task object has an assignee. So task dot assignee. If this evaluates to truthy then this span will be rendered. So I'll save back in the browser. So, look, we have no errors in the console and there is nothing on the screen. Now, there is also another way to solve this problem. So maybe you want to keep the span in the Dom, but you don't want to render the name of the assignee if it's null. So let's remove this. If. We can use what we call safe traversal operator. So because assignee can be null or undefined, we put a question mark after it before the dot. Now, with this syntax, if assignee is null or undefined, Angular is going to ignore this. Otherwise it's going to render the name property of the assignee on the screen. So let's go back to the browser. Look, we have no errors. And if we inspect the dom. So under app route we have our span, but there is nothing inside the span. So this is safe traversal operator Be aware of it. And we want to use this when dealing with complex objects.

37. Creating custom directives

app.module.ts

```
14  import { InputFormatDirective } from './input-format.dir
15
16  @NgModule({
17    declarations: [
18      AppComponent,
19      CourseComponent,
20      CoursesComponent,
21      AuthorsComponent,
22      SummaryPipe,
23      FavoriteComponent,
24      PanelComponent,
25      InputFormatDirective
26    ],
27    imports: [
```

input-format.directive.ts

```
1  import { Directive } from '@angular/core';
2
3  @Directive({
4    selector: '[appInputFormat]'
5  })
6  export class InputFormatDirective {
7
8    constructor() { }
9
10 }
11
```

TS input-format.directive.ts

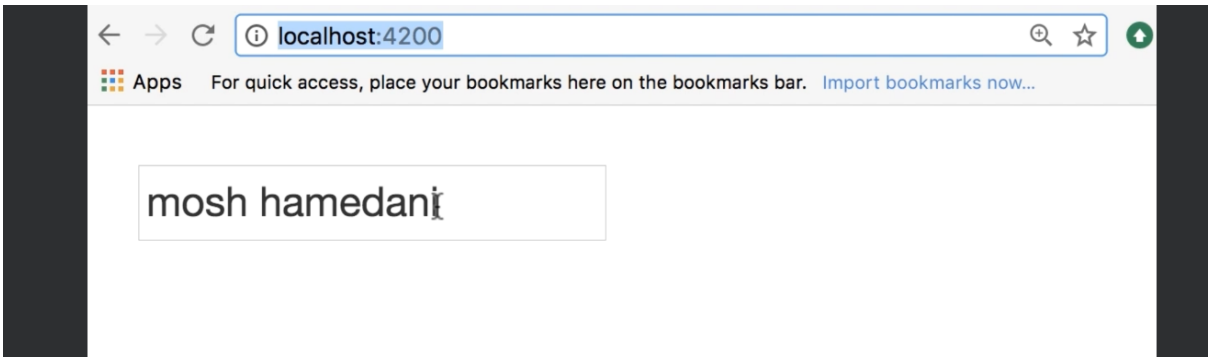
```ts
import { Directive, HostListener } from '@angular/core';

@Directive({
  selector: '[appInputFormat]'
})
export class InputFormatDirective {
  @HostListener('focus') onFocus() {
    console.log("on Focus");
  }

  @HostListener('blur') onBlur() {
    console.log("on Blur");
  }

  constructor() { }
```

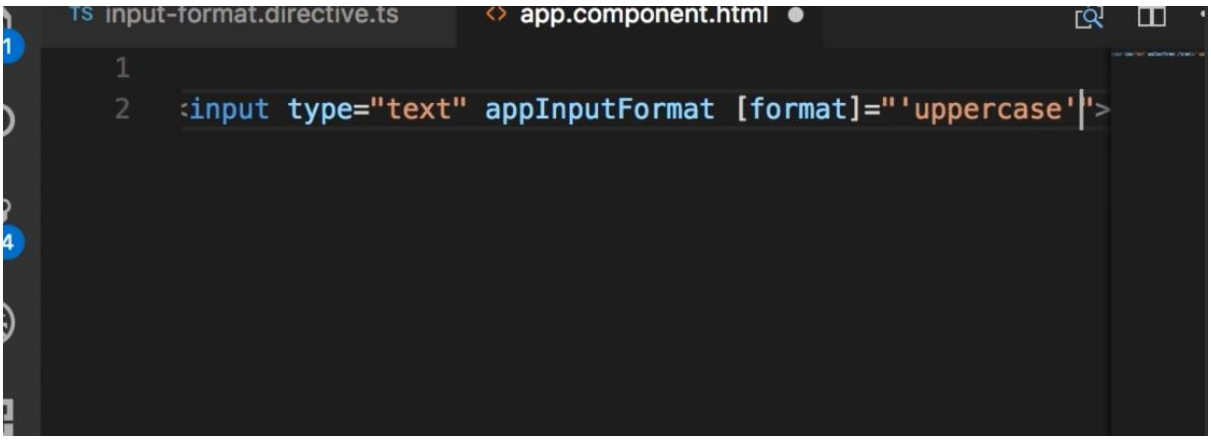
Reviews Learning tools

TS input-format.directive.ts <> app.component.html

```html
<input type="text" appInputFormat>
```

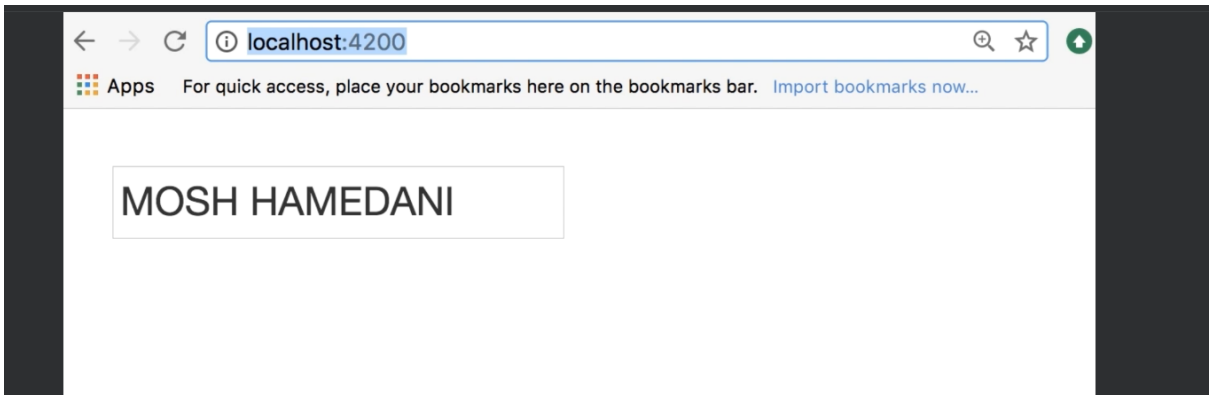

input-format.directive.ts
app.component.html
import { Directive, HostListener, ElementRef } from '@an
@Directive({
selector: '[appInputFormat]'
})
export class InputFormatDirective {
constructor(private el: ElementRef) { }
@HostListener('blur') onBlur() {
let value: string = this.el.nativeElement.value;
this.el.nativeElement.value = value.toLowerCase();
}
}

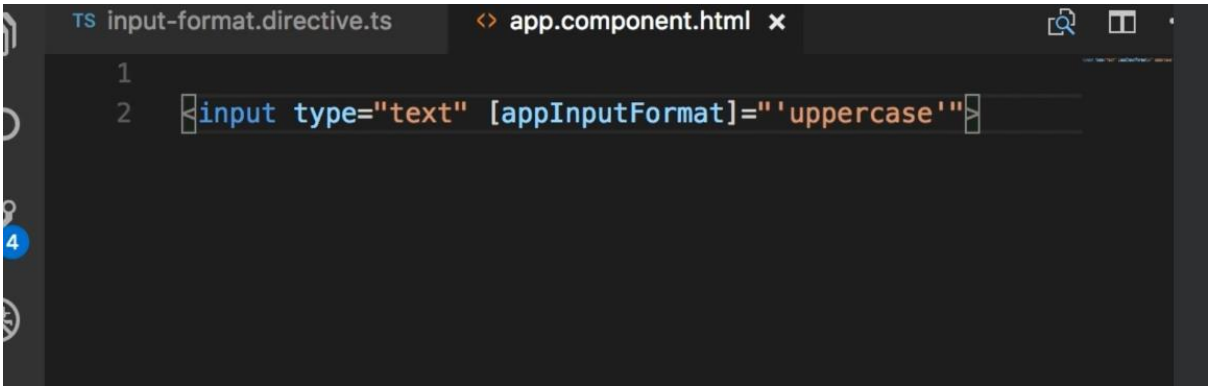

localhost:4200
Apps
For quick access, place your bookmarks here on the bookmarks bar.
Import bookmarks now...
mosh hamedani

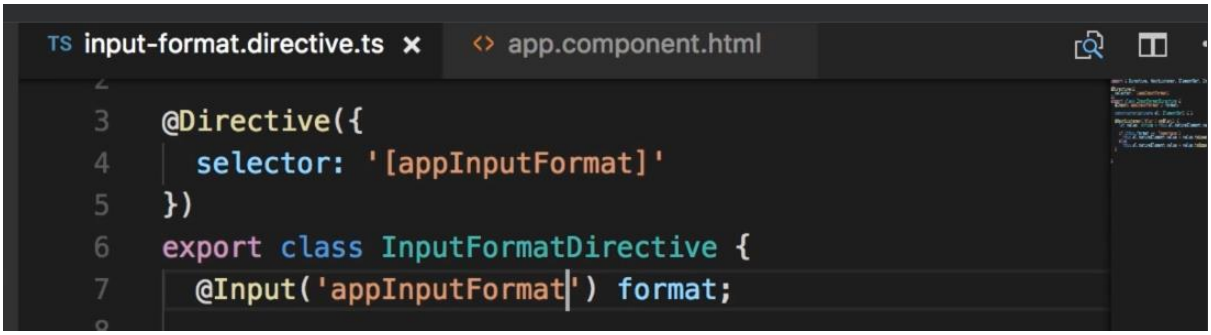

input-format.directive.ts
app.component.html
<input type="text" appInputFormat [format]="'uppercase'">
4


input-format.directive.ts
app.component.html
import { Directive, HostListener, ElementRef } from '@an
@Directive({
selector: '[appInputFormat]'
})
export class InputFormatDirective {
@Input('format') format;
constructor(private el: ElementRef) { }
@HostListener('blur') onBlur() {
let value: string = this.el.nativeElement.value;
this.el.nativeElement.value = value.toLowerCase();
}


input-format.directive.ts
app.component.html
@HostListener('blur') onBlur() {
let value: string = this.el.nativeElement.value;
if (this.format == 'lowercase')
this.el.nativeElement.value = value.toLowerCase();
else
this.el.nativeElement.value = value.toUpperCase();
}


localhost:4200
Apps
For quick access, place your bookmarks here on the bookmarks bar.
Import bookmarks now...
MOSH HAMEDANI



