**Angular 5**

**Section 4**

**Section 4: Lecture 55 //Understanding Angular Error Messages**

No need for notes

**Section 4: Lecture 56 //Debugging code in the browser using Source maps**

1. Go into sources and web pack to access various files in your angular application and also check their values.
2. You can also debug line by line here.

Section 4: Lecture 57 //Using Augury to dive into Angular Apps

1. Google Angular Augury - it’s a tool for debugging.
2. Once you click on install it will add the Augury extension to your crome browser.

**Section** **5**

**Section 5: Lecture 59 //Module Introduction**

1. Understanding Components and Databinding – now we will dive deeper into components and data binding.
2. We will understand deeper connection between components and databinding.
3. Then we will modify our app.

**Section 5: Lecture 60 //Splitting app into components**

1. We need to split the app into components to reuse the components as our app grows bigger and this also focuses on separation of concerns and reusability of various components.
2. Let’s create a new cockpit component:

ng g c cockpit --spec false

1. Let’s create one more component i.e. server element:

ng g c server-element –spec false

1. Add below code to cockpit.component.html

<div class="container">

<div class="row">

<div class="col-xs-12">

<p>Add new Servers or blueprints!</p>

      <label>Server Name</label>

      <input type="text" class="form-control" [(ngModel)]="newServerName">

      <label>Server Content</label>

      <input type="text" class="form-control" [(ngModel)]="newServerContent">

      <br>

      <button class="btn btn-primary"

          (click)="onAddServer()">Add Server</button>

      <button class="btn btn-primary"

          (click)=" onAddBlueprint()">Add Server Blueprint</button>

</div>

</div>

</div>

1. Add below code to cockpit.component.ts
2. import { Component, OnInit } from '@angular/core';
3. @Component({
4. selector: 'app-cockpit',
5. templateUrl: './cockpit.component.html',
6. styleUrls: ['./cockpit.component.css']
7. })
8. export class CockpitComponent implements OnInit {
9. //serverElements = [];
10. newServerName = '';
11. newServerContent = '';
13. constructor() { }
14. ngOnInit() {
15. }
16. onAddServer(){
18. this.serverElements.push({
19. type: 'server',
20. name: this.newServerName,
21. content: this.newServerContent
22. });
23. }
25. onAddBlueprint(){
26. this.serverElements.push({
27. type: 'blueprint',
28. name:this.newServerName,
29. content: this.newServerContent
30. });
31. }
32. }

6. Put below code in server-element.component.html:

<div

class="panel panel-default"

>

<div class="panel-heading">{{ element.name }}</div>

<div class="panel-body">

<p>

<strong \*ngIf="element.type === 'server'" style="color: red">{{ element.content }}</strong>

<em \*ngIf="element.type === 'blueprint'">{{ element.content }}</em>

</p>

</div>

</div>

7. Add below code to app.comonent.ts :

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

//styleUrls: ['./app.component.css']

styles: [`

h3{

color: dodgerblue;

}

`]

})

export class AppComponent {

serverElements = [];

name = '';

