CS572 Modern Web Applications Programming
Maharishi University of Management
Department of Computer Science
Assistant Professor Asaad Saad

PREV NEXT

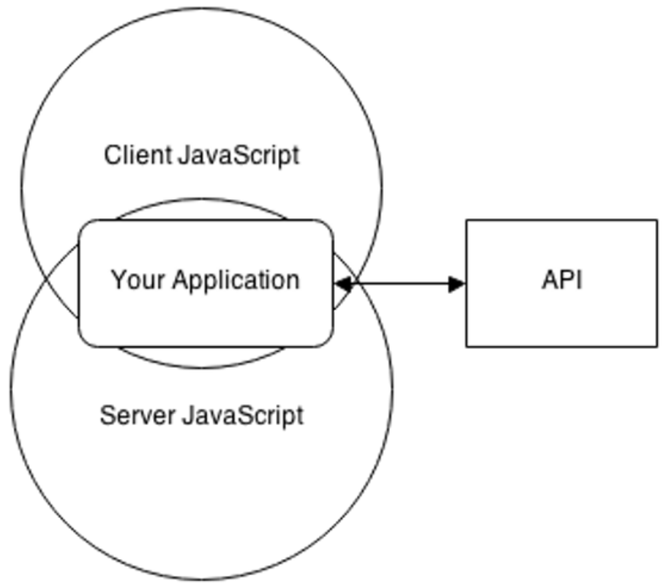
OUTLINE

Search...

Isomorphic JavaScript

In web development, an isomorphic application is one whose code can run both in the **server and the client**.

In SPA applications, the first request made by the web browser is processed by the server while subsequent requests are processed by the client.



http://isomorphic.net

Why Isomorphic?

- **One language** that runs on the client & server
- **SEO-friendly**
  - Google can now crawl JavaScript applications on websites. As of May 23rd, 2014, Google bot executes JavaScript.
- **Speed**
  - Faster to render HTML content, directly in the browser. Better overall user experience.
- **Easier code maintenance**

PREV NEXT

Single Page Application - SPA

A single-page application (SPA) is a web application that fits on a single web page with the goal of providing a user experience similar to that of a desktop application.

In an SPA, either all necessary code – HTML, JavaScript, and CSS – is retrieved with a single page load, or the appropriate resources are dynamically loaded and added to the page as necessary, usually in response to user actions.

The page does not reload at any point in the process, nor does control transfer to another page, although the location hash or the HTML5 History API can be used to provide the perception and navigability of separate logical pages in the application.

Interaction with the single page application often involves dynamic communication with the web server behind the scenes.

https://en.wikipedia.org

Angular

Angular is a framework that will provide us flexibility and power when building our apps (One framework for mobile & desktop).

- Takes advantage of ES6
- Web components
- Framework for all types of apps
- Speed improvements

Do I have to use TypeScript?

No, you don’t have to use TypeScript to use Angular, but you probably should.

- Angular does have an ES5 API.
- Angular is written in TypeScript.

We’re going to use TypeScript in this course because it’s great and it makes working with Angular easier. But again it is not required.

PREV NEXT

OUTLINE

Search...

Build Your Environment

Angular CLI will install and setup the environment for us:

- Setup TypeScript and all Typings (using **`typescript`** and **`typings`**)
- Watch all TS files and ES6 files and convert them to JS (ES5) so all browsers will understand (**`tsc -w`**)
- Bundle all modules and create the bundle.js (using **`Webpack`**)
- Create a NodeJS web server when in Dev mode (using **`light-server`**)
- Push the refreshed code to browser every time you change your code (using **`browser-sync`**)
- Run all above simultaneously (using **`concurrently`**)

PREV NEXT

Angular CLI

You can have an Angular app up and running like this:

```
npm install -g @angular/cli
ng help
ng new my-new-app
ng new my-new-app –d [--dryRun] // no files will be created
ng new my-new-app --skip-install // create files but do not run: npm install
cd my-new-app
ng serve [--host 0.0.0.0 --port 4201] // [-o] to open the browser
```

ng new
The Angular CLI makes it easy to create an application that already works, right out of the box.

ng generate
Generate components, routes, services and pipes with a simple command.

ng serve
Easily put your application in production (Bundler is ready)

PREV NEXT

Angular Packages

Since Angular is very modular framework, these are going to be pulled in as separate modules.

The main packages are:

- `@angular/core`
- `@angular/common`
- `@angular/compiler`
- `@angular/platform-browser`
- `@angular/platform-browser-dynamic`

Other optional packages we'll use:

- `@angular/router`
- `@angular/http`
- `@angular/forms`

Angular Dependencies

We want to use all new ES6 features that aren't yet supported by browsers. This is why we need transpilers, loaders, polyfills, and shims.

These dependencies help provide some functionality for Angular that make our apps better.