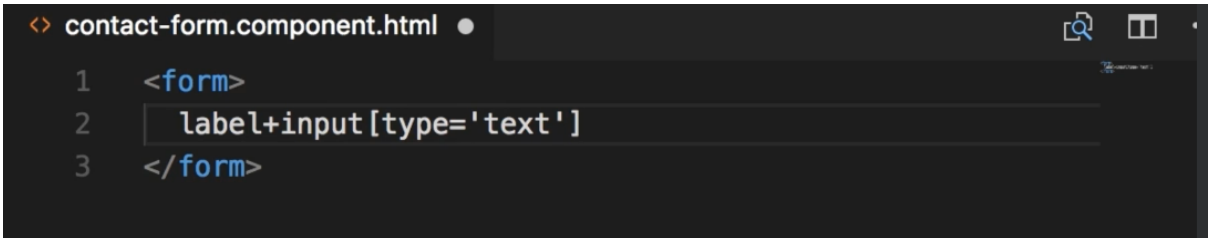


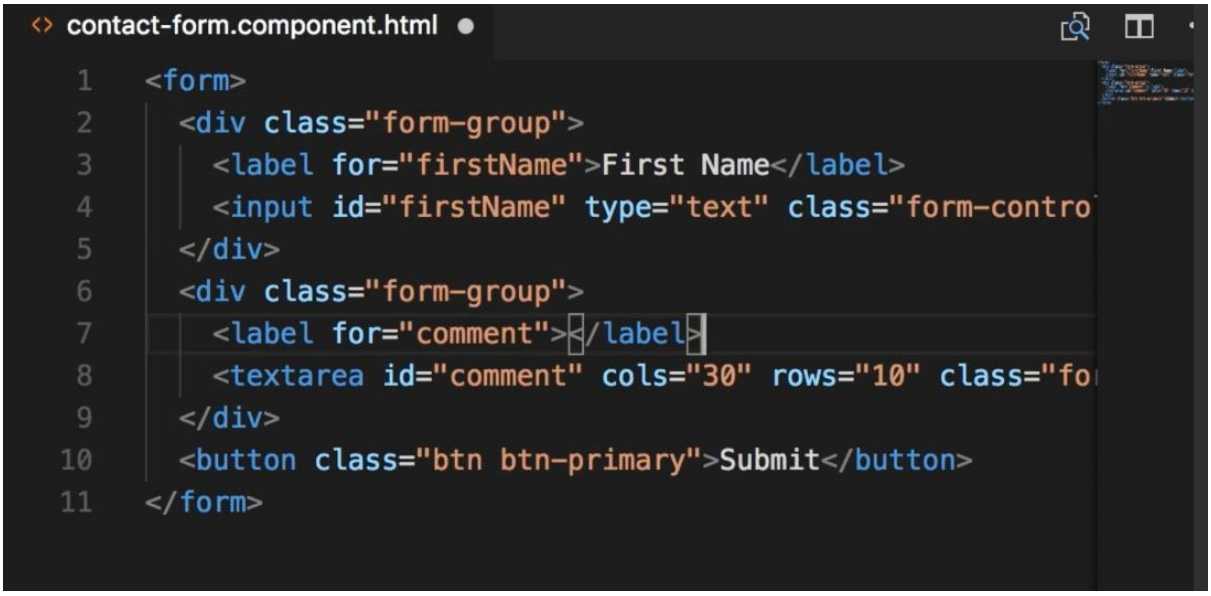
There are times that you want to have control over the behavior of Dom elements. For example, look at this input field here. If I type my name like this and press tab. Note that my name becomes lowercase. Now, this is a simplified example. In a real world application, you may want to add this behavior. So let's say we have an input field for the user to enter their phone number. So they add one, two, three, four, five, 67890. Now, the standard way to format an American phone number is like this. So the first three digits in parentheses, then three more digits and then an hyphen. So as the user types their phone number into this input field and moves away, you may want to reformat this value using a standard American phone number format. In situations like that, we use directives. So let me show you how to implement a simple directive in Angular. Now similar to components and services, we can create a directory from scratch or we can use angular CLI to generate a directive with some boilerplate code. So open up the terminal and type ng g d for directive and let's call this directive input dash format. So this creates two files. One is a unit test file and the other is the actual directive file. And it also modifies Appmodule. So let's quickly go to Appmodule. In our ngmodule inside declarations. You can see now we have input format directive. So as I told you before in this declarations array, we should register all the components, all the pipes and all the directives that are part of this module. If you don't do this, you're going to get a runtime error. Now let's go to our input format directive. So this is very similar to a component. You can see we have a TypeScript class called Input Format directive, and this class is decorated with the directive decorator function. Here we have a selector exactly like components, but the selector has square brackets, which basically means any elements that has this attribute app input format. If Angular finds an element with this attribute, it's going to apply this directive on that element. Now, as a best practice, it's good to prefix your

directives so they don't clash with the standard HTML attributes or other directives defined in third party libraries. So here we have app as a prefix to app input format directive. Now let's move on to the actual implementation of this directive. So here we want to handle two Dom events, focus and blur. First, on the top we need to import a decorator function. Host Listen here from angular core, this decorator allows us to subscribe to the events res from the dom element. That is the dom element that is hosting this directive. Or in other words, does the dom element that has this attribute. So let's see how we can use host listener. First, I'm going to define a method here on Focus. So whenever we get the focus in our input field, we want this method to be called. For now I just want to log something in the console. Unfocus. Now we need to decorate this method with the host listener decorator. So assign host. Listener. And as the argument we need to supply the name of the Dom event, that is focus. Now I'm going to select this code, duplicate it. And change the second instance to. Blur. On blur and on blur. Let's test the application up to this point. So let's go to App.component.html. Here, I'm going to add an input and apply app input format attribute on this element with this attribute. Angular is going to apply our custom directive to this input field. So let's test this. Back in the browser. So here is our input field. I'm going to click here and then outside. Now let's take a look at the console. So we have these two messages. Onfocus and onblur. Now let's implement the logic to change the value of this input field to a lowercase string. In this particular demo, we don't really need this. Onfocus I just wanted to show you how it works because we do the formatting once the user clicks outside this input field. So let's delete this. In Onblur. I need to get the value for this input field. First, we need a reference to the host element. So in our constructor we need to inject an element reference object so private l of type element ref. This is a service defined in Angular that gives us access to a Dom object. So let's import this at the top. You can see this is defined in angular core. Okay. Now, as a best practice, we should always put our constructors before all our methods. So this tells me immediately what are the dependencies of this class? Now, in Onblur, first we need to read the value of this input field. So let value equal this dot l and here we access the native element property. This gives us access to the actual Dom object. So from here we can read. The value property. And then we can simply set this that native element dot value to value dot. Now here the type of value is any, as you can see here. So I'm going to use type annotation to tell TypeScript compiler that this is actually a string. And with this we can access all the methods defined in the string class. So here I'm going to use two lowercase. Let's test the application. So back in the browser, I'm going to type my name here. Press tab. Beautiful. Now it's lowercase. Now back in App.component.html, it would be nice to have the flexibility to tell the directive about the target format. Maybe here we want to reformat the string as lowercase, but somewhere else we want to reformat it as uppercase. So if we had a property like format, then we could use property binding like this. We can set this to uppercase string. Note that here I've put my uppercase word in single quotes to specify that this is a string, not a property of the app component. Okay, so how do we implement this? You should already know we're going to define a field called format and mark it as an input property. As simple as that. So back in our directive. Let's define this new field format and decorate it with the input decorator. As a best practice, I told you always use a string alias here to keep the contract of your directives or your components stable. Now, finally, we need to import this type. From Angular core. And then in Onblur method. We can check if this dot format equals lowercase. We will reformat this string as lowercase otherwise. We're going to reformat it as. Uppercase. Okay, that's better. So let's go back to our App.component.html. Here. I've used uppercase, so let's test the application. I'm going to type my name one more

time. Tab. Now it's uppercase. Beautiful. The only issue here is that we have to apply this directive as an attribute and then use property binding to set the target format. Since we have only one input property here, it would be nicer to set the target format while applying the directive as an attribute like this. We could use square brackets here. And then set this to uppercase. Then we wouldn't have to bind to the format property. This is cleaner, right? So how can we implement this? Very easy. Let's go back to our directive. So all we have to do here is to change this format to the selector of our directive. So app input format. Let's try it back in the browser. I'm going to type my name. Tab. Beautiful. So here's a lesson you can use custom directives to have more control over behavior of Dom elements. You can pass data to your directives using input properties, and if you have only one property, you can use the selector of that directive as an alias for that property. And this simplifies the usage of your custom directive. And finally, you can use the host listener decorator to subscribe to the events raised from the host Dom object. Mosh posted an announcement · 5 years ago This language gives you 2x-4x more job opportunities Hi guys, Python is super-hot right now and I've had a ton of requests to create a Python tutorial.