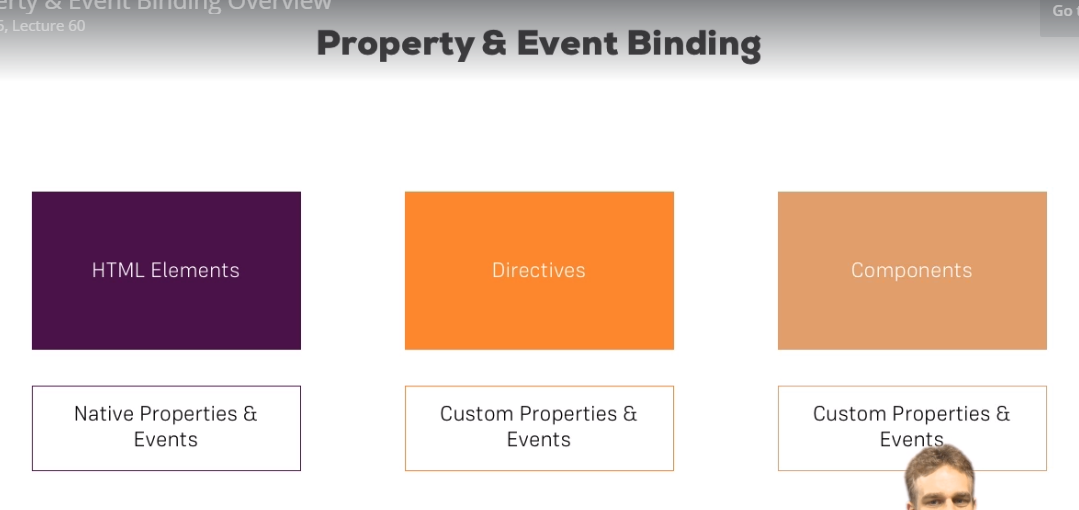
}

8. Below code would be there in app.compomnent.html



**Section 5: Lecture 61 //Property and Event Binding overview**

1. In the last lecture we split up our app into components, but one issue is that we need to pass the data into these components.
2. In the event binding, when we click to a button something happens i.e. an event gets emitted to which we are able to listen, and we can use that data with **$event.**
3. Now we need the above behaviour from our components i.e. we need to send data into a component or receive data/event from the component.
4. We need to send and receive an event, and Angular gives us the great tools to implement this flow.
5. We can use property and event binding not only in the HTML elements but also in Directives and finally we can also use this in our own components, as shown in the screenshot below:

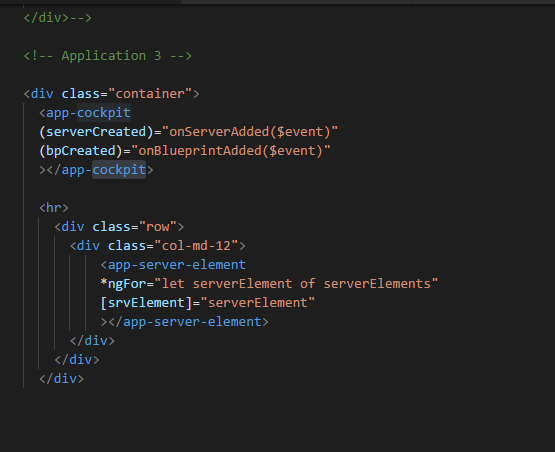


1. We can have our own custom properties and custom events. We can emit our own events.

**Section 5: Lecture 62 //Binding to custom properties**

1. Now, we have learnt that we can use property binding to bind to our custom properties, properties of our own components.
2. Add the below code in server-element.component.ts :
3. import { Component, OnInit } from '@angular/core';
4. @Component({
5. selector: 'app-server-element',
6. templateUrl: './server-element.component.html',
7. styleUrls: ['./server-element.component.css']
8. })
9. export class ServerElementComponent implements OnInit {
10. element: {type: string, name: string, content: string};
11. constructor() { }
12. ngOnInit() {
13. }
14. }
15. element is the property of server-element.component.ts, so, we cannot access it from outside.
16. Add below code in app.component.ts :
17. import { Component } from '@angular/core';
18. @Component({
19. selector: 'app-root',
20. templateUrl: './app.component.html',
21. styleUrls: ['./app.component.css']
22. // styles: [`
23. // h3{
24. // color: dodgerblue;
25. // }
26. // `]
27. })
28. export class AppComponent {
29. serverElements = [{type: 'server', name: 'Testserver', content: 'Just a test!'}];
30. name = '';
31. }

5. We have app.component.ts as shown above; now we will have app-server-element defined in the app.component.html as shown below:



1. We need to access app.component.html from outside; it would be great if we are able to bind to it. In true sense all the properties of a component are only accessible only inside that component and we also don’t want all our properties bendable from outside – we would be very explicit about which property we want to expose.
2. Now, if we want to allow parent component to be able to bind to this property i.e. element property of server-element.component.ts - then we need to add something to that element property. We need to add a decorator. We know that the decorators era not only available for the classes. The decorator we need to add here is **@Input() element**.
3. Now, to allow the parent component to be able to bind to this property, you need to add a decorator; as we know that the decorators are not only available for the classes. **@Input()** now needs to be imported from ‘@angular/core’
4. Now, we are successfully exposing this property to the world i.e. element property.