- **core-js**: Adds es6 features to browsers that don't have them
- **zone.js**: Creates execution context around my app. Helps with change detection and showing errors. Provides stack traces.
- **rxjs**: (ReactiveX) Libraries that help create asynchronous data streams. Gives us Observables, the preferred way of handling async events in Angular (Promises in AngularJS).

Application

- An Angular Application is nothing more than a **tree of Components**.
- One of the great things about Components is that they're **composable**. This means that we can build up larger Components from smaller ones. The Application is simply a Component that renders other Components.
- Because Components are structured in a parent/child tree, when each Component renders, **it recursively renders its children Components.**

PREV NEXT

What is a Module?

A module is a simple class/object. But this class is decorated with **`@NgModule()`** factory decorator, where we pass metadata/object to it to describe this module and have Angular controls it.

```
@NgModule({
        imports: [ .. ],
        declarations: [ .. ],
        providers: [ .. ],
        bootstrap: [ .. ] })
class AppModule {}
```


PREV NEXT

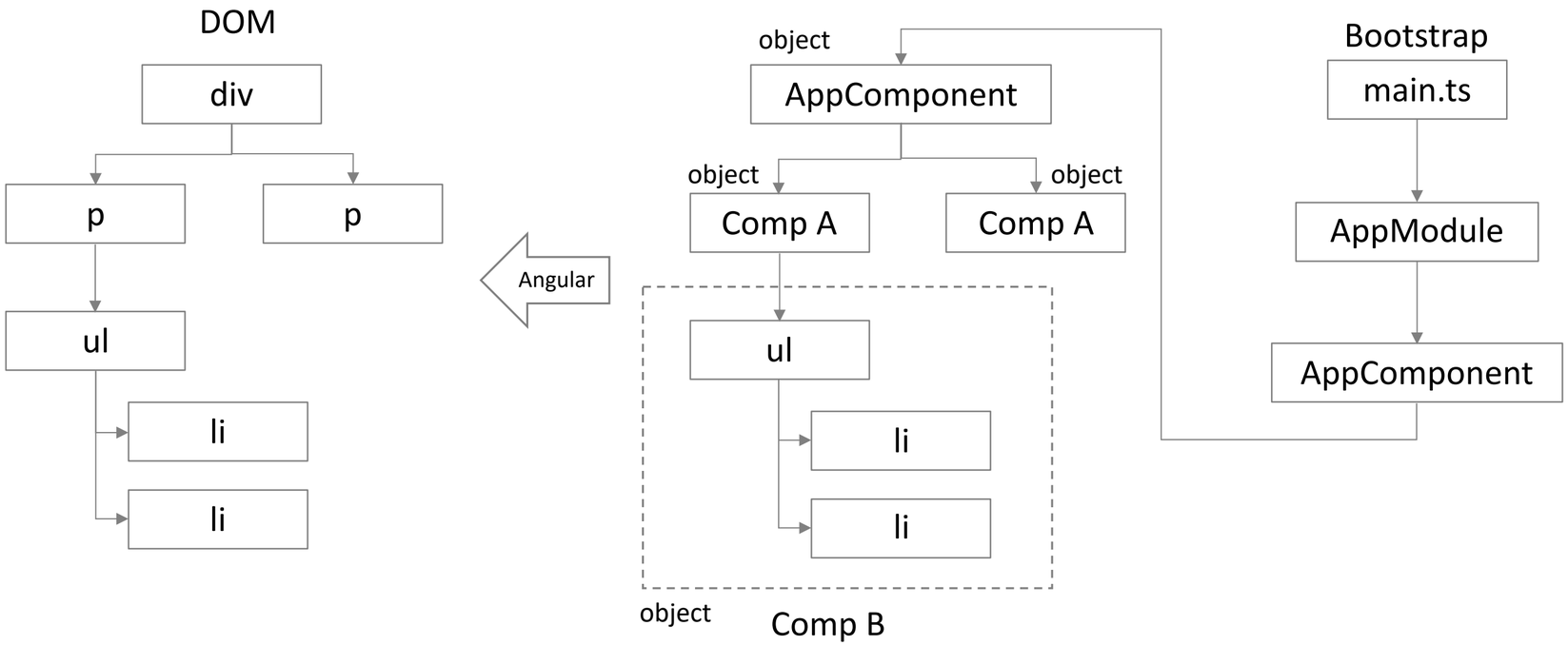
What is a Component?