38. Building a bootstrap form





TS contact-form.component.ts

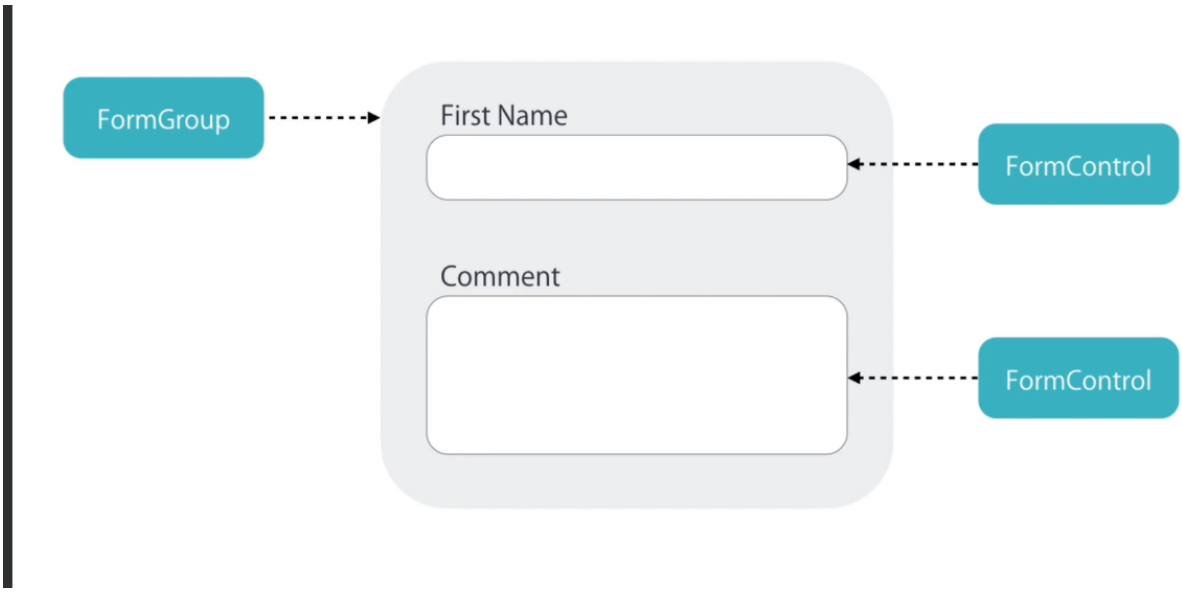
```ts
import { Component, OnInit } from '@angular/core';

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

  constructor() { }

  ngOnInit() {
  }

}
```

TS contact-form.component.ts <> app.component.html

```html
<contact-form></contact-form>
```

39. Types of form

FormGroup, FormControl



Creating Controls

Template-driven **Reactive**

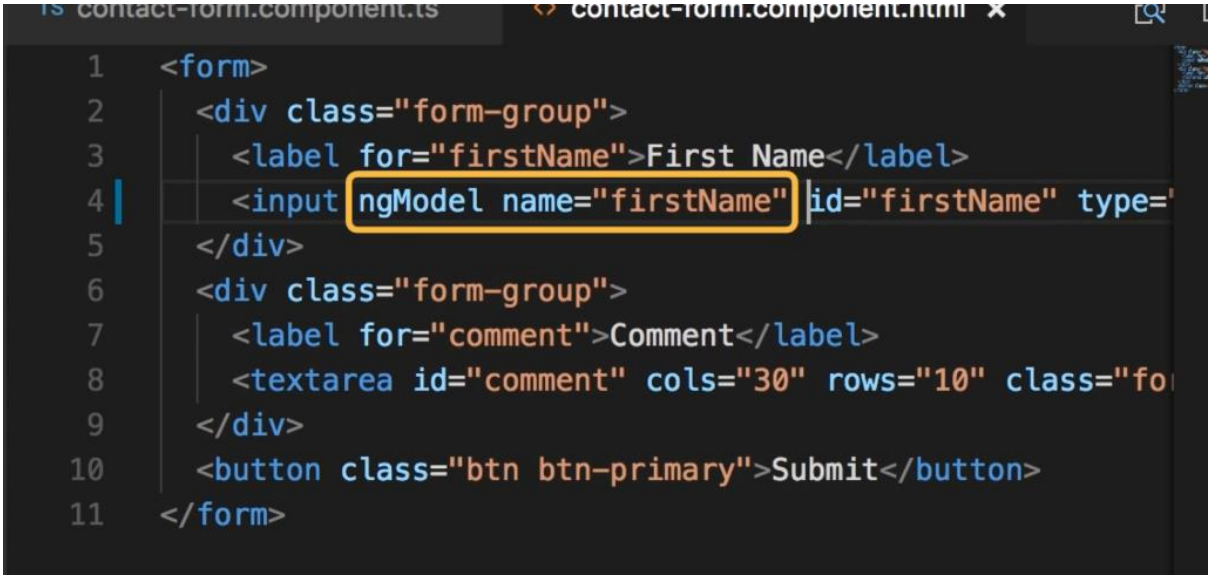
Reactive

- More control over validation logic
- Good for complex forms
- Unit testable

Template-driven

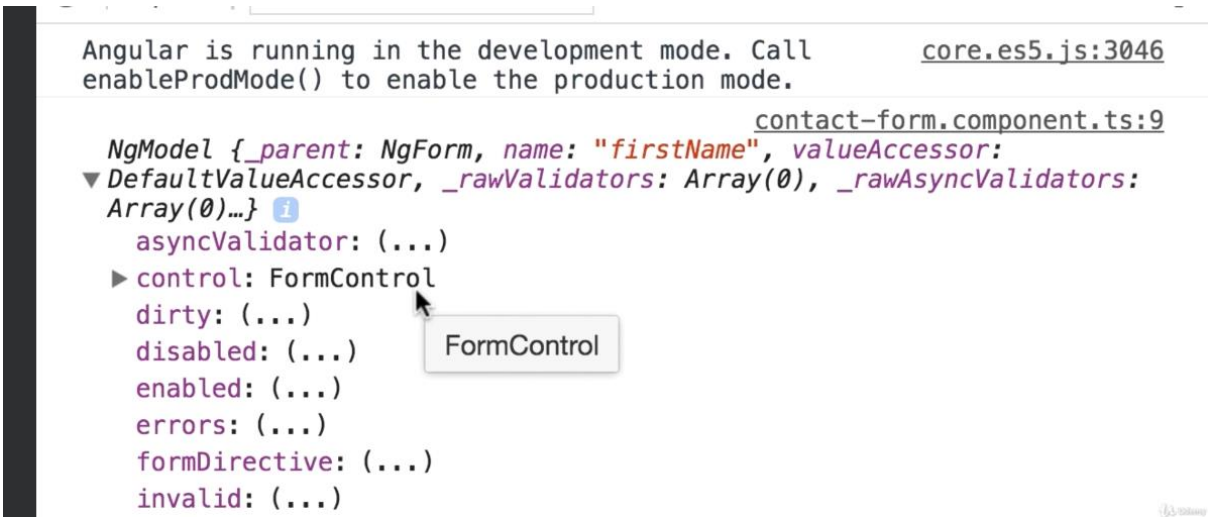
- Good for simple forms
- Simple validation
- Easier to create
- Less code

40. Ngmodel



So we have built this basic bootstrap form. Now we want to add validation to this using template driven approach. So as we explained in the last lecture with template driven approach, we apply a directive to our input fields and Angular will create a control object and associated with that input field under the hood. Now guess what? You already know this directive. That is ngmodel. So here on this first name input field, I'm going to apply the Ngmodel directive. We use this before with the two way binding syntax or banana in a box. Remember that? So we used it like this. Here's a box, here is a banana and we use Ngmodel like this and bound this to a property or a field in our component like firstname. Now here we don't need two way binding yet. Perhaps we can add this in the future. But all I want to show you is that when you apply the Ngmodel directive on an input field. In its simplest form, without any binding, Angular will create a control object and associated with this input field. Under the hood. Now let's preview this in the browser. So back in Chrome we get this error. If Ngmodel is used within a form tag, either the name attribute must be set or the form control must be defined as standalone. So basically the reason we got that error is that we didn't set a name attribute here. This is a requirement because every time we apply this ngmodel on an input field and it needs a way to distinguish these control objects. So here we set the name attribute. I'm going to use the same value as the ID, So first name. Save. Now back in the browser, the error is gone. Now I want to show you that control object under the hood. So these two attributes are all you need to set in order to use the template driven approach. But in this video I want to show you what is happening under the hood. So

I want to handle the change event of this input field. And here we're going to call the log method. Now I need a reference to this Ngmodel directive. I want to pass that reference to the log method so we can log it on the console. For that we're going to create a template variable. So hashtag let's call that template variable. First name we could call it anything and as the value I'm going to use NG model. So when Angular sees this, it's going to set this template variable to the NG model directive that is applied on this input field. Okay. And then we can pass this variable to our log method. So here is the log method. We pass first name. Now let's go and implement this method. So back in our component. I'm going to delete the stuff that we don't need. Let's just keep it simple. Here is the log method. Let's give it a parameter called x and simply log it on the console. This is just for diagnostics. Now, back in the browser, I'm going to type my name here, press tab. Now let's take a look at the console. Okay, here's our NG model. So you can see NG model is an object with these properties. Now look at this property control. This is the control object that I was telling you about. Now, if you click on the value look, it's an instance of the form control class in Angular. Let's expand this. Here we have a bunch of properties and many of these properties come in pairs. For example, we have this dirty which determines if the value of the input field is changed from the moment the form was initialized. In this case, because I typed my name here, dirty is true. Now the opposite property is pristine. Pristine means clean. So dirty should be true. And pristine is false. Okay, now we have another pair invalid and valid. In this case, because we haven't implemented validation yet, this input field is considered valid. So invalid should be false and valid should be true. Now, if you have any validation errors, they are available here. Currently errors is null because we don't have any errors. We have another pair which is touched and untouched. In this case, because I touched the input field or in other words, I put the focus there and then moved away. Touched should be true and untouched should be false. We have another important property value which returns the current value in the input field. So what I want you to note here is that we use this form control class to track state changes and the validity of input fields. When we apply the Ngmodel directive, along with the name attribute on an input field, Angular automatically creates an instance of the form control class and associates it with this input field. Now, to finish up this lecture, I'm going to apply Ngmodel on our text area as well. And set the name attribute to comment in the next lecture.



