

Building Blocks

SD 555 – Web Application Development III

Maharishi International University

Department of Computer Science

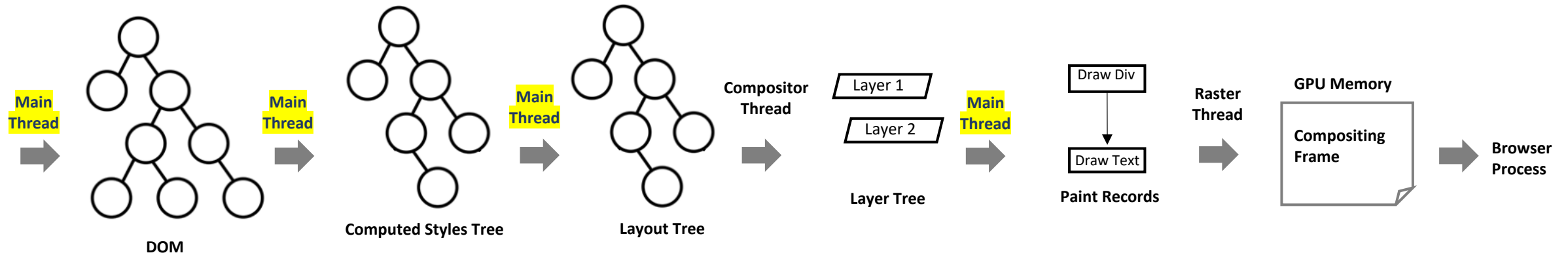
Associate Professor Asaad Saad

Maharishi International University - Fairfield, Iowa



All rights reserved. No part of this slide presentation may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying or recording, or by any information storage and retrieval system, without permission in writing from Maharishi International University.

Renderer Process



For every tab in your browser, a Renderer process is created. The Renderer process core job is to turn HTML, CSS, and **JavaScript** into a web page that the user can interact with.

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.

Angular~~JS~~ (Do Less Accomplish More)

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

- Support policy with security

- Browser compatibility fixes



Angular CLI

Make sure you are using a compatible Node version

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

```
npm install @angular/cli -g
ng help
ng new new-app -s -t -S --routing false // inline template and style,
                                         without test files, without router
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)

Schematics Configurations

```
"schematics": {  
  "@schematics/angular:component": {  
    "style": "css"  
    "skipTests": true,  
    "inlineTemplate": true,  
    "inlineStyle": true,  
    "flat": true,  
    "displayBlock": true, // :host { display: block; }  
    "changeDetection": "Default", // "OnPush"  
    "viewEncapsulation": "Emulated"  
  }  
}
```

Project Structure

The directory structure for your app will look like this:

```
| - src/  
    | - index.html  
    | - main.ts // bootstrap our app  
    | - app/  
        | - app.component.ts // main app root component  
        | - app.config.ts    // main app configuration  
| - package.json  
| - tsconfig.json  
| - angular.json
```


Angular Dependencies

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

zone.js: Creates execution context around my app. Helps with change detection and showing errors. Provides stack traces.

rxjs: help create asynchronous data streams. Gives us Observables, the preferred way of handling events in Angular.

Bootstrap

`main.ts`: This is where we bootstrap our app.

- `app.component.ts`: The root component.

- `app.config.ts`: The top-level configurations for our 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.

Angular Building Blocks

Most Angular building blocks like **components**, **pipes**, **directives**, **guards**, and many more, have been historically authored with OOP.

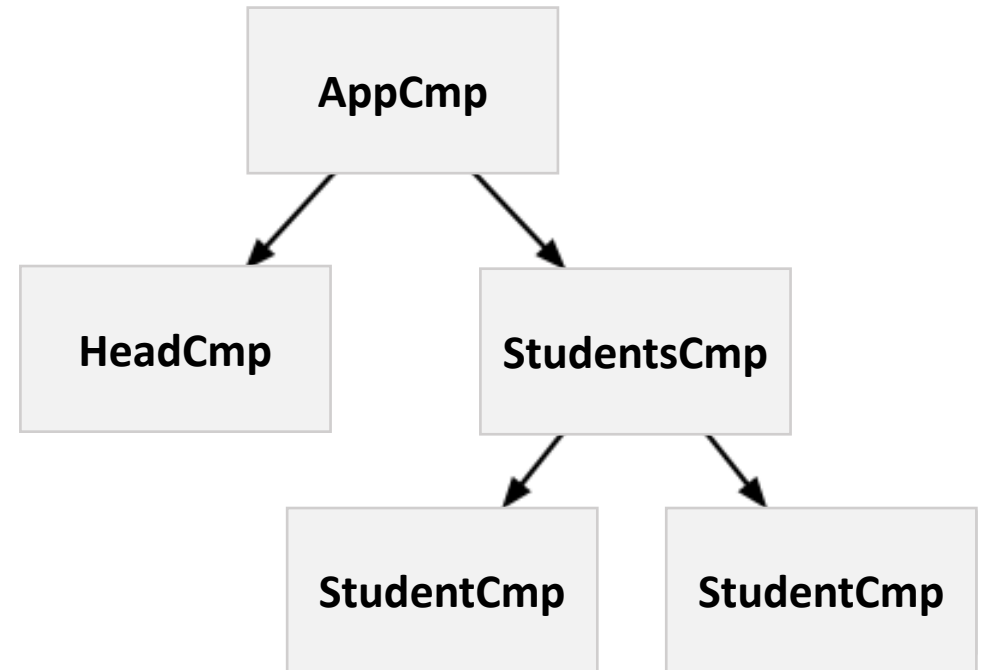
All of those building blocks are represented by a class, data inside of them is stored as a property, and their behavior is described via methods.

Classes are marked with one of Angular's decorators: @Component, @Pipe, @Directive, and @Injectable.

*Note: Some of those building blocks do not have to be classes, which allow DI using **inject()** method, removing the constraint of the @Injectable() decorator.*

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.



What are Components?

Components are custom HTML tags (UI elements), and each component has:

- **state** (data)
- **template/view** (data representation in the DOM)

What is an Angular Component?

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

```
@Component({  
  selector: '',  
  template: ``, // or templateUrl  
  standalone: true,  
  imports: [],  
  styles: `` // or styleUrls - all styles are encapsulated  
})  
class AppComponent {}
```

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
```

```
--flat // no new folder  
-t // inline template  
-s // inline styles  
-S // skip creating test file
```

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!';  
}
```

DOM

One-Way Data Binding (State to DOM)

State

Control Flow - Conditionals

```
@if(a > b){
```

```
    {{a}} is greater than {{b}}
```

```
}@else if(b > a){
```

```
    {{a}} is less than {{b}}
```

```
}@else{
```

```
    {{a}} is equal to {{b}}
```

```
}
```

Control Flow - Repeaters

```
@for(user of users; track user.id; let idx = $index, e = $even){
```

```
  Item #{{ idx }}: {{ user.name }}
```

```
}@empty{
```

```
  Empty list of users
```

```
}
```

Inside `@for` contents, several implicit variables are available, they can be aliased via a `let` segment:

<code>\$count</code>	Number of items in a collection iterated over
<code>\$index</code>	Index of the current row
<code>\$first</code>	Whether the current row is the first row
<code>\$last</code>	Whether the current row is the last row
<code>\$even</code>	Whether the current row index is even
<code>\$odd</code>	Whether the current row index is odd

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({  
  template: '<p class="box"></p>',  
  styles: [`.box { height: 100px; width: 100px; } `],  
  // encapsulation: ViewEncapsulation.ShadowDom  
  // encapsulation: ViewEncapsulation.None  
  // encapsulation: ViewEncapsulation.Emulated is default  
})
```

Emulated Encapsulation


```
@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

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>
..etc*

ngStyle Directive

To set a given DOM element CSS properties

```
imports: [NgStyle],
template: `
  <div [ngStyle]="{color: 'white', 'background-color': 'blue'}">
    White text on blue background
  </div>
`
```

ngClass Directive

Given the following defined CSS selectors:

```
imports: [NgClass],
template: `
  <div [ngClass]="{special: false, big: true}">text</div>
`,
styles: `
  .special { background-color: yellow }
  .big { font-size: 36px }
`
```

We can also use a list of class names to specify multiple classes should be added to the element.

