



Angular Application Development



Dan Wahlin

About the Instructor



Blog

<http://blog.codewithdan.com>



Twitter

@DanWahlin

Dan Wahlin founded Wahlin Consulting which provides consulting and training services on Web technologies such as JavaScript, Angular, Node.js, Express, C#, ASP.NET MVC, Web API and Docker. He's also published many developer courses on Pluralsight.com and Udemy.com, multiple books on Web technologies and published hundreds of technical articles and blog posts. Dan is a member of the Google GDE, Microsoft MVP, Microsoft Regional Director and Docker Captain groups and speaks at conferences and user groups/meetups around the world. Stay up on the latest technologies with his *Code with Dan - Web Weekly Newsletter*.

Getting Started

- Class Length
- Class Time
- Breaks
- Lunch
- Restrooms
- Cellphones (please mute)
- Personal Introductions

Introductions

Please introduce yourself:

- Name
- Web Development Experience?
- Hobbies



Pre-Reqs

- Web developers with prior experience using the following:
 - HTML
 - CSS
 - JavaScript
- **IMPORTANT:** To get the most out of this class you need to have **existing hands-on experience** with JavaScript, HTML and CSS. If you don't meet these pre-reqs please talk with the instructor.

Course Expectations

- Learn how TypeScript can be used
- Learn Angular core concepts and best practices
- Hands-on course - expect to work with a lot of Angular and TypeScript code
- Learning is not effective unless you're relaxed and having fun. Have fun and enjoy it! ☺

Table of Contents

- Angular Overview
- TypeScript JumpStart
- Angular JumpStart Application
- Components and Modules
- Template Expressions and Pipes
- Component Properties and Data Binding

Table of Contents

- Services, Providers and HttpClient
- Routing
- Route Guards and Lazy Loading
- Template and Reactive Forms
- Bonus Content