41. Adding validation

Name validation:

42. Specific validation errors

```
1   orm>
2   div class="form-group">
3     <label for="firstName">First Name</label>
4     <input required minlength="3" maxlength="10"  ngModel n
5     <div class="alert alert-danger" *ngIf="firstName.touche
6   /div>
7   div class="form-group">
8     <label for="comment">Comment</label>
9     <textarea ngModel name="comment" id="comment" cols="30"
10  /div>
11  button class="btn btn-primary">Submit</button>
12  form>
```

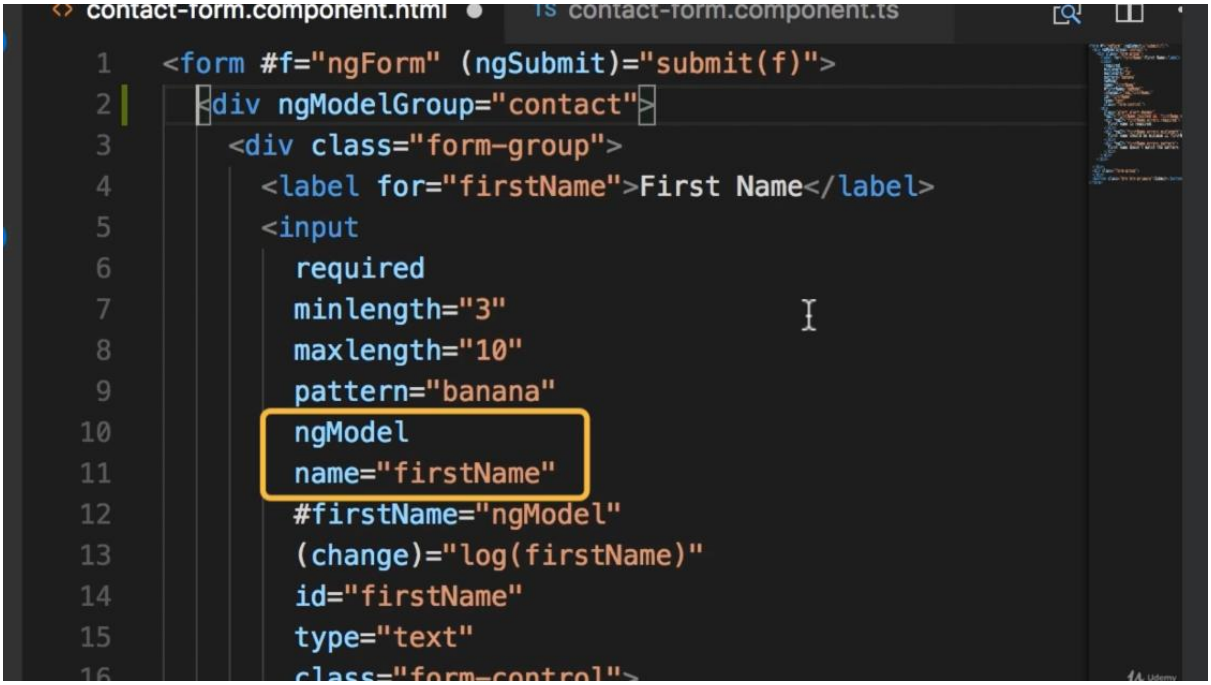
43. Ngform

contact-form.component.html ● TS contact-form.component.ts

```
1  <form #f="ngForm" (ngSubmit)="submit(f)">
2    <div class="form-group">
3      <label for="firstName">First Name</label>
4      <input
5        required
6        minlength="3"
7        maxlength="10"
8        pattern="banana"
9        ngModel
10       name="firstName"
11       #firstName="ngModel"
12       (change)="log(firstName)"
13       id="firstName"
14       type="text"
15       class="form-control">
16     <div
```

You can see details in of form in logs

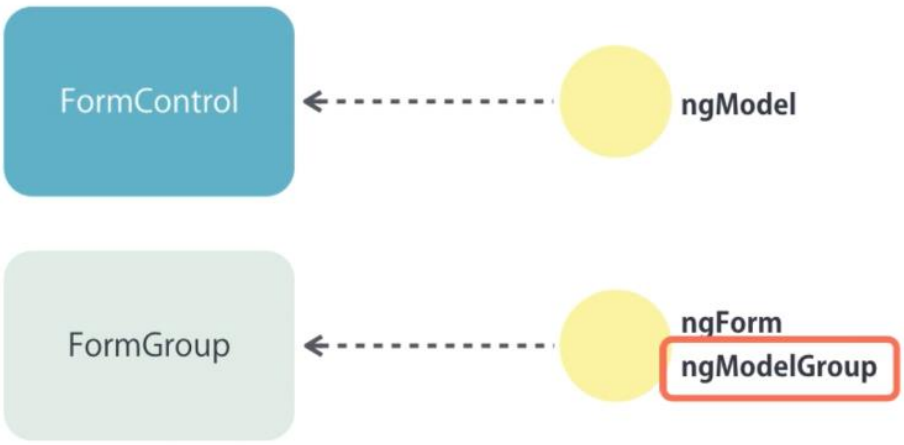
44. NgmodelGroup



Sometimes when you're working with complex forms, you might have multiple groups in your form. So back in our previous example, look at this value property. I've made a tiny change in our form and now instead of the first name property, we have contact, which is a complex object. So this contact currently has only one property which is first name. Potentially here we could have last name, email, whatever. Also, we could have address property, which could be another complex object. We could have shipping building. You got the point. So how do we represent a structure like this? Well, just like we have the NG

model directive, we also have NG model group directive. So back in our template I'm going to add a div here. And apply NG model group directive to it and set the value to contact. Okay, so this represents the sub property in our value object. Now here's our first form group that includes our first name field. I'm going to collapse this markup. And cut it and move it inside our model group. Okay. Save. But all we have to do. So here's our model group called Contact. Inside this group, we have our first name field. And with this change now, our value object is a complex object that is hierarchical. So we have the contact group and here we have the first name. So in your applications, the API you have on the server may expect a complex nested object structure like this. If that's the case, you use the NG model group directive. Also, similar to the NG model, you can get a reference to this directive using a template variable. So here we can define a template variable called contact. And set it to NG model group directive. Now we can reference this contact variable anywhere in this template and this is useful if you want to validate an entire group as a whole. Let's say you have a group called Billing Details and you want to display all the validation errors for the billing details on top of that group. If that's the case, then you would have a div. Here you would have NGF and we can use this template variable contact to see if this entire group is valid or not. And of course, inside this div we're going to have all the validation errors for that

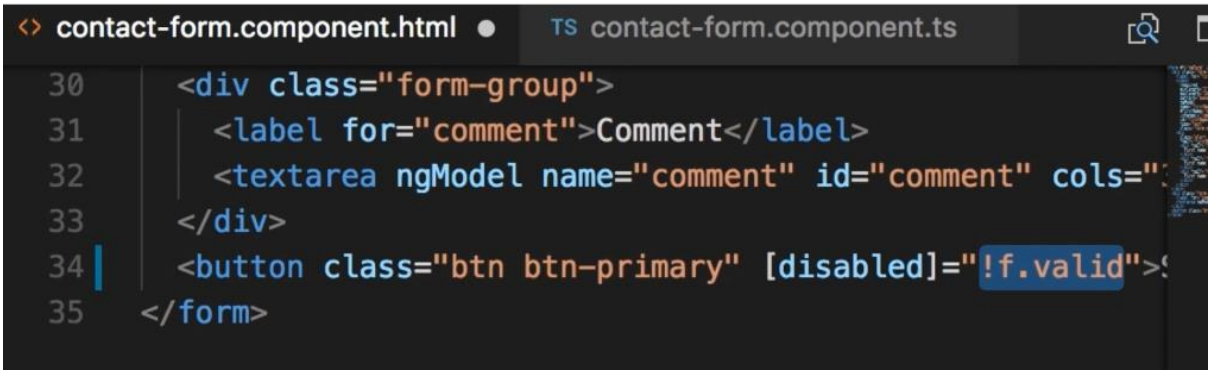
45. Control classes and directives



Now, if you're confused about all these classes and directives, let me make it super simple for you. In Angular, we have two classes to keep track of the state of input fields and their validity. One is form control, which represents only one input field and the other is Form group, which represents a group of input fields. Now, when we apply the Ngmodel directive on an input field, Angular automatically creates a form control object and associates that with that input field. So this is form control and ngmodel. Now form group class is used to represent an entire form and optionally groups within a form. As we explained before, we

have a directive called NG form that is automatically applied to all form elements. So Angular automatically applies the NG form directive to your forms, and this will internally create a form group object and associate it with your form. And with this form group object, we can track the state changes of the form and its validity. Now, if you have a complex form with multiple subgroups, we can optionally apply the Ngmodel directive on a subgroup. And this directive, similar to the form Directive, will also create a form group object for that group. Now you might be curious what is the difference between Ngform and Ngmodel group? The difference is that the Ngform directive exposes an output property which you saw earlier that is now submit, and we use this to handle the submit event of forms and to model group doesn't have that output property because it doesn't make sense to submit a part of a form. So this is all about form control, form group and the related directives.