A component is a simple class/object. But this class is decorated with **`@Component()`** factory decorator, where we pass metadata/object to it to describe this component and have Angular controls it.

```
@Component({
      selector: '',
      template: ''})
class AppComponent {}
```

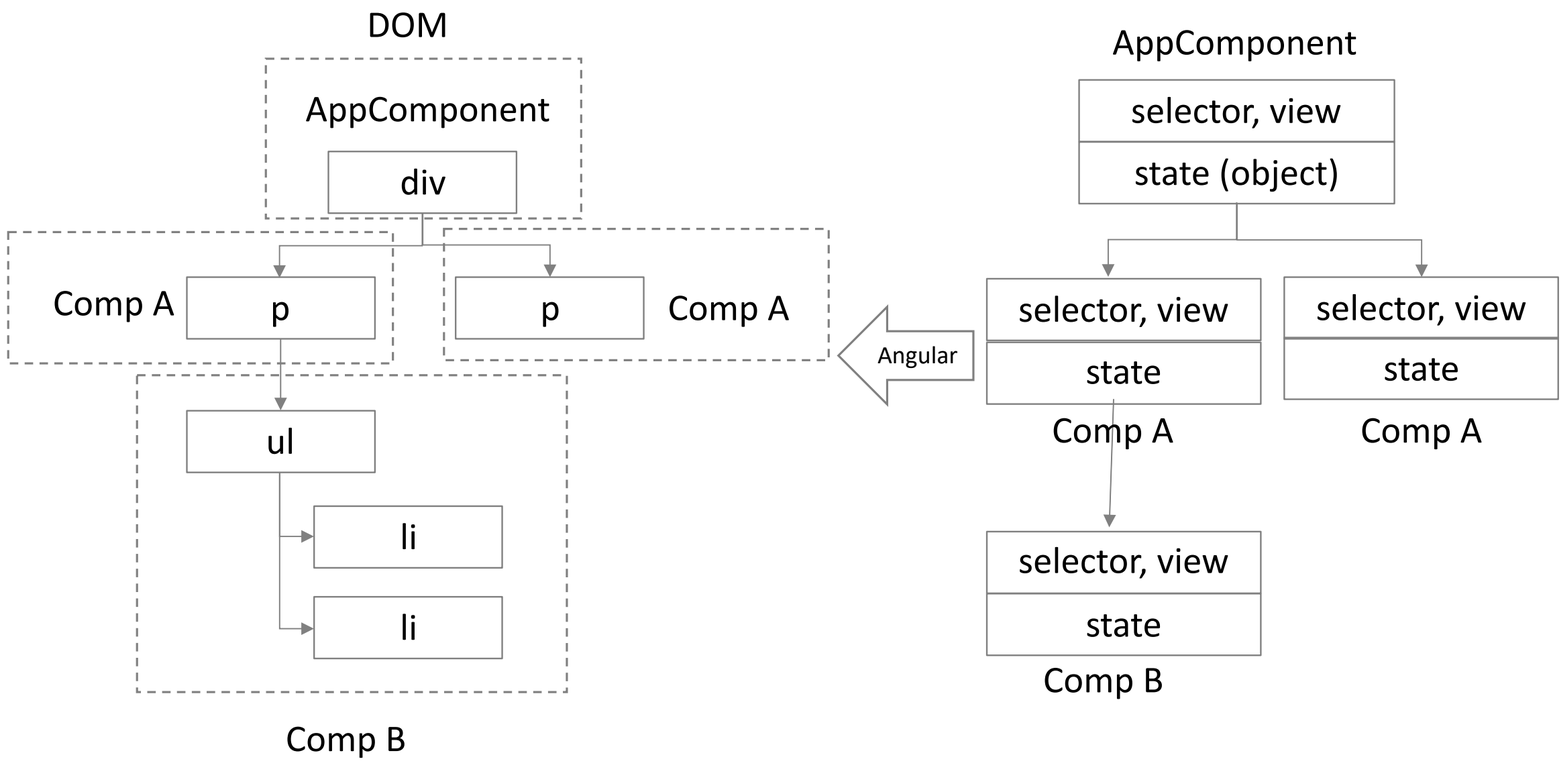
A component represents one element in Angular Application, this element has a tag name (`selector`), and has view/`template` and state/object. The `template` is what being rendered to the DOM by Angular.

Angular Page vs HTML Page

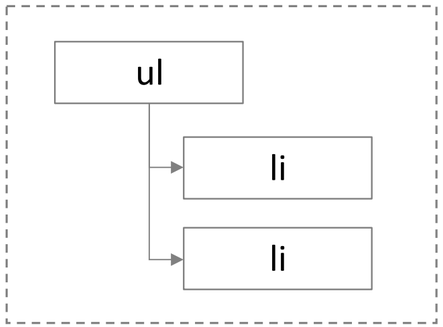


15

Components vs HTML Tags



DOM (template) and Component State



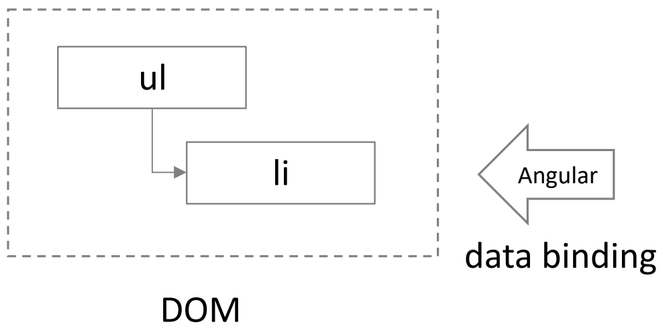
DOM

Angular

```
@Component({
   selector: 'compB',                    selector
   template: `<ul>
                 <li></li>               view
                 <li></li>
             </ul>`})
class ComponentB {}                      state
```