Samples and Lab Code

<https://github.com/DanWahlin/AngularAppDevCourseCode>



Angular Application Development



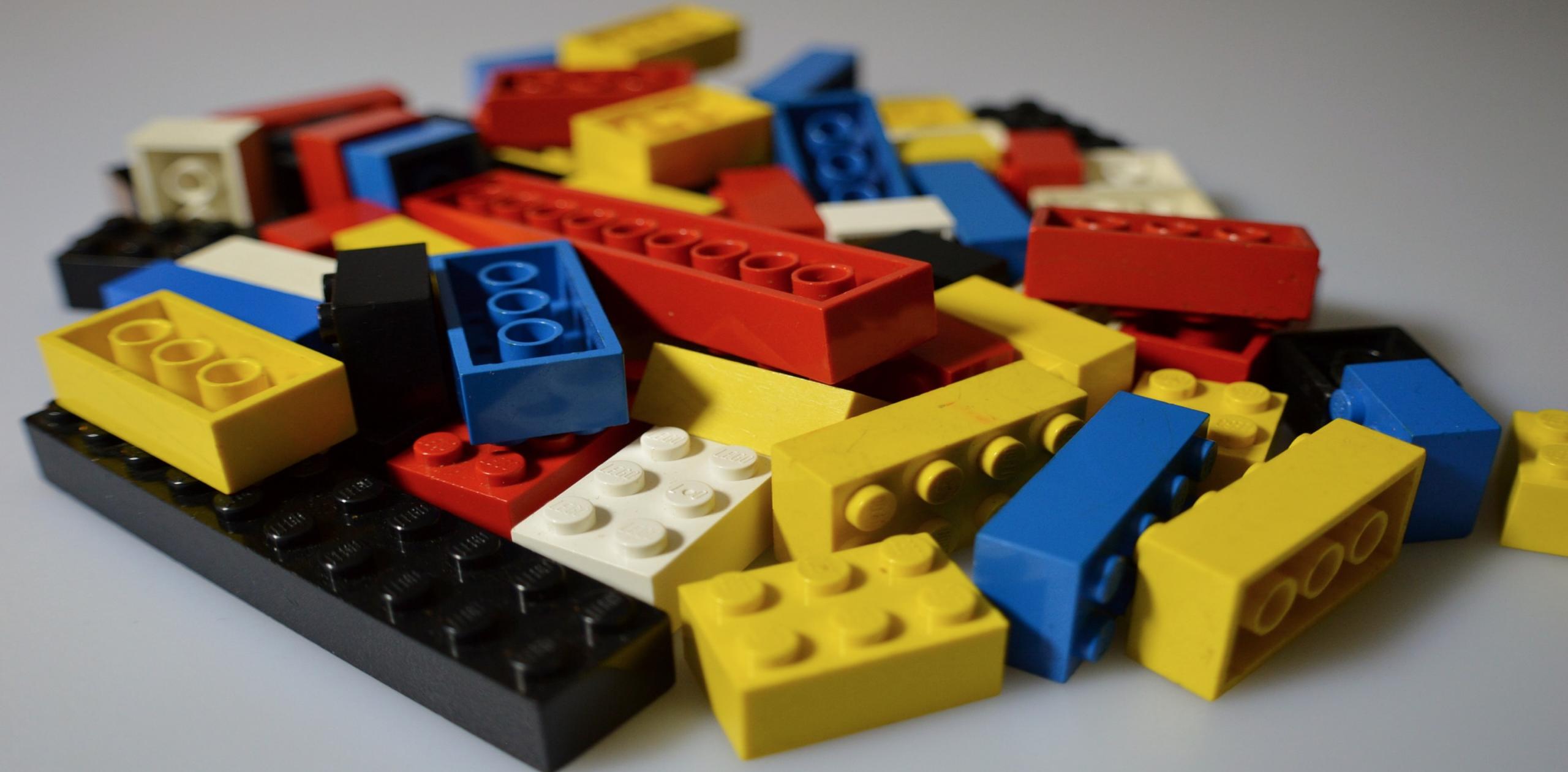
Introduction to Angular

Agenda

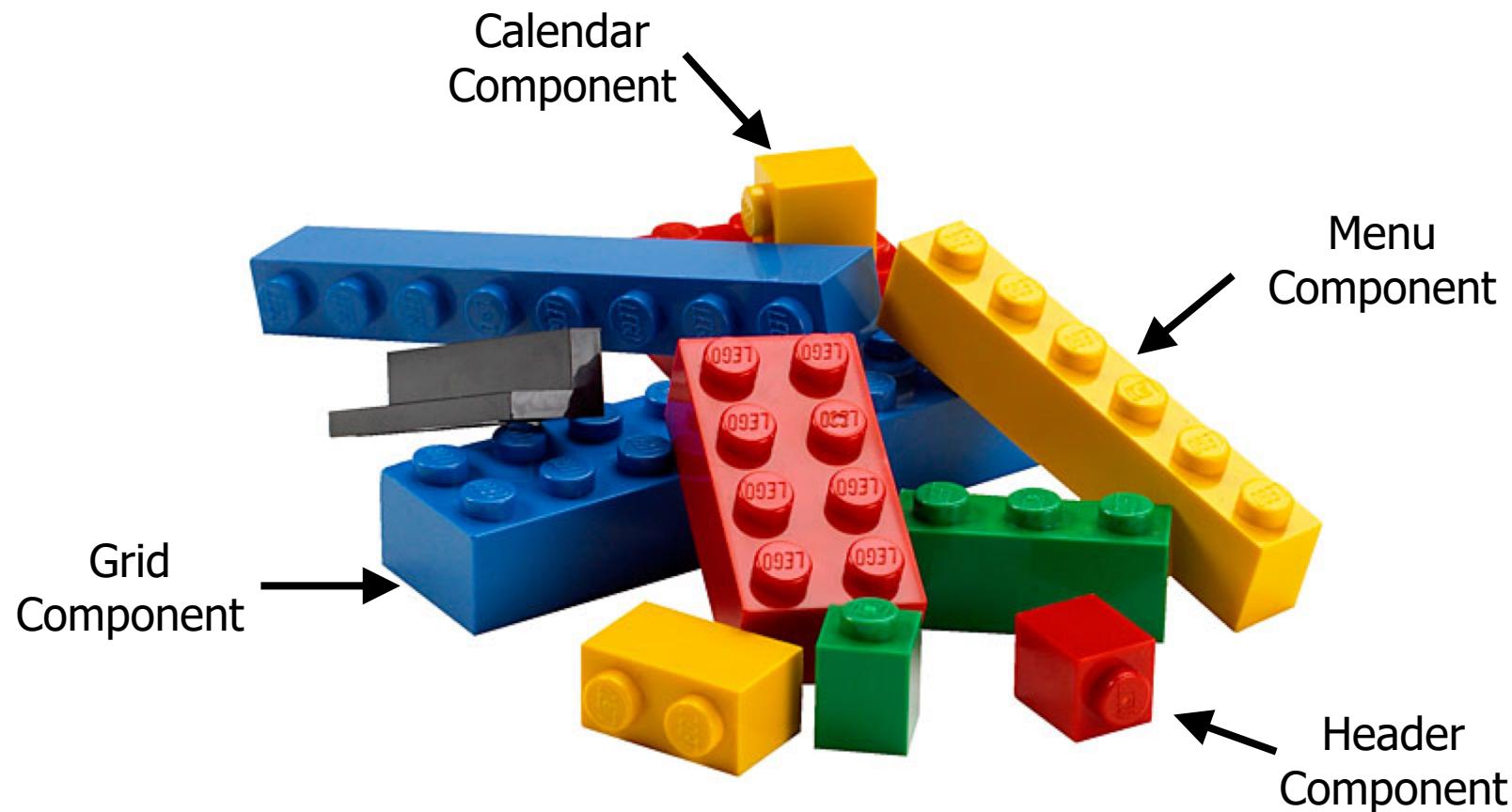
- Application Building Blocks
- Angular Feature Overview
- The Big Picture
- Angular CLI
- Angular Docs



Application Building Blocks

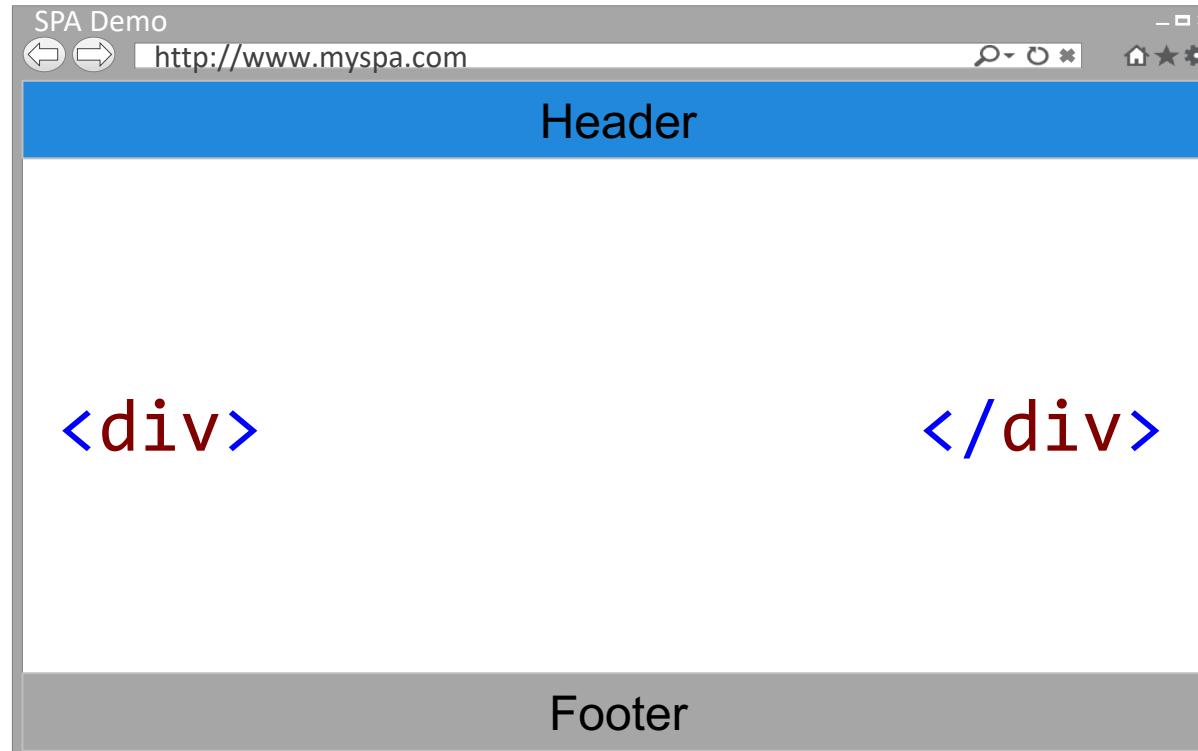


Component Building Blocks



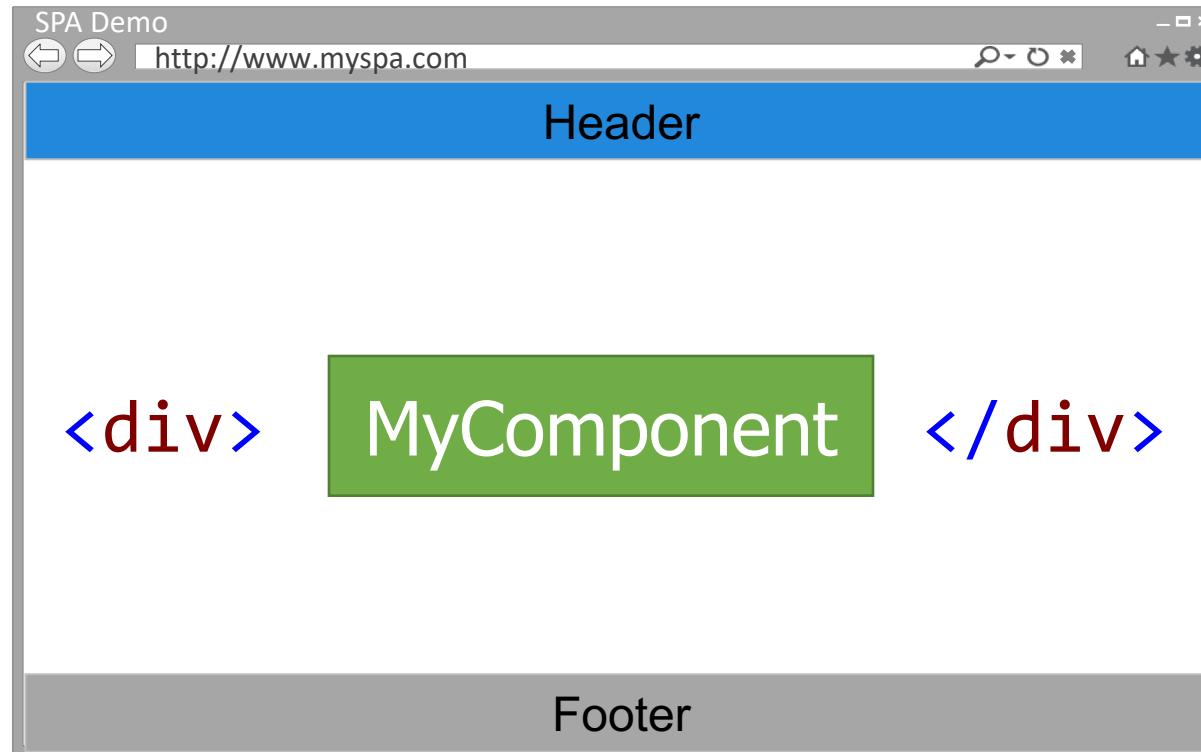
Components and Dynamic Web Applications

Dynamic Web Applications are composed of one or more components (SPAs, mobile, etc.)