Below is the decorator we need to add:

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

@Input() element: {type: string, name: string, content: string};

1. So the contents of the server-element.component.ts become as below:
2. import { Component, OnInit, Input } from '@angular/core';
3. @Component({
4. selector: 'app-server-element',
5. templateUrl: './server-element.component.html',
6. styleUrls: ['./server-element.component.css']
7. })
8. export class ServerElementComponent implements OnInit {
9. @Input() element: {type: string, name: string, content: string};
10. constructor() { }
11. ngOnInit() {
12. }
13. }

**Section 5: Lecture 63 //Assigning an Alias to Custom Properties**

1. In the last lecture we learnt how to bind our own properties with @Input() decorator.
2. You can also give the name in the parenthesis of the Input that you want to use outside i.e. when this property is used by some other component as below :

@Input('srvElement') element: {type: string, name: string, content: string}; //srvElement is an alias to the property named here for element

1. Now, here 'srvElement' will work and element will no longer work.
2. Above is the way in which we can assign an alias to the any property of a component.
3. Now the code of server-element.component.ts becomes like below:
4. import { Component, OnInit, Input } from '@angular/core';
5. @Component({
6. selector: 'app-server-element',
7. templateUrl: './server-element.component.html',
8. styleUrls: ['./server-element.component.css']
9. })
10. export class ServerElementComponent implements OnInit {
11. @Input('srvElement') element: {type: string, name: string, content: string};
12. constructor() { }
13. ngOnInit() {
14. }
15. }
16. And the code of app.component.html becomes as below:
17. <div class="container">
18. <app-cockpit></app-cockpit>
19. <hr>
20. <div class="row">
21. <div class="col-md-12">
22. <app-server-element
23. \*ngFor="let serverElement of serverElements"
24. [srvElement]="serverElement"
25. ></app-server-element>
26. </div>
27. </div>
28. </div>

**Section 5: Lecture 64 //Binding to the custom events**

1. We learnt how to pass information from one component down (i.e. to the child component) to another component i.e. we learnt how to pass information from parent component down to other component if something is changed.
2. Now, we will learn how to pass the communication from the child component to the parent component. Here the parent component which implements the other component i.e. child component.
3. Here, from the child component i.e. cockpit.component.ts to app.component.ts that something is changed in cockpit.component.ts i.e. child to parent component.
4. Here we want to inform our parent component i.e. the app.component in this case new server was created, because the code which would have simply executed is commented out.
5. Now in the cockpit.component we want to implement the above methods.
6. Update the code of app.component.ts as below, now in onServerAdded and onBlueprintAdded, we expect to get event i.e. serverData and bluePrintData repectively.
7. import { Component } from '@angular/core';
8. @Component({
9. selector: 'app-root',
10. templateUrl: './app.component.html',
11. styleUrls: ['./app.component.css']
12. // styles: [`
13. // h3{
14. // color: dodgerblue;
15. // }
16. // `]
17. })
18. export class AppComponent {
19. serverElements = [{type: 'server', name: 'Testserver', content: 'Just a test!'}];
20. name = '';
21. onServerAdded(serverData: {serverName: string, serverContent: string}){
23. this.serverElements.push({
24. type: 'server',
25. name: serverData.serverName,
26. content: serverData.serverContent
27. });
28. }
30. onBlueprintAdded(bluePrintData: {serverName: string, serverContent: string}){
31. this.serverElements.push({
32. type: 'blueprint',
33. name: bluePrintData.serverName,
34. content: bluePrintData.serverContent
35. });
36. }
37. }

7. Update the code of app.component.html as below:

<div class="container">

<app-cockpit

(serverCreated)="onServerAdded($event)"

(bluePrintCreated)="onBluePrintAdded($event)"

></app-cockpit>

<hr>

<div class="row">

<div class="col-md-12">

<app-server-element

\*ngFor="let serverElement of serverElements"

[srvElement]="serverElement"

></app-server-element>

</div>

</div>

</div>

8. Now, we need to make both serverCreated and blueprintCreated as the events that we will emit from cockpit.component.ts, this we will do by assigning the property to new EventEmitter<> ; There are greater than and smaller than sign behind the EventEmitter because the event emitter is a generic type. So here we need to define the type of event we are going to emit. EventEmitter should be imported from ‘@angular/core’

serverCreated = new EventEmitter<{serverName: string, serverContent: string}>();

blueprintCreated = new EventEmitter<{serverName: string, serverContent: string}>();