<compB></compB>

Data Binding

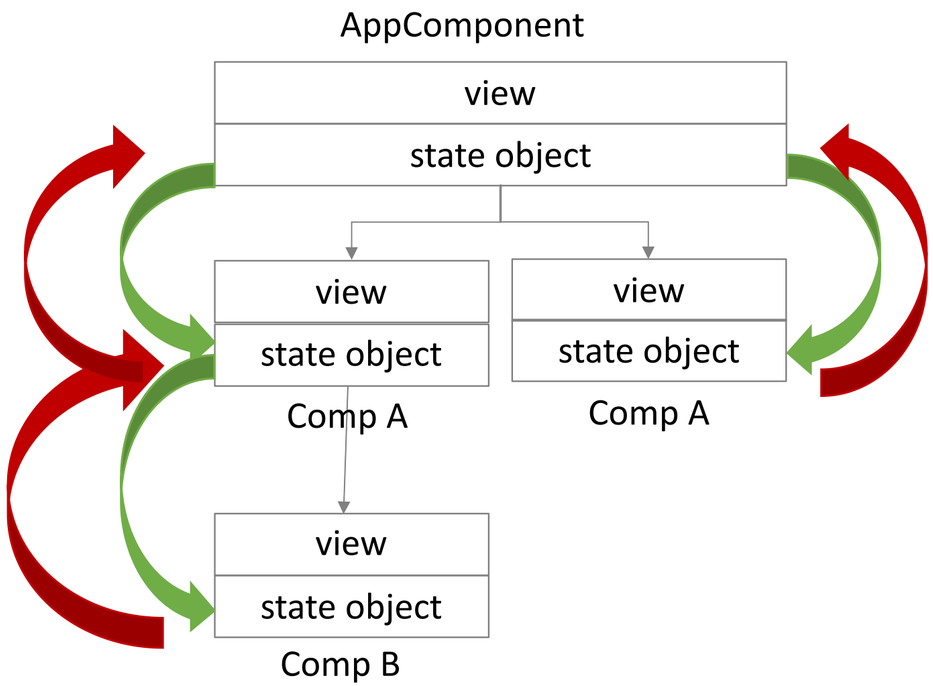



```
@Component({
   selector: 'compB',                          view
   template: `<ul>
                 <li> {{ message }} </li>
              </ul>`})
```

```
class ComponentB {
   message = 'Hello there!';                   state
}
```

- You change the state (`message` property)
- Angular reflects the change to the view
- Angular renders the view to real DOM (one way data binding)
- Notice that changing the DOM won't affect your app state (`message` property)

Communication: Input and Output



Communication between components: a parent component may pass data from their state object to their child state object through **inputs**, a child component may pass data to it's parent's state object through **outputs**.

Passing Input/Output

```
<input name="username" onkeyup="doSomething()" />
```

Using JS to pass input/output to **<input>** Native Component

```
@Component({ template: `<input [name]="field" (keyup)="doSomething()" />`})

class Component {
    field = 'username';
    doSomething(){}
}
```

Using Angular to pass input/output to **<input>** Native Component

```
@Component({ template: `<weather [unit]="tempUnit" (onStorm)="doSomething()"></weather>`})

class Component {
    tempUnit = 'F';
    doSomething(){}
}
```

Using Angular to pass input/output to **<weather>** Custom Component

```
class Weather {
    @Input() unit;
    @Output() onStorm = new EventEmitter();
}
```

Environment Setup - Structure

The directory structure for your app will look like this:

```
|- app/
       |- app.component.ts // main app component
       |- app.module.ts // main app module
       |- main.ts // bootstrap our app

|- index.html
|- package.json          } Angular Application (SPA)

|- tsconfig.json
|- typings.json          } To use TypeScript
```

Getting started

Getting started with Angular requires **three major files:**

- `main.ts`: This is where we bootstrap our app. *This is similar to using ng-app in AngularJS.*
- `app.module.ts`: The top level module for our app. The module defines a certain section of our site.
- `app.component.ts`: The main component that encompasses our entire app.

The reason we separate bootstrapping out into its own file is that we could bootstrap a number of different ways. We're going to bootstrap our app for the browser, but it could be done for mobile, universal, and more.

PREV NEXT

Bootstrapping the App

To bootstrap an Angular application in the JIT mode, you pass a module to bootstrapModule in `main.ts`.

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```

main.ts

This will compile AppModule into a module factory and then use the factory to instantiate the module

1. Define dependent modules (Eager loading)
2. Define components our module consist of (Create instance)
3. Define services (Create instance)

NgModules

NgModules are distribution of Angular components and pipes.