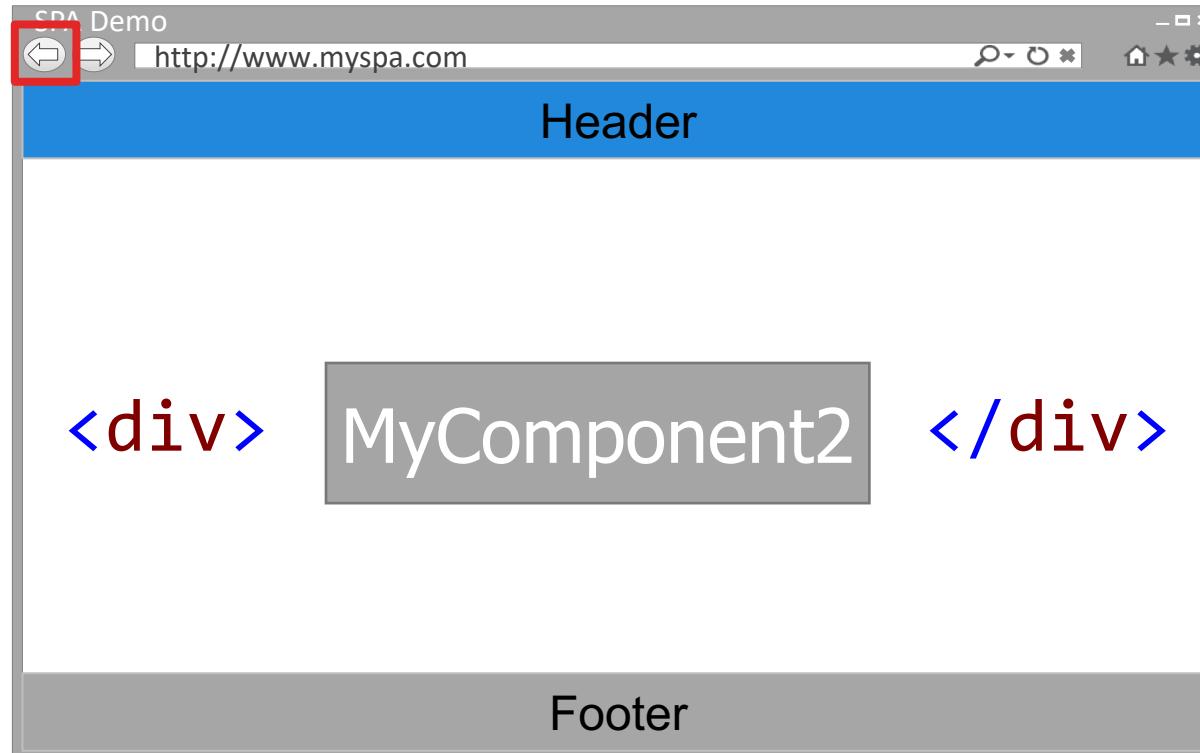
Loading Components

Components can be dynamically replaced with other components using routing

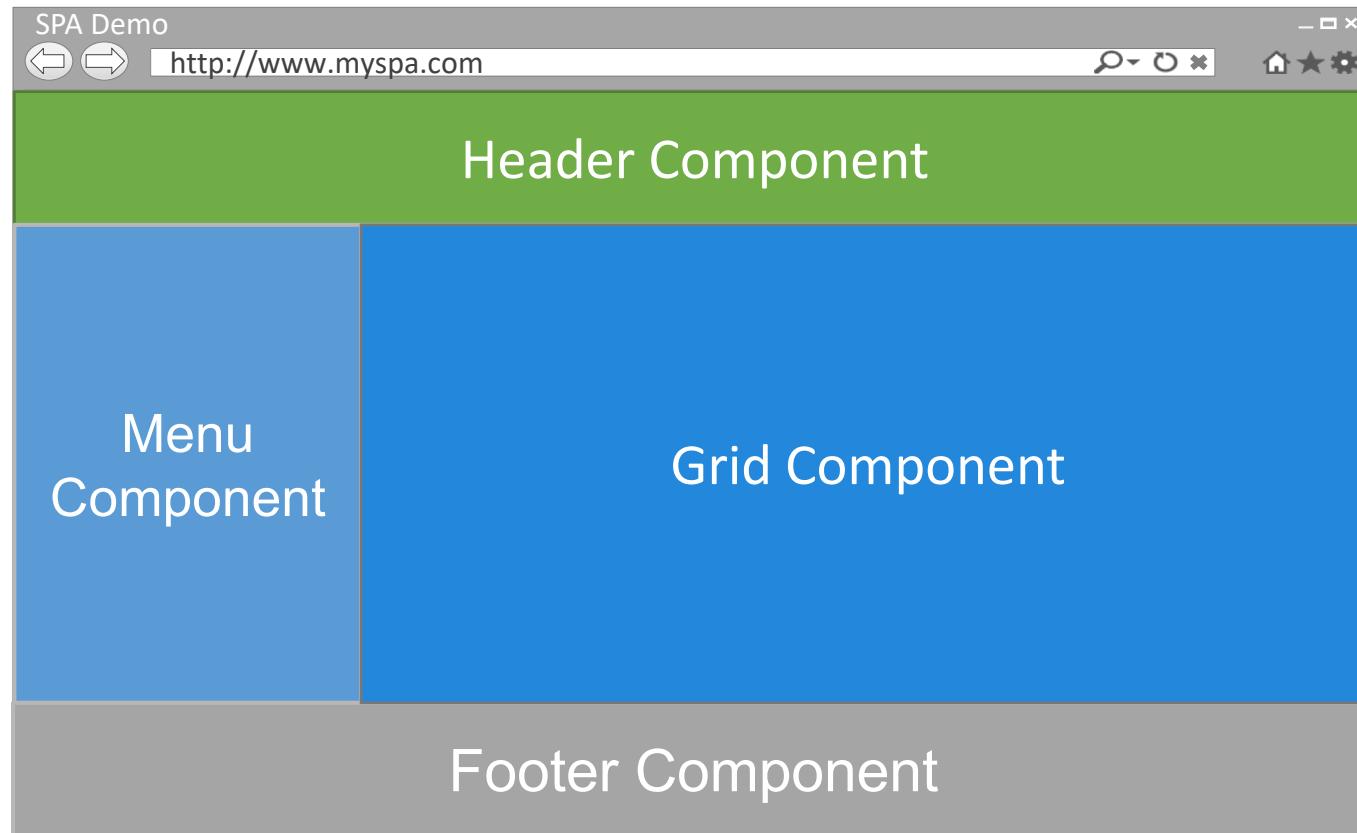


Components, SPAs and History

A history of components that have been displayed is kept allowing navigation forward and backward