7. Now, we need to add the decorator to make the property listenable from outside i.e. @Output(); Import @Output() from ‘@angular/core’ below is the code for the same with the decorator:

@Output() serverCreated = new EventEmitter<{serverName: string, serverContent: string}>();

@Output() blueprintCreated = new EventEmitter<{serverName: string, serverContent: string}>();

8.There we will use addServer() methodused to emit the event. So, here we are emitting our own events and we are passing our own data. Contents of cockpit.component.ts will become:

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

@Component({

selector: 'app-cockpit',

templateUrl: './cockpit.component.html',

styleUrls: ['./cockpit.component.css']

})

export class CockpitComponent implements OnInit {

//serverElements = [];

@Output() serverCreated = new EventEmitter<{serverName: string, serverContent: string}>();

@Output() blueprintCreated = new EventEmitter<{serverName: string, serverContent: string}>();

newServerName = '';

newServerContent = '';

constructor() { }

ngOnInit() {

}

onAddServer(){

this.serverCreated.emit({serverName : this.newServerName,

serverContent: this.newServerContent});

}

onAddBlueprint(){

this.blueprintCreated.emit({serverName : this.newServerName,

serverContent: this.newServerContent});

}

}

9. Contents of app.component.ts will become:

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

// styles: [`

// h3{

// color: dodgerblue;

// }

// `]

})

export class AppComponent {

serverElements = [{type: 'server', name: 'Testserver', content: 'Just a test!'}];

name = '';

onServerAdded(serverData: {serverName: string, serverContent: string}){

this.serverElements.push({

type: 'server',

name: serverData.serverName,

content: serverData.serverContent

});

}

onBlueprintAdded(blueprintData: {serverName: string, serverContent: string}){

this.serverElements.push({

type: 'blueprint',

name: blueprintData.serverName,

content: blueprintData.serverContent

});

}

}

10. Content of server-element.component.html will become:

<div

class="panel panel-default"

>

<div class="panel-heading">{{ element.name }}</div>

<div class="panel-body">

<p>

<strong \*ngIf="element.type === 'server'" style="color: red">{{ element.content }}</strong>

<!-- <strong \*ngIf="element.type === 'blueprint'" style="color: blue">{{ element.content }}</strong> -->

<em \*ngIf="element.type === 'blueprint'" style="color: blue">{{ element.content }}</em>

</p>

</div>

</div>

11. Here we made our components communicable. Component communication is very important in any app.

**Section 5: Lecture 65 //Assigning an Alias to custom events**

1. Just like @Input(), you can also assign alias to @Output() as shown in the example below:

@Output('bpCreated') blueprintCreated = new EventEmitter<{serverName: string, serverContent: string}>();

1. App.component.html will get updated as below:
2. <div class="container">
3. <app-cockpit
4. (serverCreated)="onServerAdded($event)"
5. (bpCreated)="onBlueprintAdded($event)"
6. ></app-cockpit>
8. <hr>
9. <div class="row">
10. <div class="col-md-12">
11. <app-server-element
12. \*ngFor="let serverElement of serverElements"
13. [srvElement]="serverElement"
14. ></app-server-element>
15. </div>
16. </div>
17. </div>

**Section 5: Lecture 66 //Custom properties and event binding summary**

1. We will use services when the distance between the component will grow too much i.e. there are chains of components communicating.

**Section 5: Lecture 67 //Understanding view encapsulation**

1. Now the properties applied in the app.component.css file will not work for the child components as it clearly belongs to the app.component.html template. The blue color was defined in the .css file of the app.component. Now, it is not defined inside the cockpit.component.
2. The behaviour that the properties defined in the CSS file of a particular component will be applied to that component only – is the behaviour given to us by angular and not default behaviour.
3. This is the default behaviour of the view encapsulation in Angular.
4. Now, we will copy the blue color for paragraph in the server-element.component
5. The styles defined inside the particular component are defined in the .css file of that component; and those will be applied to that particular component only.