In many ways they are similar to ES6 modules, in that they have declarations, imports, and exports.

Modules can be loaded **eagerly** when the application starts or in a **lazy** way from the router when we need them.

PREV NEXT

OUTLINE

Search...

Modules

@NgModule is the decorator that gives us an Angular module.

A module is the way that we can bundle sections (components) of our applications into a singular focused package.

A module can include many parts including other modules (**imports**), components and/or directives (**declarations**), and services used to access data (**providers**).

Every component belongs to a NgModule.

Each module should represent a feature in our application.

PREV NEXT

Modules

To define a Module we pass the following Metadata to the factory decorator:

- **imports**: Other **modules** (native or custom built) that are parts of our application. (HttpClientModule, FormModule, RouteModule, BrowserModule).
  All these modules and their functionalities will be available to the app.
- **declarations**: These are any **components** or **directives** or **pipes** that you want access to in your application.
- **providers**: Configure dependency injection. These are **services** and used as a singular place to access and manipulate certain data. Services declared here share the same instance between all application unless they are declared in each component, then a new instance of the service will be initialized.
- **bootstrap**

PREV NEXT

app.module.ts

Destructuring

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

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

@NgModule({
        imports: [ BrowserModule ],
        declarations: [ AppComponent ],
        providers: [ userAPI ],
        bootstrap: [ AppComponent ] })

export class AppModule {}
```

Bootstrap and entry components. The top level module will use the bootstrap attribute to tell the module what to use when creating our app. The style guide and conventions prefers to use an **AppComponent** as the top level of our app.

Bootstrap and Entry Components

The bootstrap property defines the components that are instantiated when a module is bootstrapped.

Angular creates a component factory for each of the bootstrap components. And then, at runtime, it'll use the factories to instantiate the components.

PREV NEXT

What are Components?

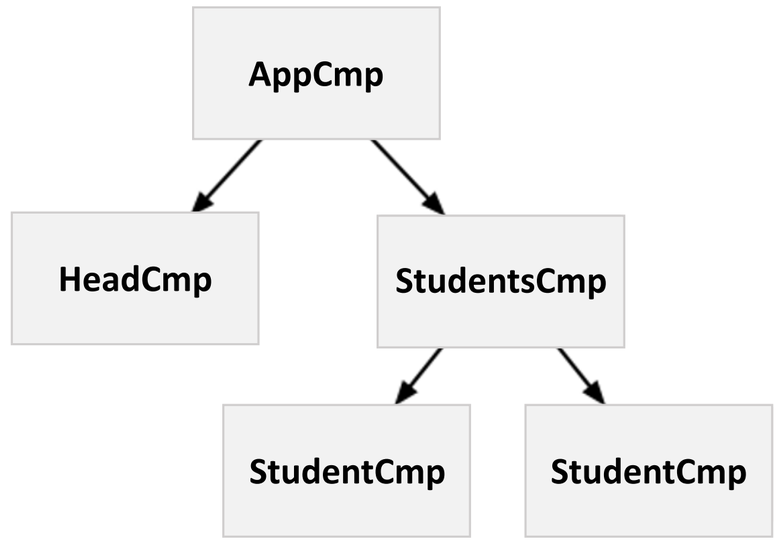
In our Angular apps, we write HTML markup that becomes our interactive application, but the browser understands only limited built-ins markup tags like `<select>` or `<form>` or `<video>` all have functionality defined by the browser.

What if we want to teach the browser **new tags**? What if we wanted to have a **`<weather>`** tag that shows the weather? Or what if we wanted to have a **`<login>`** tag that creates a login panel?

PREV NEXT

Components

To build an Angular application you define a set of components, for every UI element, screen, and route. An application will always have root components that contain all other components.



Creating a new Component

We can build our component from scratch but it's easier to use Angular CLI

```
ng generate component myComponent
ng g c myComponent
```

Notice the changes in `module.ts`

```
ng g c --flat myComponent
```

`--flat` No new folder

```
ng g c --inline-template myComponent
ng g c –t myComponent
```

`-t` No Template file (inline)

```
ng g c --inline-style myComponent
ng g c -s myComponent
```

`-s` No Style file (inline)

```
ng g c --spec myComponent
ng g c --flat –s –t myComponent ––spec false
```

app.component.ts

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

@Component({
        selector: 'my-app',
        template: ` <div class="big">
                        <h1>Welcome to CS572!</h1>
                    </div>
                  `,
        styles: [` .big { font-size: 20px; } `] })

export class AppComponent {}
```

Template

A component must have a template, which describes how the component will be rendered on the page.

You can define the template:

- **inline** using **`template`** property
- **externally** using **`templateUrl`** property

In addition to the template, a component can define styles using:

- **`styles`** property
- **`styleUrls`** property

By default the styles are encapsulated.