46. Disabling the submit button



47. Working with drop down lists

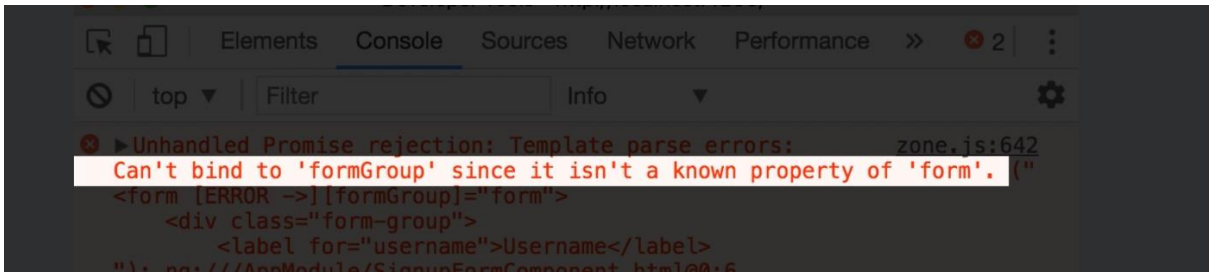
```
    </div>
    <div class="form-group">
      <label for="contactMethod">Contact Methodd</label>
      <select ngModel name="contactMethod" id="contactMeth
        <option value=""></option>
        <option *ngFor="let method of contactMethods" [valu
      </select>
    </div>
    <p>
      {{ f.value | json }}
    </p>
    <button class="btn btn-primary" [disabled]="!f.valid">S
  </form>
```

ABOVE IS TEMPLATE DRIVEN FORM NOW BELOW IS REACT FORM

48. Creating controls programmatically

signup-form.component.ts signup-form.component.html

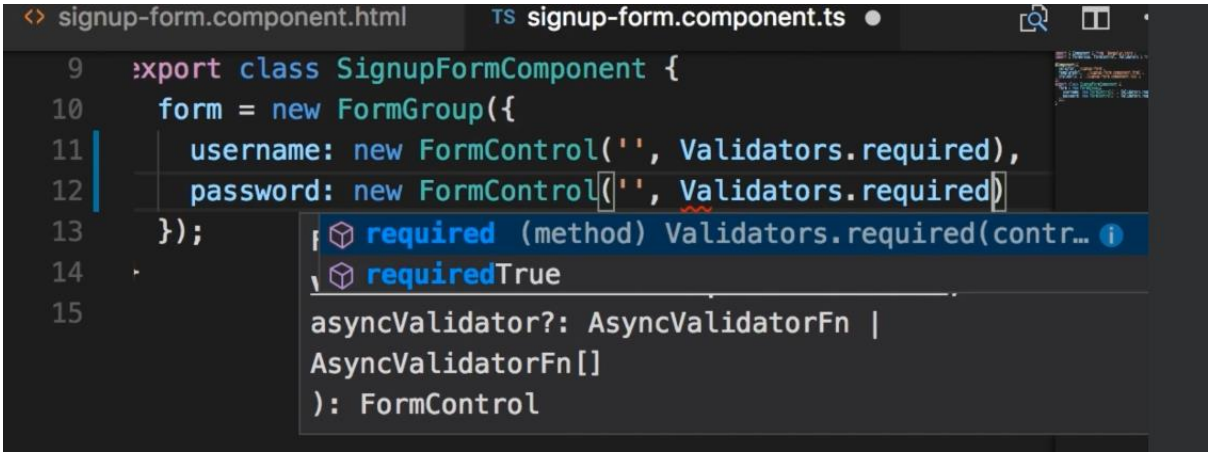
```
<form [formGroup]="form">
    <div class="form-group">
        <label for="username">Username</label>
        <input
            formControlName="usernam"
            id="username"
            type="text"
            class="form-control">
    </div>
    <div class="form-group">
        <label for="password">Password</label>
        <input
            id="password"
            type="text"
            class="form-control">
    </div>
```

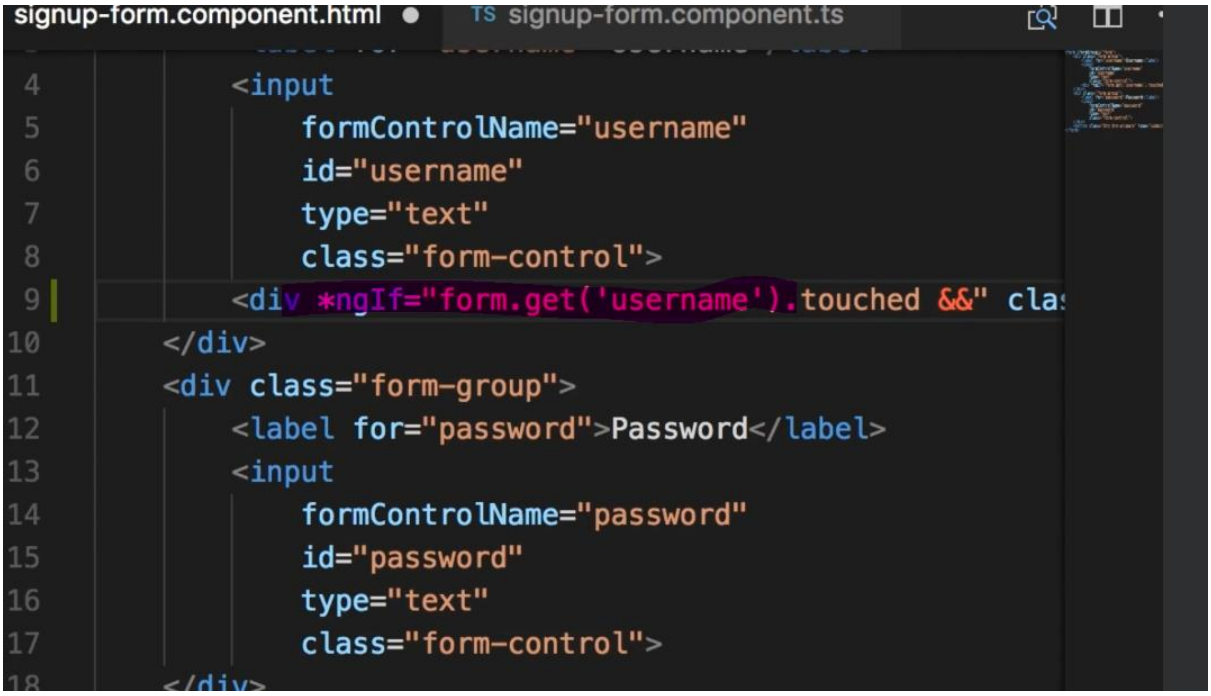


n.component.ts signup-form.component.html app.module.ts

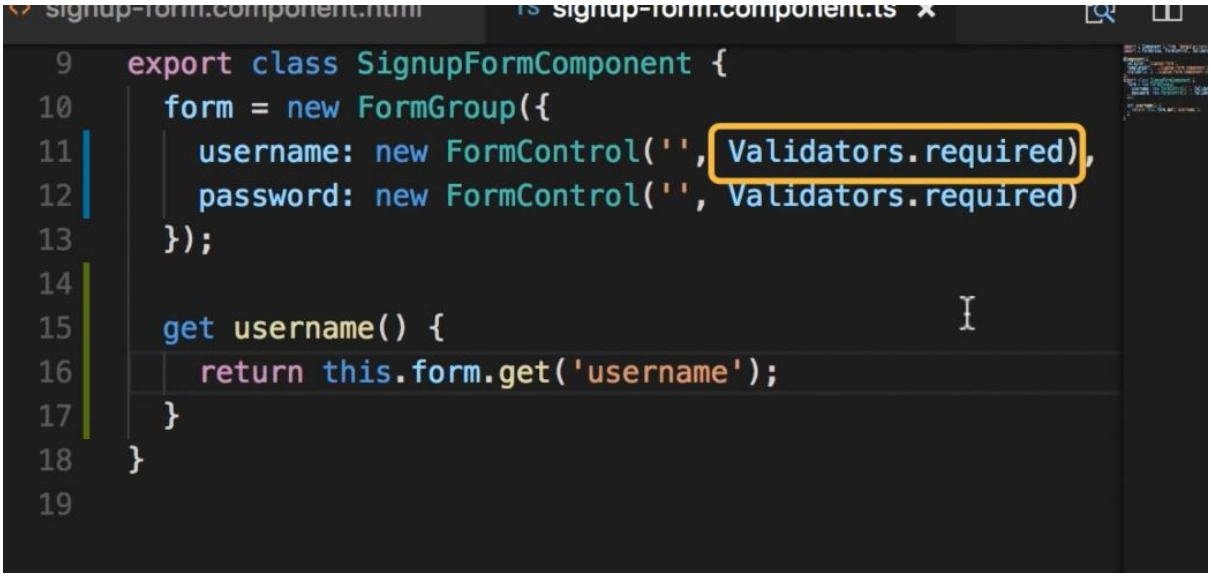
```
    ],
    imports: [
      BrowserModule,
      FormsModule,
      ReactiveFormsModule
    ],
```