Building SPAs with Components



Angular Feature Overview

AngularJS

Version 1.x of the framework

Angular

Version 2 or higher of the framework

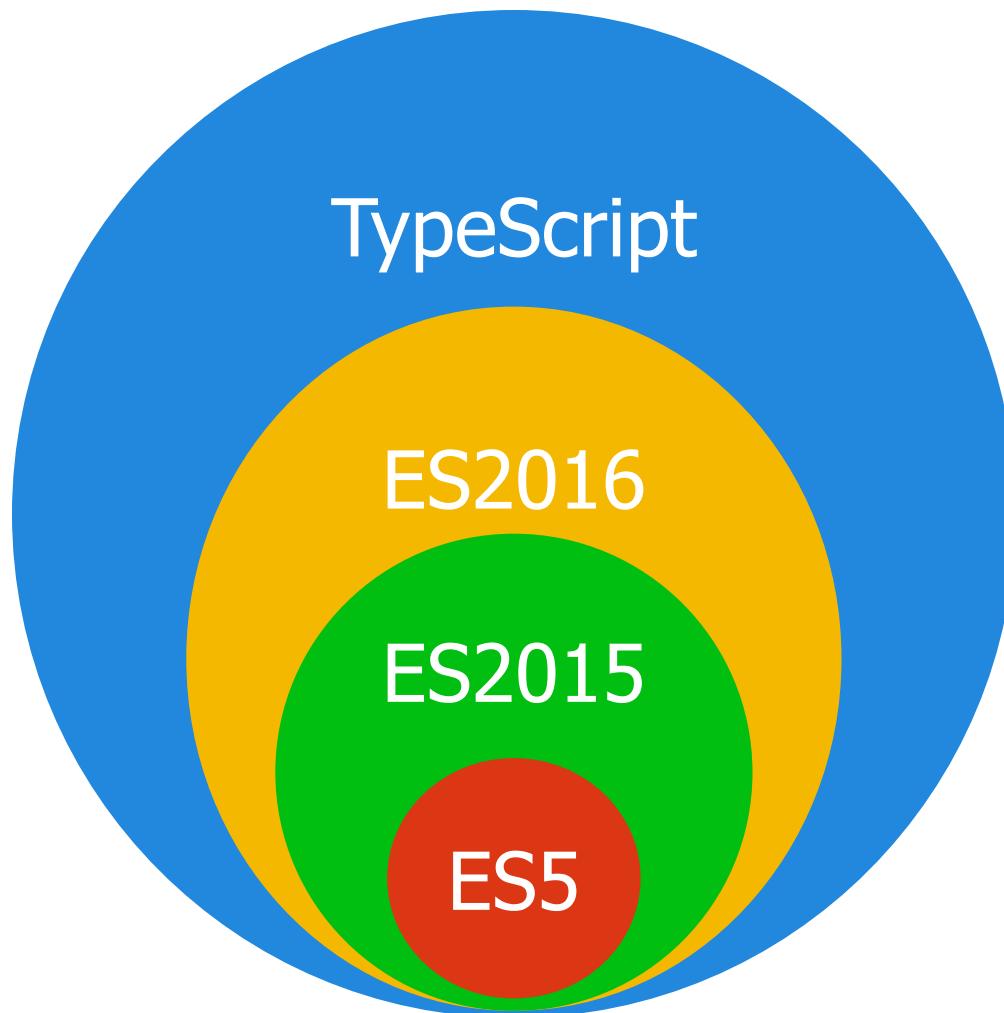
Key Angular Features

- Leverage advances in browsers
- Use latest language features (ES2015/TS)
- Ultrafast change detection
- Modular framework
- Ahead of Time Compilation (AOT)
- Fast initial load from the server with Angular Universal
- Mobile application support - Ionic, NativeScript
- Desktop apps – Electron



Angular.js

Languages



Key Angular Features

Components

Templates

Data Binding

Languages
(TypeScript,
ES2015, ES5)

Services

Dependency
Injection

Decorators

Modules

Templates

Components

Data Binding

Routing

History

Services

Decorators

Additional Features...