```
<div class="base" [ngClass]="['special', 'big']">Text</div>
```

Pipes

Pipes transform displayed values within a template.

Example: *When data arrives in our component, we could just push their raw values directly to the view. That might make a bad user experience. Everyone prefers a simple birthday date like April 15, 1988 to the original raw string format — Fri Apr 15 1988 00:00:00 GMT-0700 (Pacific Daylight Time).*

```
{{ dateObj | date }} // output is 'Apr 15, 1988'
```


Built-in Pipes

async

currency

date

decimal

json

lowercase

percent

slice

uppercase

keyvalue

Examples:

```
imports: [CurrencyPipe, DatePipe, UpperCasePipe],
```

```
template:
```

```
<p>{{ myNum | currency }}</p>
```

```
<p>{{ myValue | uppercase }}</p>
```

```
<p>{{ myDate | date:"MM/dd/yy" }}</p>
```

```
<p>{{ myValue | slice:3:7 | uppercase }}</p>
```

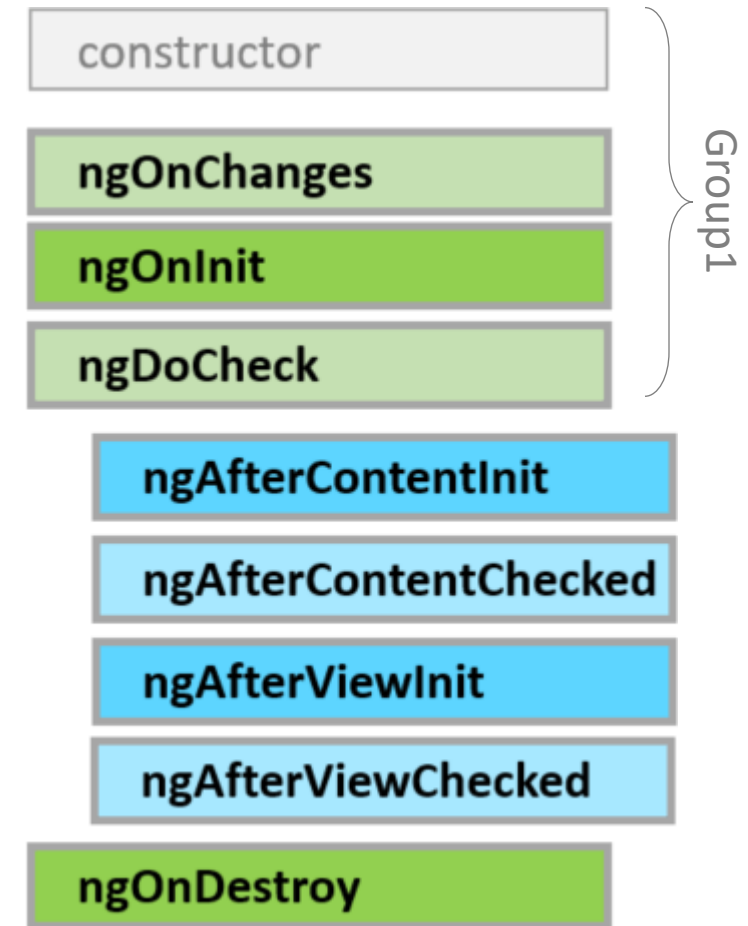
Pure and Impure Pipes

When Pipe is pure, it means that Angular will **NOT re-run** them on the value they are applied to upon each change detection cycle. This behavior makes sense, as it saves performance.

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.

Lifecycle hooks are called in **groups**, the first group runs for all children components, and then the second group.



*View = Template

*Content = elements between
opening and closing tags

Mounting Phase

These hooks are triggered **once**:

ngOnInit() Here we can access any **input()** property.

ngAfterContentInit() Here we can access any content **#ref**

ngAfterViewInit() Here we can access any Template/View **#ref**

Change Detection Phase

These hooks are triggered **every time** change detection cycle is triggered and the component need to be checked (eventually re-rendered):

ngDoCheck() Starting Change Detection

ngAfterContentChecked()

ngAfterViewChecked() Change Detection is finished