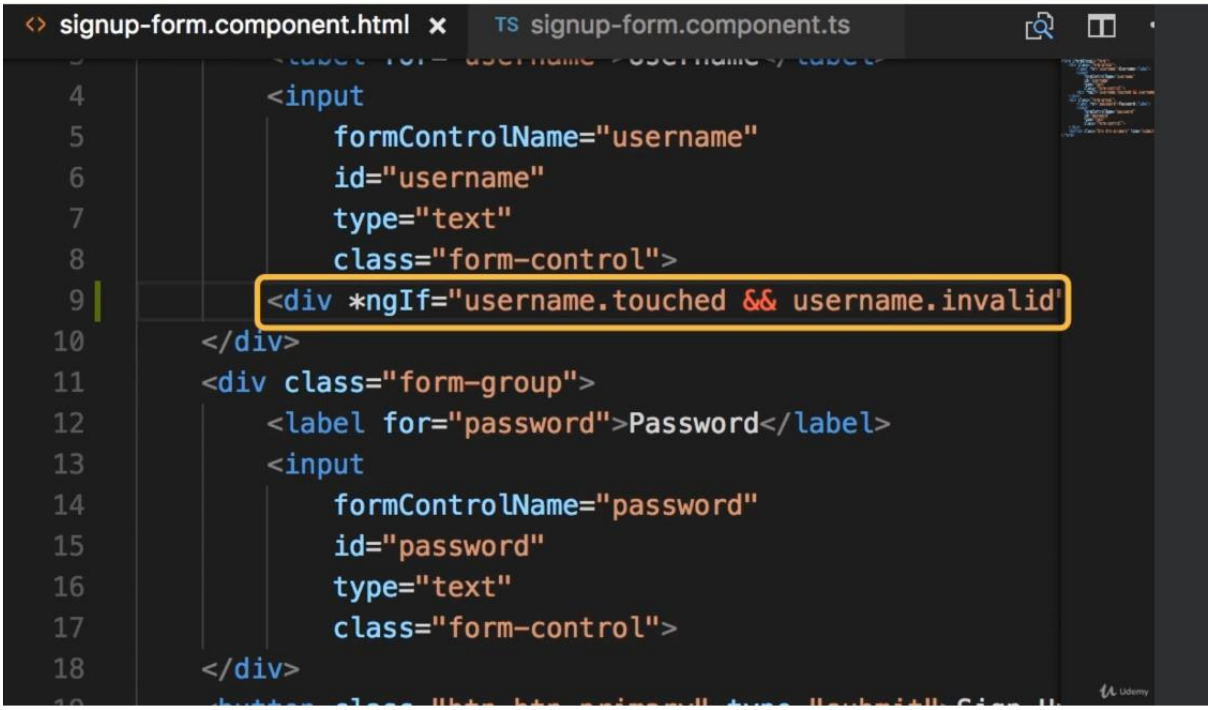
49. Adding validation


signup-form.component.html
signup-form.component.ts
export class SignupFormComponent {
form = new FormGroup({
username: new FormControl('', Validators.required),
password: new FormControl('', Validators.required)
});
required (method) Validators.required(contr…
requiredTrue
asyncValidator?: AsyncValidatorFn |
AsyncValidatorFn[]
): FormControl

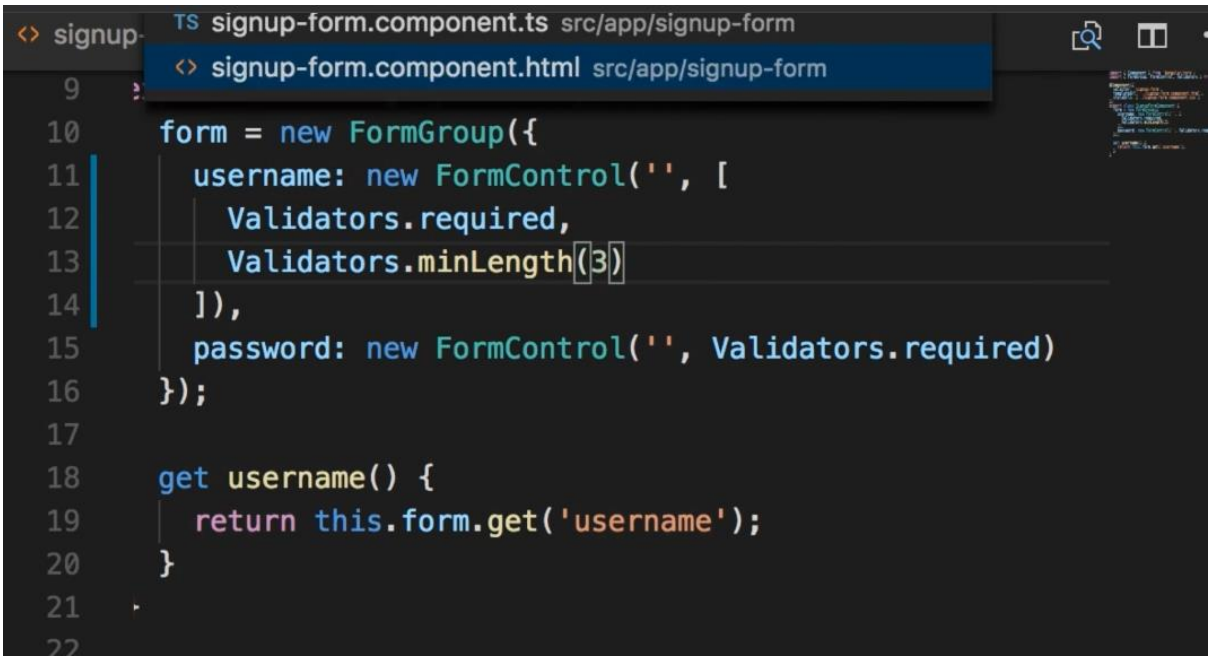

signup-form.component.html
signup-form.component.ts
<input
formControlName="username"
id="username"
type="text"
class="form-control">
<div *ngIf="form.get('username').touched &&"

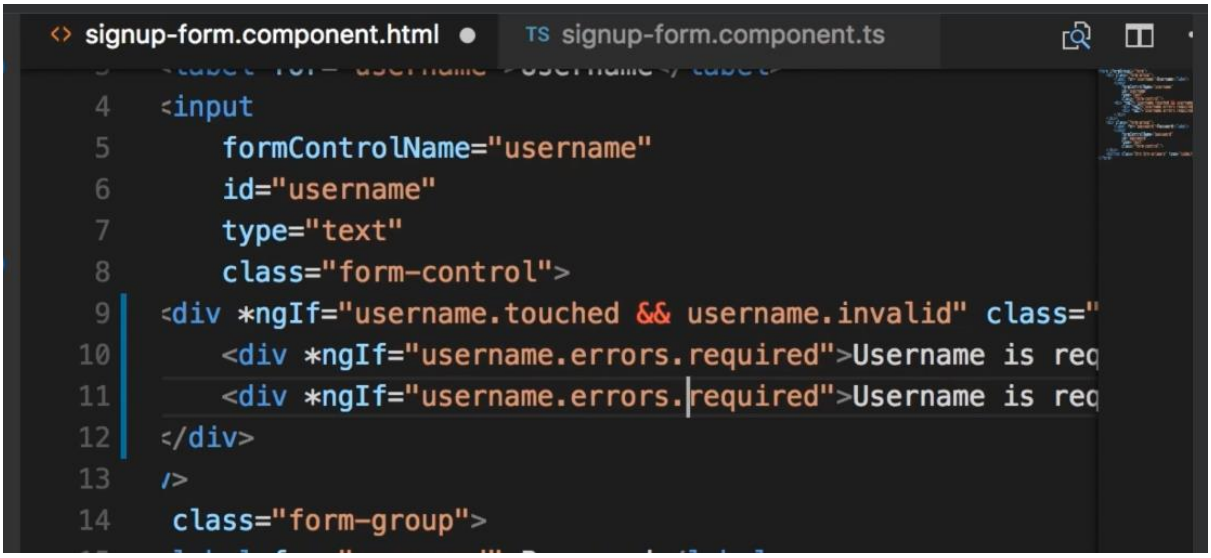
<label for="password">Password</label>
<input
formControlName="password"
id="password"
type="text"
class="form-control">


signup-form.component.ts
export class SignupFormComponent {
form = new FormGroup({
username: new FormControl('', Validators.required),
password: new FormControl('', Validators.required)
});
get username() {
return this.form.get('username');
}
}