Directives

Testing

Injection

Animations

Forms

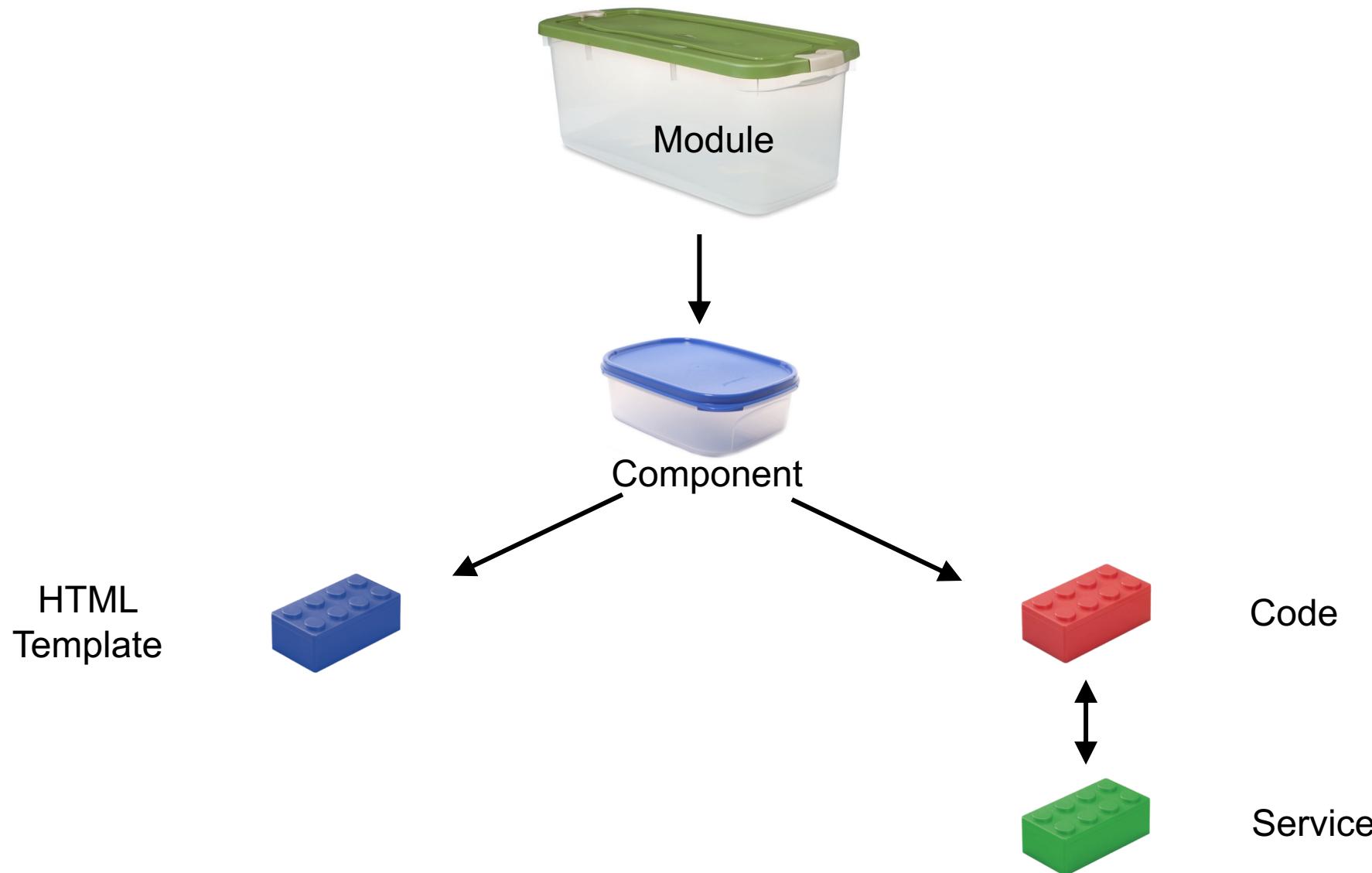
Mobile

Validation

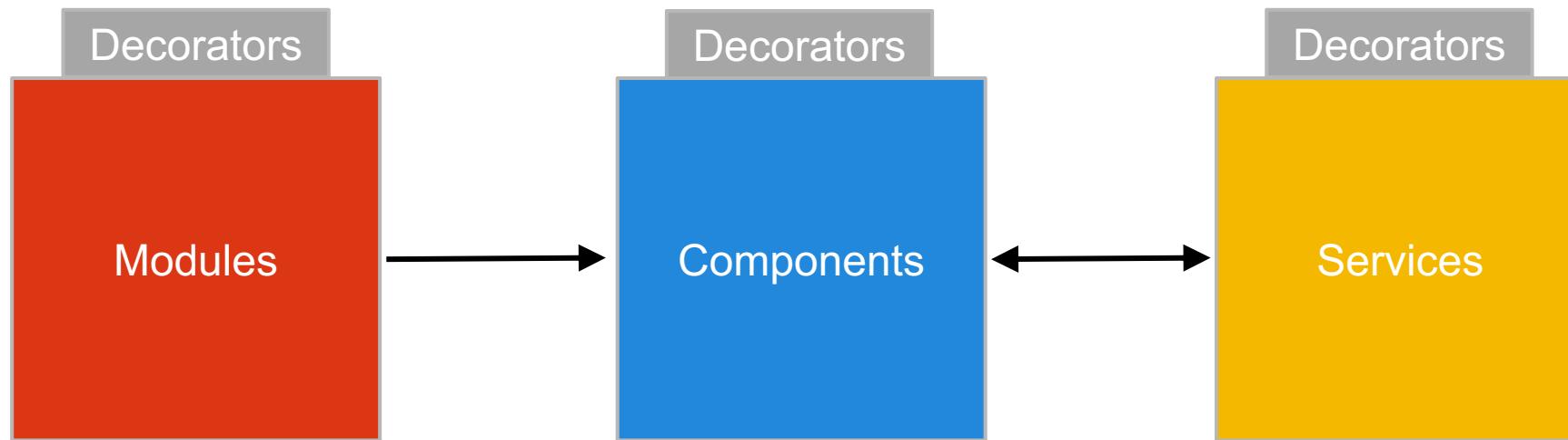
Modules

The Big Picture

How it All Fits Together

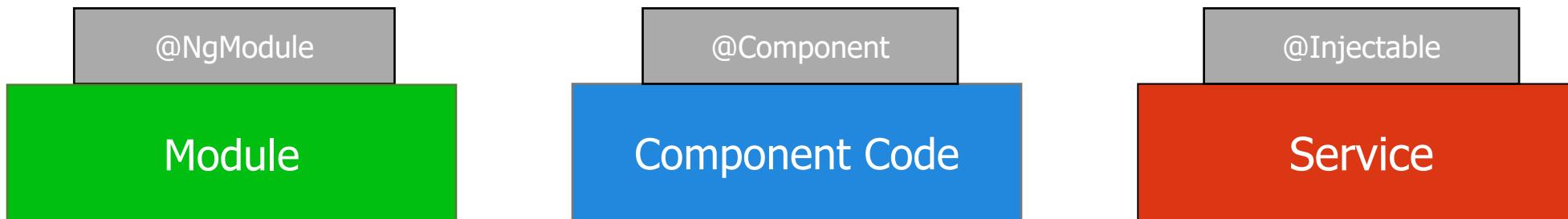


Angular Building Blocks



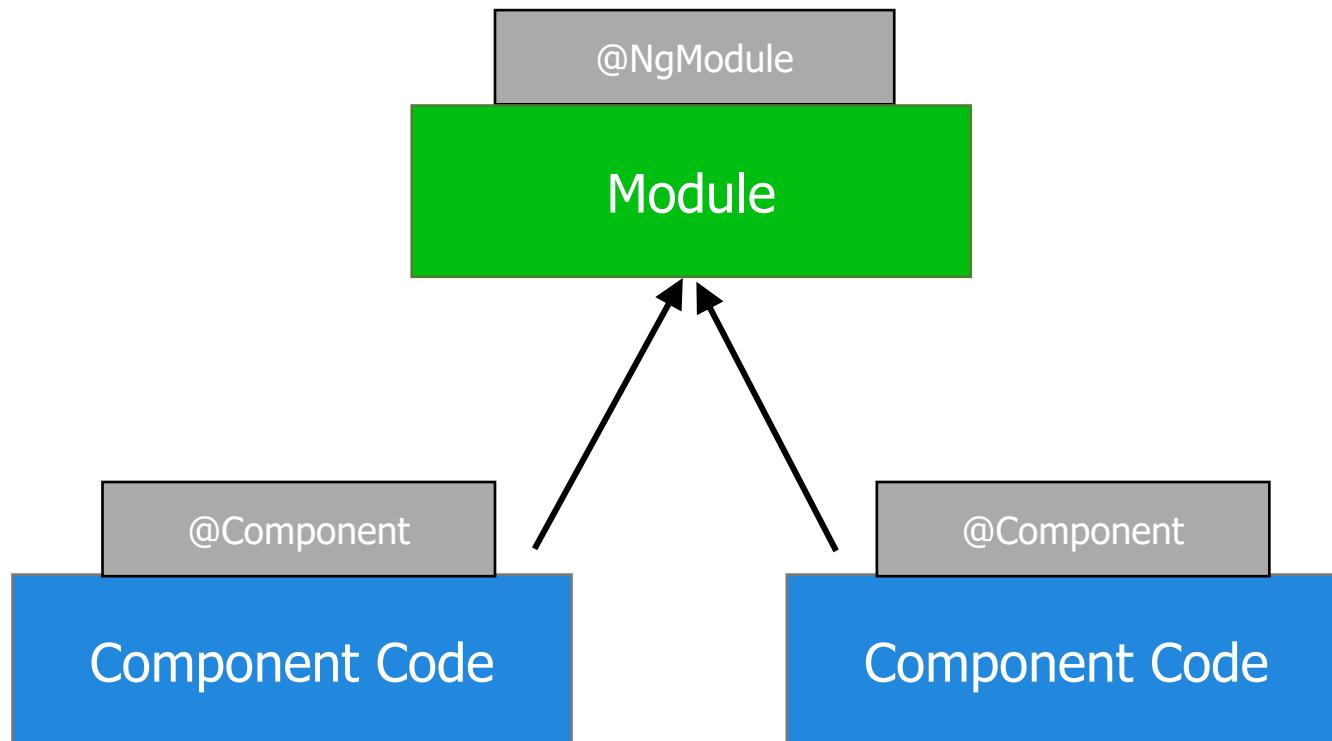
What's a Decorator?

Decorators are functions that attach metadata to classes.