**Section 5: Lecture 68 //More on view encapsulation**

1. In the last lecture we learnt that how angular encapsulates your styles, now you can overwrite this encapsulation though.
2. If we add the below code in a component, then that component will not use view encapsulation; ViewEncapsulation is imported from the ‘@angular/core’:

encapsulation: ViewEncapsulation.None

1. Once we use ViewEncapsulation.None, for any component the encapsulated IDs will not be added for that particular component. As we know that the encapsulation IDs are added by angular for each component by angular by default.
2. Now the code of server-element.component.ts becomes as shown below:

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

@Component({

selector: 'app-server-element',

templateUrl: './server-element.component.html',

styleUrls: ['./server-element.component.css'],

encapsulation: ViewEncapsulation.None

})

export class ServerElementComponent implements OnInit {

@Input('srvElement') element: {type: string, name: string, content: string};

constructor() { }

ngOnInit() {

}

}

1. The default is Emulated which means that only your component will receive the style which you define for that component.
2. Now, we can also use ViewEncapsulation.Native which uses shadow DOM technology //this should give same result as ViewEncapsulation.Emulated but most browsers don’t support it.
3. But, be aware we can also choose to .None and .Native too.
4. This is how you can change it and how view encapsulation works.

**Section 5: Lecture 69 //Using local references in templates**

1. In the cockpit.component we are using @Output and @Input to move data around – our own custom property and event binding.
2. Now, in the cockpit we are using the 2 way databinding to get the server name and content; now there is nothing wrong with that to use the 2 way databinding since I only want to save or use the data only when I click the button i.e. Add Server or the Add server blue print button. It would be enough to get the value of the input at this point of time.
3. There is the other option which we have is that we can use the local references only inside our HTML template, but not inside the typescript by using the # key as shown below; here we are putting a local reference on the input element, however, we can put local reference on any of the HTML elements.
4. As, shown below we can add the local reference by adding a # tag. Here, we can add the #serverNameInput – this doesn’t hold the reference to the value but it holds the reference to the element.
5. Now, in onAddServer(serverNameInput) – here we added serverNameInput
6. Local reference can only be used inside the template and not inside the .ts file.
7. type="text"
8. class="form-control"
9. #serverNameInput>
10. Now, we can use any local reference anywhere in the app, updated code in cockpit.component.html is shown below:
11. <div class="container">
12. <div class="row">
13. <div class="col-xs-12">
14. <p>Add new Servers or blueprints!</p>
15. <label>Server Name</label>
16. <!-- <input type="text" class="form-control" [(ngModel)]="newServerName"> -->
17. <input
18. type="text"
19. class="form-control"
20. #serverNameInput>
21. <label>Server Content</label>
22. <input type="text" class="form-control" [(ngModel)]="newServerContent">
23. <br>
24. <button class="btn btn-primary"
25. (click)="onAddServer(serverNameInput)">Add Server</button>
26. <button class="btn btn-primary"
27. (click)="onAddBlueprint(serverNameInput)">Add Server Blueprint</button>
28. </div>
29. </div>
30. </div>
31. Updated code of cockpit.component.ts is as below:
32. import { Component, OnInit, Output, EventEmitter } from '@angular/core';
33. @Component({
34. selector: 'app-cockpit',
35. templateUrl: './cockpit.component.html',
36. styleUrls: ['./cockpit.component.css']
37. })
38. export class CockpitComponent implements OnInit {
39. //serverElements = [];
40. @Output() serverCreated = new EventEmitter<{serverName: string, serverContent: string}>();
41. @Output('bpCreated') blueprintCreated = new EventEmitter<{serverName: string, serverContent: string}>();
42. newServerName = '';
43. newServerContent = '';
45. constructor() { }
46. ngOnInit() {
47. }
48. onAddServer(nameInput: HTMLInputElement){
49. console.log(nameInput.value);
50. this.serverCreated.emit({serverName : nameInput.value,
51. serverContent: this.newServerContent});
52. }
54. onAddBlueprint(nameInput: HTMLInputElement){
55. this.blueprintCreated.emit({serverName : nameInput.value,
56. serverContent: this.newServerContent});
57. }
58. }

//with comments

1. nAddServer(nameInput: HTMLInputElement){ //here we get the element with all its properties given by the local variable
2. console.log(nameInput.value); //we can fetch value out of it
3. this.serverCreated.emit({serverName : nameInput.value,
4. serverContent: this.newServerContent});
5. }

8. So, local reference is very good way to get some input from the template and we can use the same inside the template also.

**Section 5: Lecture 70 //Getting Access to the Template& DOM with @ViewChild**