50. Specific validation errors



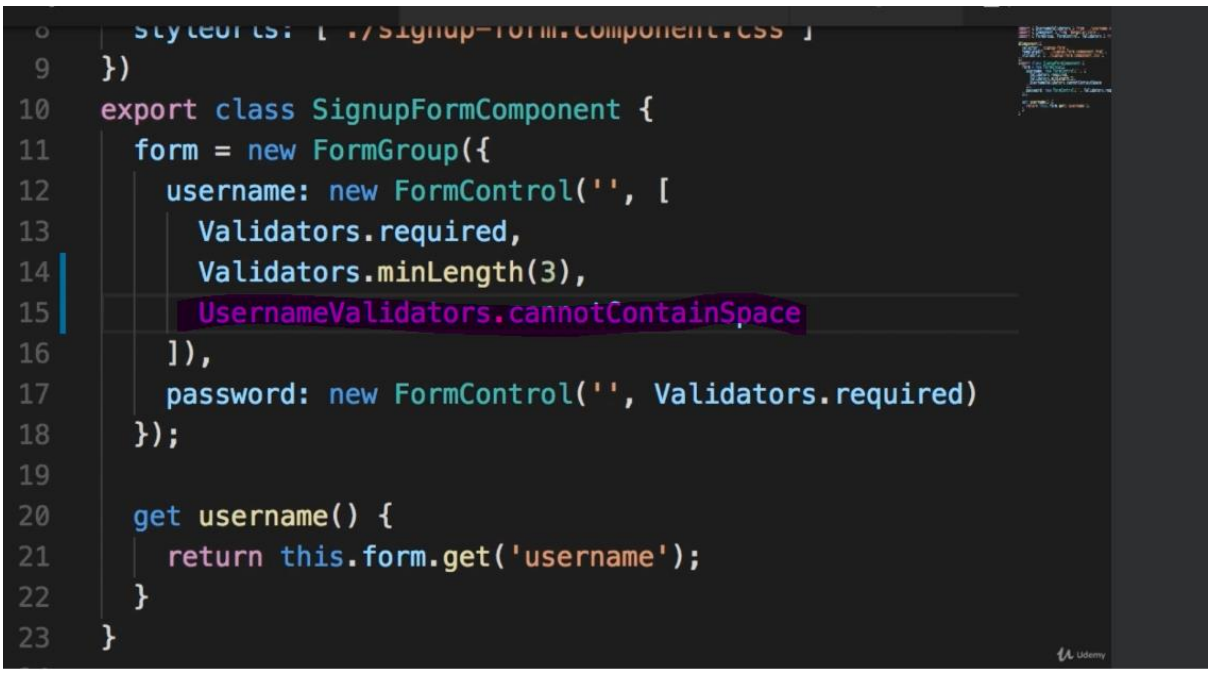


Now, if you're confused about all these classes and directives, let me make it super simple for you. In Angular, we have two classes to keep track of the state of input fields and their validity. One is form control, which represents only one input field and the other is Form group, which represents a group of input fields. Now, when we apply the Ngmodel directive on an input field, Angular automatically creates a form control object and associates that with that input field. So this is form control and ngmodel. Now form group class is used to represent an entire form and optionally groups within a form. As we explained before, we have a directive called NG form that is automatically applied to all form elements. So Angular automatically applies the NG form directive to your forms, and this will internally create a form group object and associate it with your form. And with this form group object, we can track the state changes of the form and its validity. Now, if you have a complex form with multiple subgroups, we can optionally apply the Ngmodel directive on a subgroup. And this directive, similar to the form Directive, will also create a form group object for that group. Now you might be curious what is the difference between Ngform and Ngmodel group? The difference is that the Ngform directive exposes an output property which you saw earlier that is now submit, and we use this to handle the submit event of forms and to model group doesn't have that output property because it doesn't make sense to submit a part of a form. So this is all about form control, form group and the related directives. So in the last lecture we added this required validator here. Now let me show you how to add multiple validators. So as you saw earlier, the second parameter of the constructor of the form control class requires either a validator function or an array of validator functions. So here, if you want to have multiple validators, you can add them in an array like this. So required. I'm also going to apply validators min length of three. Note that here when we call this min length method, it returns a validator function. So we're not calling this method in the sense of performing some kind of validation. We're calling this with an argument to get a validator function. Okay, so here are multiple validators. Now let's go back to our template and display specific error messages. So this is the div for displaying validation errors. I'm going to remove this error message here. Cut. Inside this div we're going to have multiple divs each for a different kind of validation error, exactly like what we did in the last section when we built template driven forms. So we need two divs. One with the error message. Username is required and we want to render this. Only if this username has errors and

required. So if this expression evaluates to truthy, then we're going to display this validation message. Now I'm going to duplicate this for the second error message. I'm going to change this to min length. And the error message should be username should be minimum three characters. Or we can make this dynamic. As you learned in the last section. Add interpolation here and here we type username that errors that minlength dot required length. All right, let's test this. So back in the browser. I put the focus here tab away. We get the first error. Username is required. Now I type only one character and here we get the second error. Username should be minimum three characters. Beautiful. So to assign multiple validators to a form control, simply pass them using an array.

51. Implementing custom validation

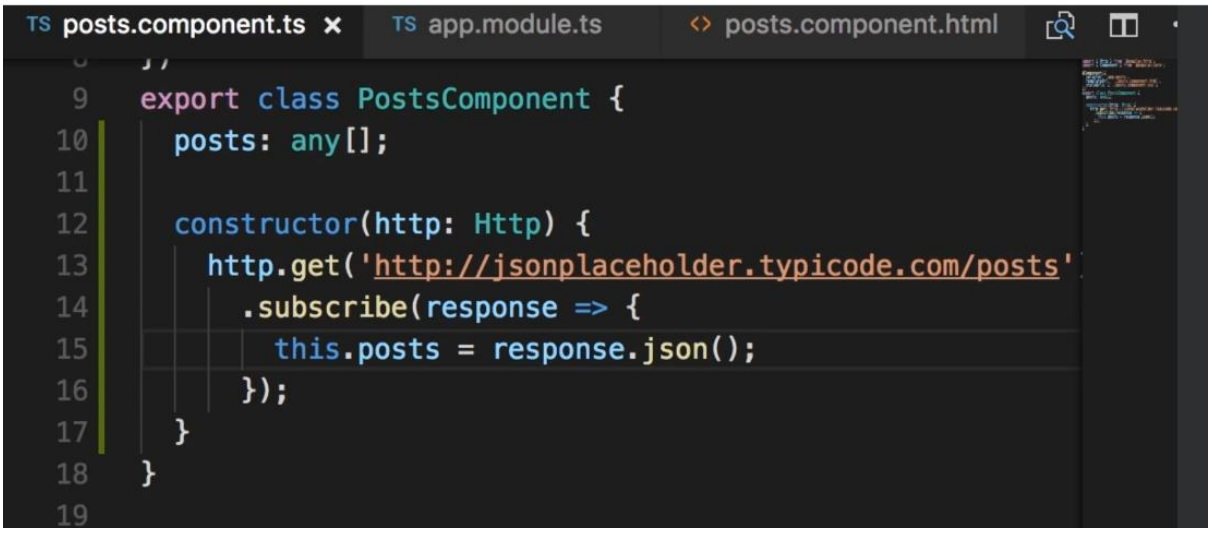
```
1  import { AbstractControl, ValidationErrors } from '@angu
2
3  export class UsernameValidators {
4      static cannotContainSpace(control: AbstractControl)
5          if ((control.value as string).indexOf(' ') >= 0)
6              return { cannotContainSpace: true };
7
8          return null;
9      }
10 }
```



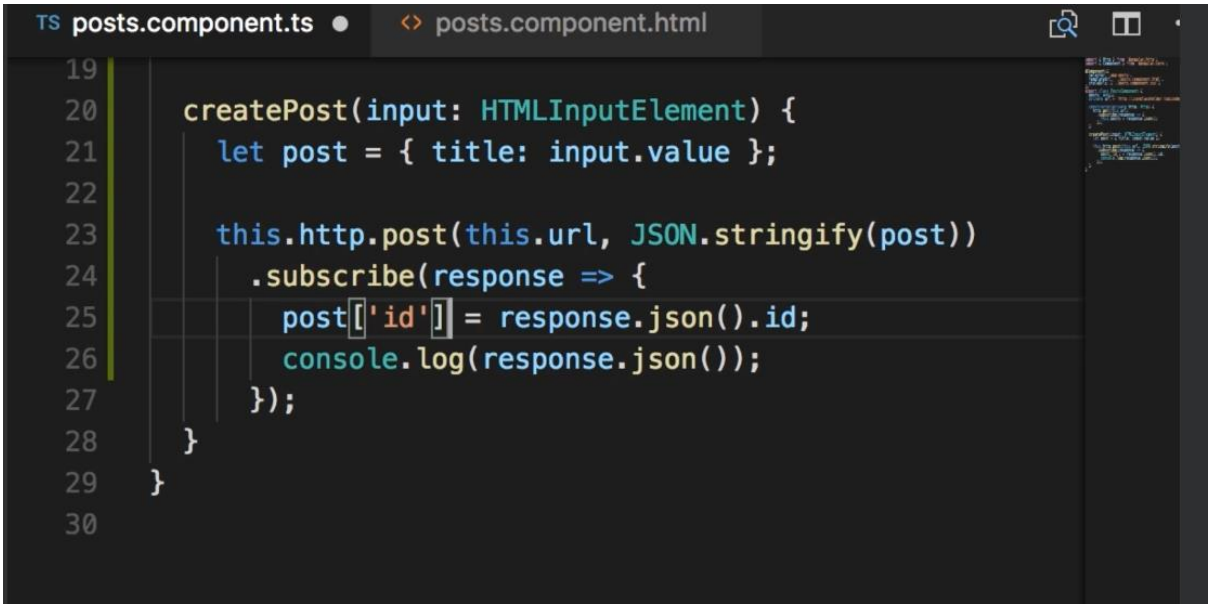
```
9  })
10 export class SignupFormComponent {
11   form = new FormGroup({
12     username: new FormControl('', [
13       Validators.required,
14       Validators.minLength(3),
15       UsernameValidators.cannotContainSpace
16     ]),
17     password: new FormControl('', Validators.required)
18   });
19
20   get username() {
21     return this.form.get('username');
22   }
23 }
```