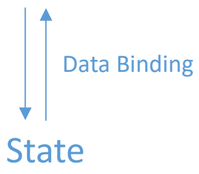
Template Binding

We can pass data from our component class to our template very easily and bind them together by defining a property and access it with string interpolation syntax in our template:

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

@Component({
        selector: 'my-app',
        template: ` {{ message }} `,
})

export class AppComponent {
        message = 'Hello there!';
}
```



A Simple Component

Here's another simple Component that renders our name property:

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

@Component({
        selector: 'my-component',
        template: '<div>Hello my name is {{name}}</div>'
})

export class MyComponent {
        private name;
        constructor() {
                this.name = 'Saad'
        }
}
```

When we use `<my-component>` in our app, this component will be created, its constructor will be called, and then rendered.

Add attribute at runtime: host

By using the `host` option, we're able to configure our host element from within the component.

```
@Component({
        selector: 'app-article',
        templateUrl: './user.component.html',
        styleUrls: ['./user.component.css'],
        host: {
                class: 'row'
        }
})
```

This tells Angular that on the host element (the `app-article` tag) we want to set the class attribute:
`<app-article class="row"></app-article>`

In Angular, a component `host` is the element this component is attached to.

Adding CSS to your component

```
@Component({
        selector: 'app-comp',
        template: ` <p class="red">Red Paragraph</p> `,
        styles: ['p { border: 1px solid black }', 'p.red { color: red; }']
})
```

```
<style>
        p [_ngcontent-yvn-3]{ border: 1px solid black }
        p.red[_ngcontent-yvn-3] { color: red; }
</style>
```

```
@Component({
        selector: 'app-comp',
        template: ` <p class="red">Red Paragraph</p> `,
        styleUrls: ['./user.component.css'],
})
```

```
p { border: 1px solid black }
p.red { color: red; }
```

user.component.css

38

The Shadow DOM

Shadow DOM refers to the ability of the browser to include a subtree of DOM elements into the rendering of a document, but not into the main document DOM tree.

The Shadow DOM is simply saying that **some part of the page, has its own DOM within it**. Styles and scripting can be **scoped** within that element so what runs in it only executes in that boundary.

- The scoped subtree is called a **shadow tree**.
- The element it's attached to is its **shadow host**.

Not all elements can host a shadow tree *v1*
Like <input> <textarea> <img>..etc

DOM Composition

By using Shadow DOM, we have hidden the presentation details of the name tag from the document. The presentation details are encapsulated in the Shadow DOM.

- The content is in the document.
- The presentation is in the Shadow DOM.

PREV NEXT

OUTLINE

Search...

View Encapsulation

The 3 states of view encapsulation in Angular are:

- **None**: All elements/styles are leaked - no Shadow DOM at all.
- **Emulated**: Tries to emulate Shadow DOM to give us the feel that we are scoping our styles. This is not a real Shadow DOM but a strategy that works in all browsers.
- **ShadowDom**: This is the real deal as shadow DOM is completely enabled. Not supported by older browsers.

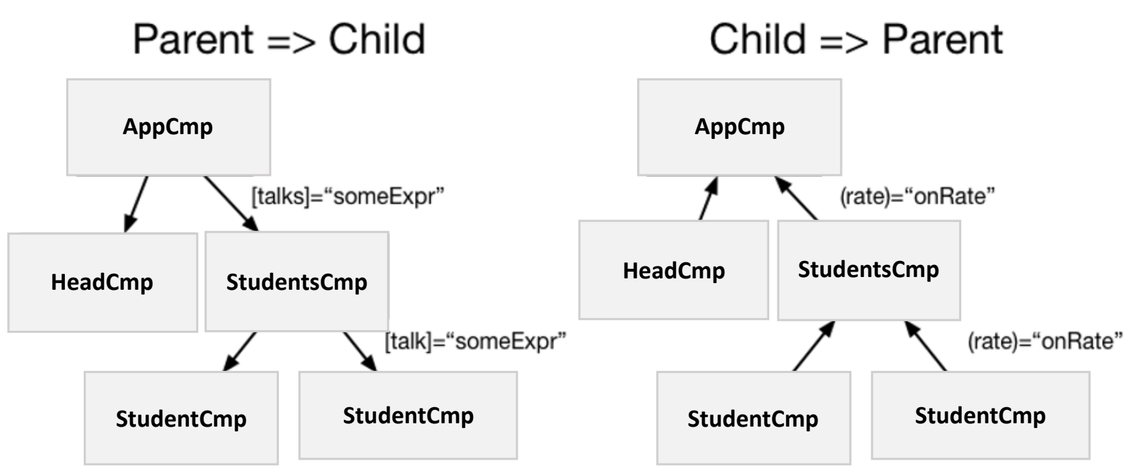
```
@Component({
        templateUrl: 'test.html',
        styles: [` .box { height: 100px; width: 100px; } `],
        // encapsulation: ViewEncapsulation.ShadowDom
        // encapsulation: ViewEncapsulation.None
        // encapsulation: ViewEncapsulation.Emulated is default
})
```


PREV NEXT

Input and Output Properties

A component has **input** and **output** properties, which can be defined in the component decorator or using class property decorators.