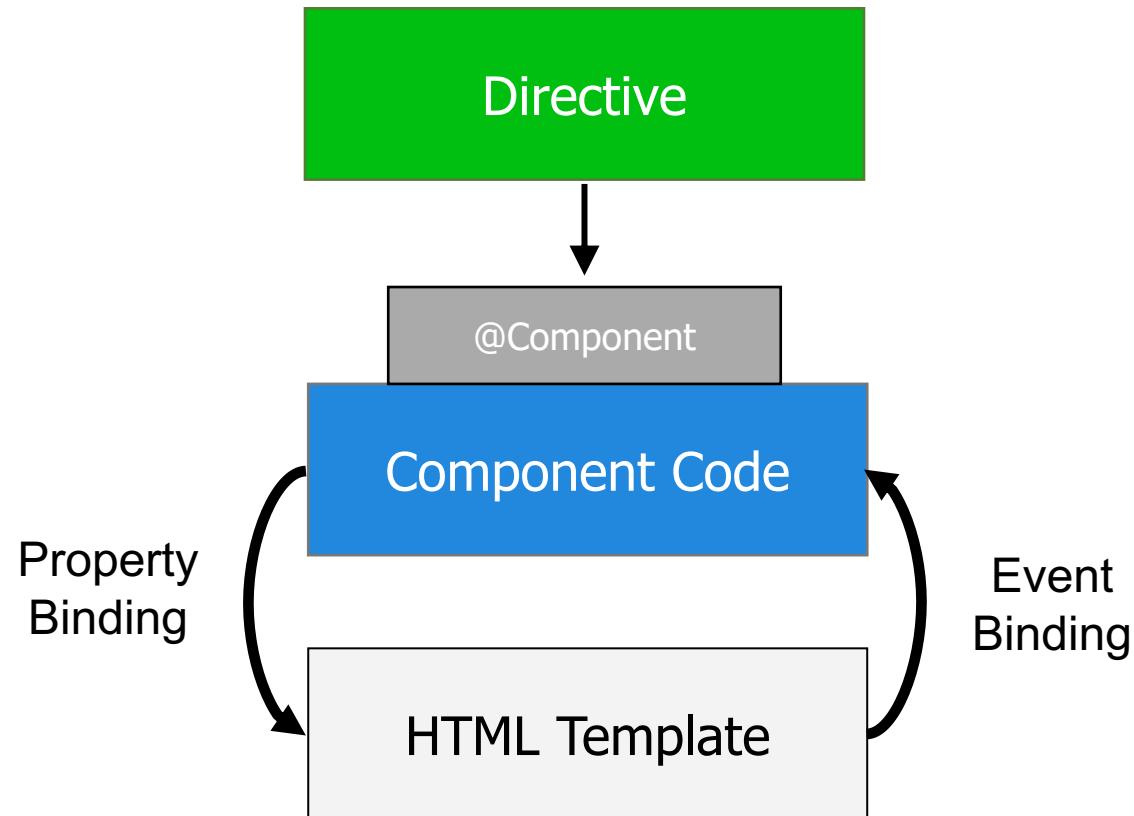
What's a Module?

Modules are used to "group" related functionality.



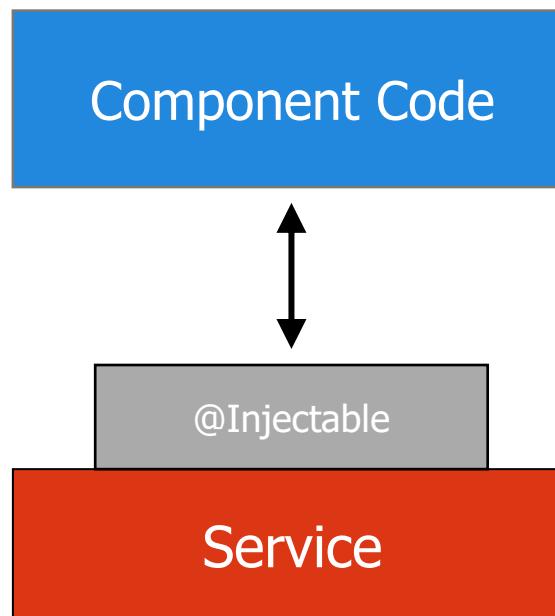
What's a Component?

Components are the building blocks of Angular. They're composed of code and an HTML template.

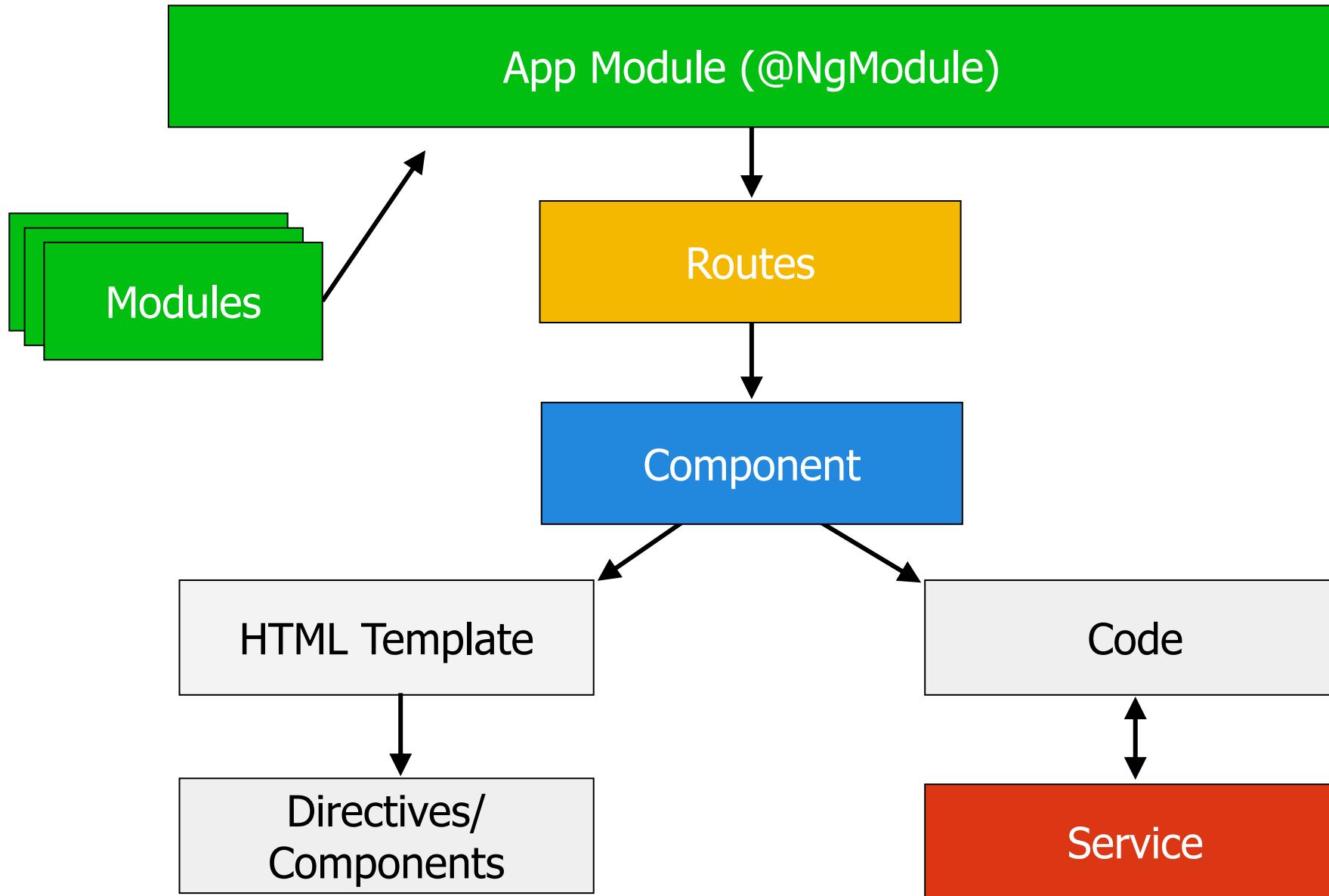


What's a Service?