CONSUMING HTTP SERVICES

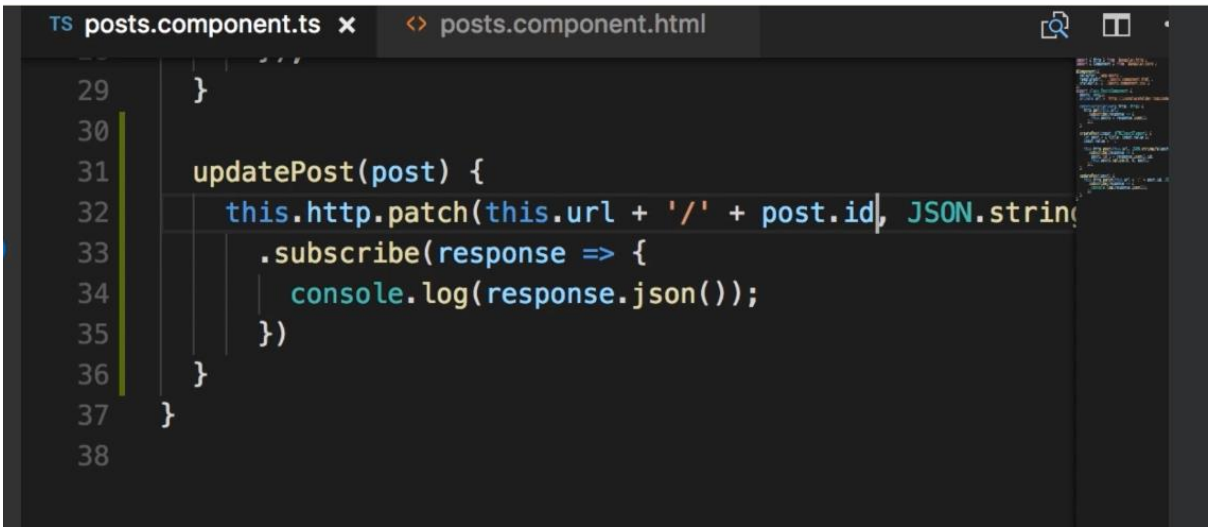
52. Getting Data

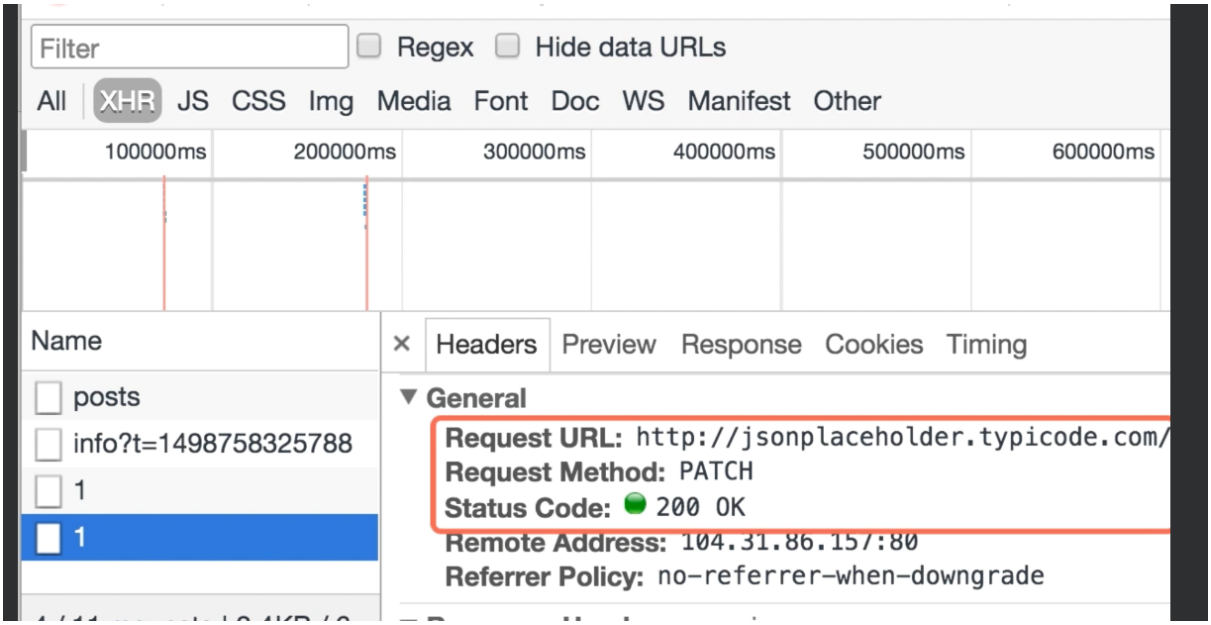


53. Creating data

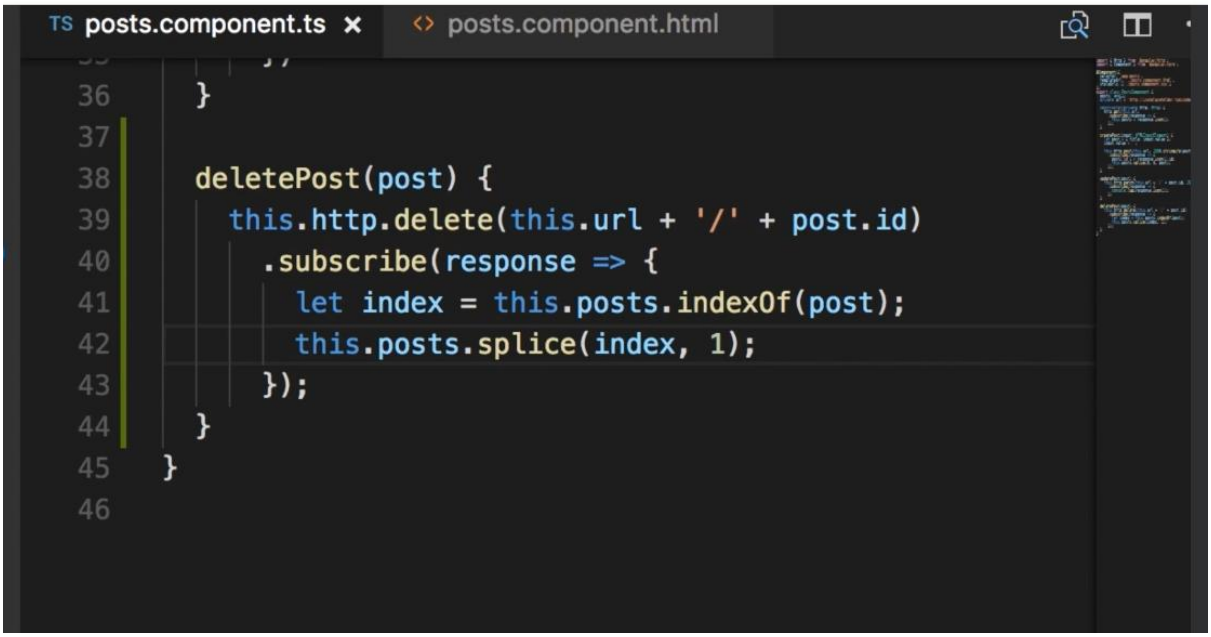


54. Updating data



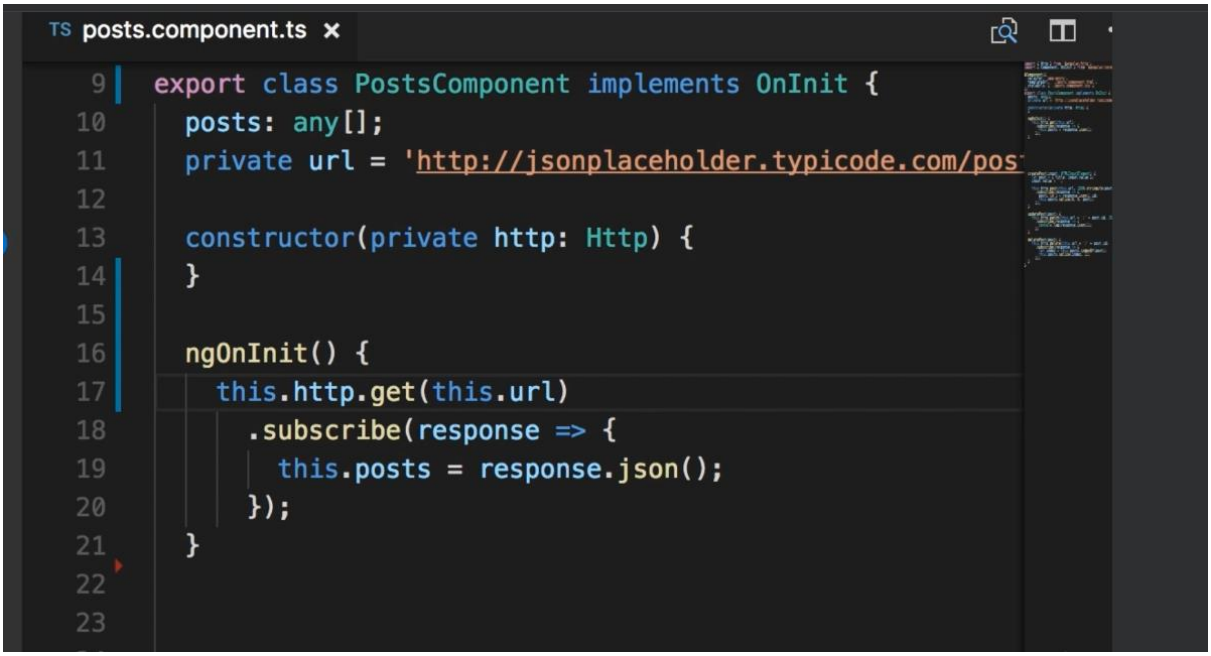


55. Delete data



56. OnInit interface

All right. Now let's take a look at an issue with this implementation. So here in the constructor of Post's component, we're calling the get method of the Http object to get the posts from the server as a best practice. Constructors should be very small and lightweight, so it shouldn't perform expensive operations like calling the server. Then you might ask when can we get these posts from the server? Well, components in Angular have what we call lifecycle hooks. So there are special methods that we can add to our component, and Angular will automatically call these methods at specific times during the lifecycle of a component. For example, when Angular creates a component, renders it creates and renders its children, or when it destroys a component. These are examples of life cycle events. At this point, Angular will call specific methods in our component if they are defined. One of these methods is ng on init. So earlier in the course when we created a component with angular CLI, you notice that all our components had something like this. Implements on init. So on Angular website, let's take a look at this on init interface. You can see this interface is defined in angular core library and this is the shape of this interface. So this interface declares a method called NG on init, which takes no parameters and returns void. And this is the method that Angular will call when it initializes our component. So when we add implements on init to the declaration of a class, we're telling TypeScript compiler that this class should conform with the structure of the Oninit interface. In other words, it should have a method called NG on init. Now here we have a compilation error because we have not imported on init on the top. So. Let's import this. Now we have another compilation error. Class post component incorrectly implements interface on init. Property ng on init is missing in type post component. This error message has a problem because NG on init is a method, not a property. So if we simply add a method here ng on init. You can see the compilation error is gone. So on. Init is an interface that we refer to as a lifecycle hook. There are multiple lifecycle hooks in Angular. For example, we have on changes do check after content in it and so on. Each of these interfaces declare a method with the same name, prefixed with NG. So for on init we have a method called NG on init. Now technically we don't necessarily have to add this implements on init on the top as long as we have a method called ng on init defined in our class. Angular will automatically call this when it initializes our component, but we use implements keyword to add compile time checking. So when we add implements on init TypeScript compiler ensures that we have a method called NG on init. Now I'm going to get all the code in the constructor. Cut it and simply move it here. Of course, we need to add this here because Http is no longer a parameter to this method. So here is the lesson. Do not call Http services in the constructor of your component. If you need initialization, use NG on init method.



57. Ng container

58. Ng container

59. Ng container

60. Ng container

61. Ng container

62. Ng container

63. Ng container

64. Ng container

65. Ng container

66. Ng container

67. Ng container

68. Ng container

69. Ng container

70. Ng container

71. Ng container

72. Ng container

73. Ng container

74. Ng container