Data flows into a component via input properties. Data flows out of a component via output properties.



Inputs and Outputs

Input and output properties are the public API of a component. You use them when you instantiate a component in your application.

```
<myComponent
    [color]="colorValue" <!-- input -->
    (onComponentSelected)="componentWasSelected($event)"> <!-- output -->
</myComponent>
```

$event is a special variable here that represents the thing being emitted.

[squareBrackets] pass inputs: You can set input properties using property bindings
(parens) handle outputs: You can subscribe to output properties using event bindings

Native Components

Every native component, by default, has inputs and outputs:

For example: <input />

- Native input properties: **value**, **class**, **id**, **type**.. etc
- Native output properties: **click**, **mouseover**, **input**, **keyup**.. etc

All attributes are input

All events are output

We can simply interact with it:

```
<input [value]="prop" (keyup)="doSomething() "/>
```

Component `inputs`

With the **inputs** property, we specify the parameters we expect our component to receive. Inputs takes an array of strings which specify the input keys.

When we specify that a Component takes an input, it is expected that the class will have an instance variable that will receive the value.

```
@Component({
    selector: 'my-component',
    inputs: ['name', 'age']
})
export class MyComponent {
    name: string;
    age: number;
}
```

⟺

```
@Component({
    selector: 'my-component'
})
export class MyComponent {
    @Input() name: string;
    @Input() age: number;
}
```

```
<my-component [name]="myName" [age]="myAge"></my-component>
```

48

Passing data to `inputs`

```
@Component({
        template: `
                <my-course [grade]="stugrade" ></my-course>
                <my-course grade="{{stugrade}}" ></my-course>
                <my-course grade="100" ></my-course>
        `,
})
export class ParentComponent {
        stugrade: 100;
}
```

<ng-content>

```
import { Component, Input } from '@angular/core';
@Component({
        selector: 'comp',
        template: ` <ul>
                        <li>Message 1: {{message1}}</li>
                        <li>Message 2: {{message2}}</li>
                    </ul>
                    <ng-content></ng-content> `
})
export class Comp1Component {
        @Input() message1;
        @Input() message2;
}
```

Property Binding: When we add an attribute in brackets like `[foo]` we're saying we want to pass an **expression value** to the input named `foo` on that component.

```
<comp message1="{{msg}}" [message2]="msg">Hey</comp>
```

Assume having `msg="Hi";` in the parent component

Component outputs

When we want to send data from your component we use **output bindings**.

```
@Component({
   selector: 'my-component',
   outputs: ['onLectureEnds']
})
export class MyComponent {
  onLectureEnds = new EventEmitter();
}
```

```
@Component({
   selector: 'my-component'
})
export class MyComponent {
   @Output() onLectureEnds: new EventEmitter();
}
```

```
<my-component (onLectureEnds)="doSomething()"></my-component>
```

Native Element Output Example

```
@Component({
        selector: 'counter',
        template: ` {{ value }}
                    <button (click)="increase()">Increase</button> `
})
export class Counter {
        value: number;
        constructor() {
                this.value = 1;
        }

        increase() {
                this.value = this.value + 1;
                return false;
        }
}
```

tells the browser not to propagate the event upwards

Native Element Output Example

```
@Component({
        selector: 'counter',
        template: ` {{ value }}
                    <button (click)="increase($event)">Increase</button> `
})
export class Counter {
        value: string;
        constructor() {
                this.value = ‘no click’;
        }

        increase(e) {
                this.value = e.target.innerHTML;
                return false;
        }
}
```

What would $event refer to in this example?

Emitting Custom Events

Let’s say we want to create a component that emits a custom event, just like native components events "**`click`**" or "**`mousedown`**".

To create a custom output event we do three things:

1. Specify `outputs` property
2. Attach an **`EventEmitter`** to the output property
3. Emit an event from the **`EventEmitter`** , at the right time

An `EventEmitter` is simply an object that helps you implement the Observer Pattern. It’s an object that can maintain a list of subscribers and publish events to them.

PREV NEXT

Custom Event Example

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

@Component({
        selector: 'MWA-Lecture',
        outputs: ['onLunchBreak'],
        template: `<button (click)="start()">Start Lunch Break</button>`
})
class OutputComponent {
        onLunchBreak: EventEmitter<string>;
        constructor() {
                this.onLunchBreak = new EventEmitter();
        }

        start(): void {
                this.onLunchBreak.emit("Yes finally!! I'm hungry!");
        }
}
```

1. specified outputs

2. created an `EventEmitter` that we attached to the output property `onLunchBreak`

3. Emit an event when `start()` is called

Custom Event Example - Continued

If we wanted to use this output in a parent component we could do something like this:

```
@Component({
        selector: 'university',
        template: `<div>
                        <MWA-Lecture (onLunchBreak)="eatFood($event)"></MWA-Lecture>
                    </div> ` })

class UniversityComponent {
        eatFood(e: string) {
                console.log(`Message: ${e}`); // Yes finally!! I'm hungry!
        }
}
```

`onLunchBreak` comes from the outputs of `OutputComponent`

`$event` contains the thing that was emitted, in this case a string

Two-way Data-binding

```
@Component({
        selector: 'comp',
        template: `
            <p>Message: {{message}}</p>
            <input [value]="message" (input)="message=$event.target.value">
            `
})
export class CompComponent {
        public message: string = 'Default Message';
}
```

Main Points

Angular is primarily a **one-way** data-binding process. One-way makes our applications very predictable and we can always know that a child component will get updated after its parent components.

- () event binding: data flows out
- [] property binding: data flows into

When learning Services, We'll use these bindings to pass data between components.

In addition to **inter-component** communication, these bindings can also work within a **singular component** and pass information between class to template.

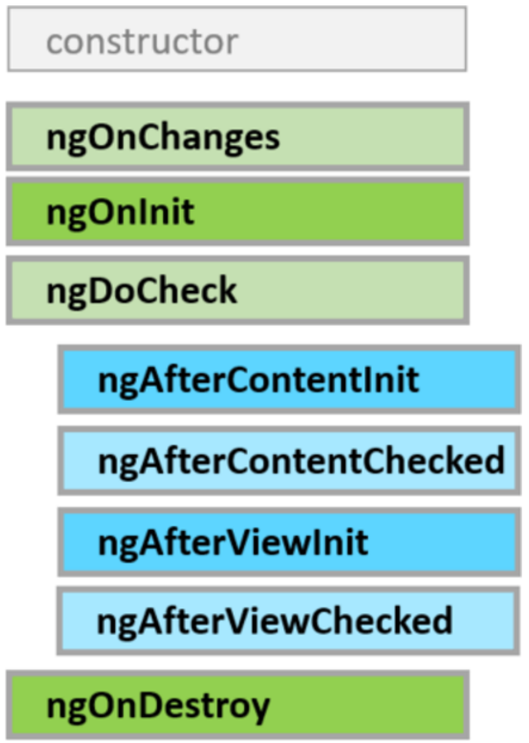
PREV NEXT

Component Lifecycle Hooks

Angular components go through a multi-stage bootstrap and lifecycle process, and we can respond to various events as our app starts, runs, and creates/destroys components.

After Angular creates a component/directive by calling `new` on its constructor, it calls the lifecycle hook methods in the following sequence at specific moments.

Angular only calls a directive/component hook method if it is defined.



62

Lifecycle Example

This component will be notified when its input properties change.

```
@Component({
        selector: 'app-cmp'
})
class AppCmp implements OnChanges {
        @Input() field1;
        @Input() field2;

        ngOnChanges(changes) {
                //..
        }
}
```

Content and View children

A component can interact with its children. There are two types of children a component can have:

- Content child/children
- View child/children.

Template Local Variables

```
import { Component } from '@angular/core';
@Component({
        selector: 'my-component',
        template: `<div>
                      <input name="title" value="Asaad" #myName />
                      <button (click)="sayMyName(myName)">Say my name</button>
                   </div>`
})
export class MyComponent {
   @ViewChild('myName', {static: true}) myFullName;
         sayMyName(name) {
            console.log(`My name is ${name.value}`)
            // OR
            console.log(`My name is ${myFullName.nativeElement.value}`)
         }
}
```

myName is now an object that represents this input DOM element (HTMLInputElement).

When {static: true} then query results available in ngOnInit, otherwise query results available in ngAfterViewInit

66

Template Variables Binding

```
<comp> <p #myParentParagraph>Parent Paragraph</p> </comp>
```

```
import { Component, ContentChild } from '@angular/core';
@Component({
        selector: 'comp',
        template: `
                <ng-content></ng-content>
                <button (click)="getData()">Get Parent Paragraph Data</button> `
})
export class CompComponent {
    @ContentChild('myParentParagraph', {static: true}) myParentPObj;
    getData(){
        console.log(this.myParentPObj.nativeElement.textContent);
    }
}
```

When {static: true} then query results available in ngOnInit, otherwise query results available in ngAfterContentInit

Main Points

Components in Angular are self-describing, they contain all the information needed to instantiate them. This means that any component can be bootstrapped. It does not have to be special in any way.

- A component knows how to interact with its host element.
- A component knows how to interact with its content and view children.
- A component knows how to render itself.
- A component configures dependency injection.
- A component has a well-defined public API of input and output properties.

PREV NEXT

Debugging Angular Apps

Augury is the most used Google Chrome Developer Tool extension for debugging and profiling Angular applications.

Augury helps Angular developers visualize the application through component trees, and visual debugging tools. Developers get immediate insight into their application structure, change detection and performance characteristics.

https://augury.angular.io/

PREV NEXT