A service is a reusable block of code.



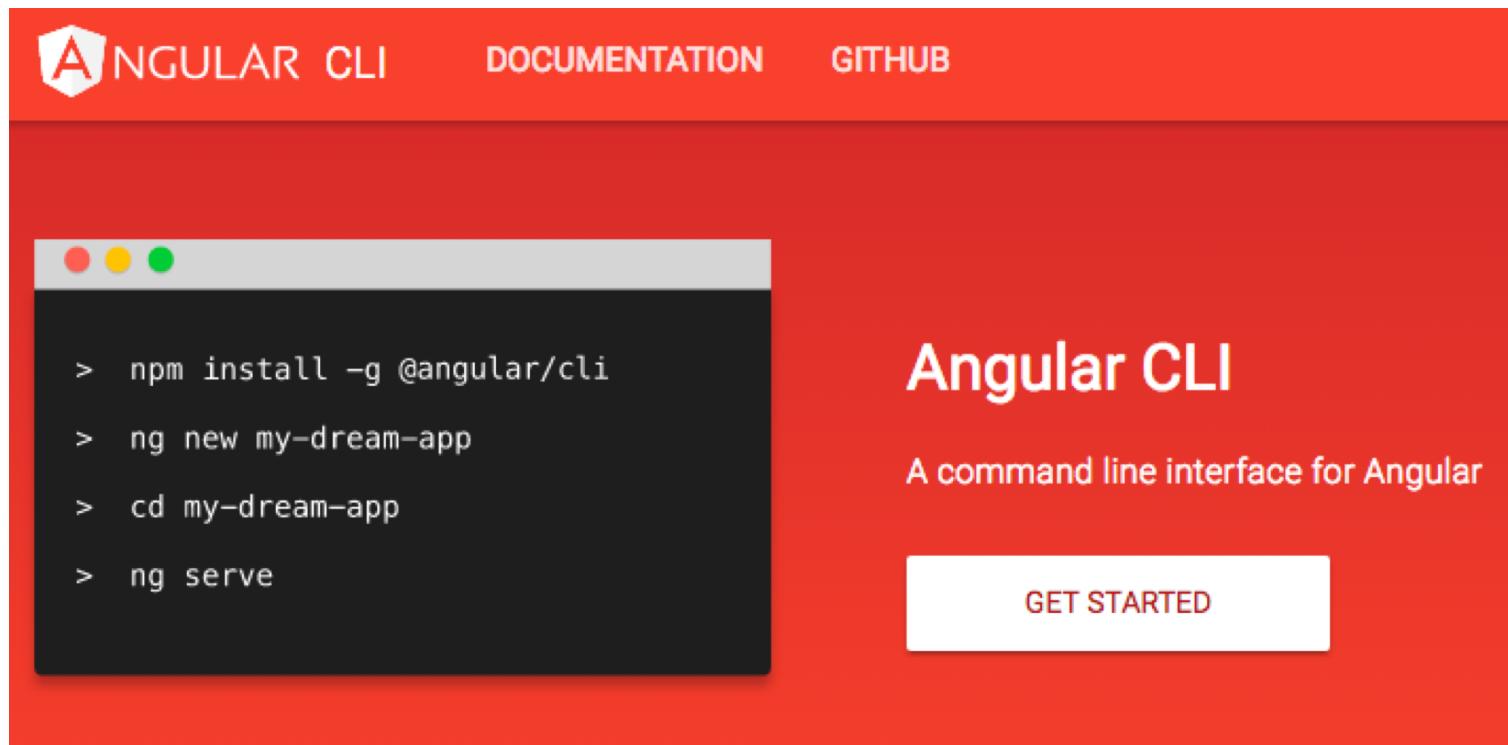
The Big Picture



Angular CLI

The Angular CLI

- Angular applications can be generated using the Angular Command-Line Interface (CLI): <https://cli.angular.io>



Angular CLI Key Features

- Easily create an Angular application that follows best practices in the Angular style guide:

<https://angular.io/docs/ts/latest/guide/style-guide.html>

- Create new components, directives, pipes, routes and services
- Create a "build" version of the application for deployment
- Run unit tests and end-to-end tests
- Serve up the application in the browser



Key Angular CLI Commands

ng --help

ng new [my-app-name]

ng [g | generate] [component | directive | route | pipe | service]

ng build

ng serve

ng lint

ng test

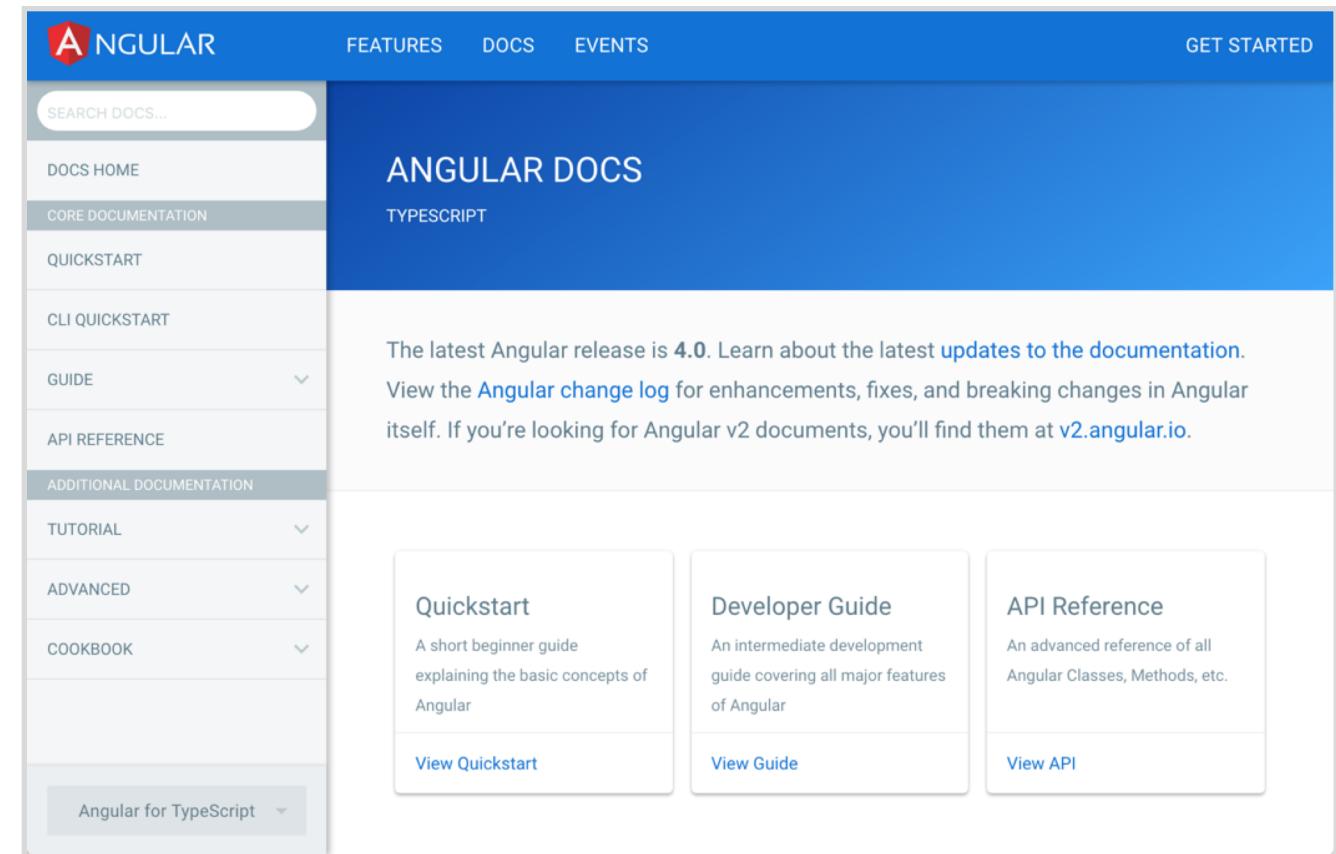
Creating an Application with the Angular CLI

```
ng new my-app
```

Angular Docs

Angular Documentation

- Angular documentation can be found at <https://angular.io/docs>
- Quickstart, tutorials, cookbook recipes, articles and an API reference



Summary

- Modules, Components, Services and Decorators are key building blocks in Angular applications
- A component consists of:
 - Code (JavaScript or TypeScript)
 - Template
 - Decorator (metadata)
- Modules are loaded using an ES2015 module loader
- Angular CLI can be used to create an initial project + more!



Lab

Getting Started with Angular



AngularJS to Angular (Bonus)

Mapping AngularJS Concepts to Angular

- Although Angular (version 2 and higher) is quite different from the original AngularJS framework, many concepts carry over
- This section will provide a general comparison between the “AngularJS” way of doing things and the “Angular” way
- Additional details about the Angular concepts that are shown will be covered later in the course (don’t feel like you need to understand them now! ☺)

<https://angular.io/guide/ajs-quick-reference>

Controllers to Components

AngularJS

```
<body ng-controller="CustomersController as vm">
  <h3>{{ vm.customer.name }}</h3>
</body>

(function() {
  angular
    .module('app')
    .controller('CustomersController',
      CustomersController);

  function CustomersController() {
    var vm = this;
    vm.customer = { id:100, name:'Jed' };
  }
})();
```

Angular

```
<my-customer></my-customer>

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

@Component({
  selector: 'my-customer',
  template: '<h3>{{customer.name}}</h3>'
})
export class CustomerComponent {
  customer = {id: 100, name: 'Jed' };
}
```

Structural Built-In Directives

AngularJS

```
<div ng-if="vm.customers.length">
  <ul>
    <li ng-repeat="cust in vm.customers">
      {{ cust.name }}
    </li>
  </ul>
</div>
```

Angular

```
<div *ngIf="customers.length">
  <ul>
    <li *ngFor="let cust of customers">
      {{ cust.name }}
    </li>
  </ul>
</div>
```

Interpolation

AngularJS

```
<div>{{ vm.customer.name }}</div>
```

Angular

```
<div>{{ customer.name }}</div>
```

One-Way Binding

AngularJS

```
<div ng-bind="vm.customer.name"></div>
```

Angular

```
<div [innerText]="customer.name"></div>
```

Event Binding

AngularJS

```
<button  
  ng-click="vm.submitCustomer()"></button>
```

Angular

```
<button (click)="submitCustomer()"></button>
```

Two-Way Binding

AngularJS

```
<input ng-model="vm.customer.name" />
```

Angular

```
<input [(ngModel)]="customer.name" />
```

Angular Removes Many Directives

AngularJS

```
ng-click="saveCustomer(customer)"  
        ng-focus="handleFocus()"  
        ng-blur="handleBlur()"  
        ng-keyup="handleKeyUp()"
```

Angular

```
(click)="saveCustomer(customer)"  
(focus)="handleFocus()"  
(blur)="handleBlur()"  
(keyup)="handleKeyUp()"
```

Angular Template Concepts Remove 40+
AngularJS Built-In Directives

Services

AngularJS

Factories

Services

Providers

Constants

Values

Angular

Class

Upgrading AngularJS to Angular

- AngularJS cannot be *directly* upgraded to Angular
- Upgrade options:
 - Rewrite the application to use new Angular features (could be a lot of work)
 - Use the UpgradeModule to incrementally upgrade an existing application (allows an upgrade to happen over time)
- Although upgrading AngularJS to Angular is outside the scope of this course, you can get more details about the process at:

<https://angular.io/docs/ts/latest/guide/upgrade.html>



Angular Application Development



The Angular JumpStart Application

Agenda

- Application Overview
- Angular JumpStart in Action
- Application Structure



Application Overview

Angular JumpStart Application Features

- Display and filter customers and orders
- Provide *card* and *grid* views for customers
- Built using TypeScript
- Uses ES2015 modules
- Http and RxJS Observables

Angular JumpStart Application Features (cont...)

- Custom pipes
- Custom directives
- Child Routes
- Lazy Loaded Routes
- Template-driven and Reactive forms

Customers Card Component

The screenshot displays the 'Customer Manager' application interface, specifically the 'Customers' section. The top navigation bar includes links for 'Customer Manager', 'Customers' (which is the active tab), 'Orders', 'About', and 'Login'. Below the navigation is a header with a user icon and the text 'Customers'. Underneath are buttons for 'Card View', 'List View', 'Map View', 'New Customer', and a 'Filter' input field.

The main content area shows a grid of customer cards. Each card contains a customer's name, profile picture, address, and a 'View Orders' link. The cards are arranged in three rows:

- Ted James**: Phoenix, Arizona
View Orders
- Michelle Thompson**: Encinitas, California
View Orders
- Zed Bishop**: Seattle, Washington
View Orders
- Tina Adams**: Chandler, Arizona
View Orders

- Igor Minar**: Dallas, Texas
View Orders
- Brad Green**: Orlando, Florida
View Orders
- Misko Hevery**: Carey, North Carolina
View Orders
- Heedy Wahlin**: Anaheim, California
View Orders

- John Papa**: Orlando, Florida
View Orders
- Tonya Smith**: Atlanta, Georgia
View Orders

Pagination at the bottom shows page numbers 1, 2, 3, and '»'.

Name	Address	Action
Ted James	Phoenix, Arizona	View Orders
Michelle Thompson	Encinitas, California	View Orders
Zed Bishop	Seattle, Washington	View Orders
Tina Adams	Chandler, Arizona	View Orders
Igor Minar	Dallas, Texas	View Orders
Brad Green	Orlando, Florida	View Orders
Misko Hevery	Carey, North Carolina	View Orders
Heedy Wahlin	Anaheim, California	View Orders
John Papa	Orlando, Florida	View Orders
Tonya Smith	Atlanta, Georgia	View Orders

Customers Grid Component

 Customer Manager

Customers Orders About Login

Customers

Card View List View Map View + New Customer Filter:

First Name	Last Name	Address	City	State	Order Total	
 Ted	James	1234 Anywhere St.	Phoenix	Arizona	\$207.98	View Orders
 Michelle	Thompson	345 Cedar Point Ave.	Encinitas	California	\$8.98	View Orders
 Zed	Bishop	1822 Long Bay Dr.	Seattle	Washington	\$229.97	View Orders
 Tina	Adams	79455 Pinetop Way	Chandler	Arizona	\$25.48	View Orders
 Igor	Minar	576 Crescent Blvd.	Dallas	Texas	\$469.98	View Orders
 Brad	Green	9874 Center St.	Orlando	Florida	\$29.98	View Orders

Orders Component

 Customer Manager

 Customer Information

 Customer Details  Customer Orders  Edit Customer

Orders for Ted James

Baseball	\$9.99
Bat	\$19.99

[View all Customers](#)

Customer Edit Component

 Customer Manager

 Edit Customer

Name

Last Name

Address

City

State

Angular JumpStart App in Action

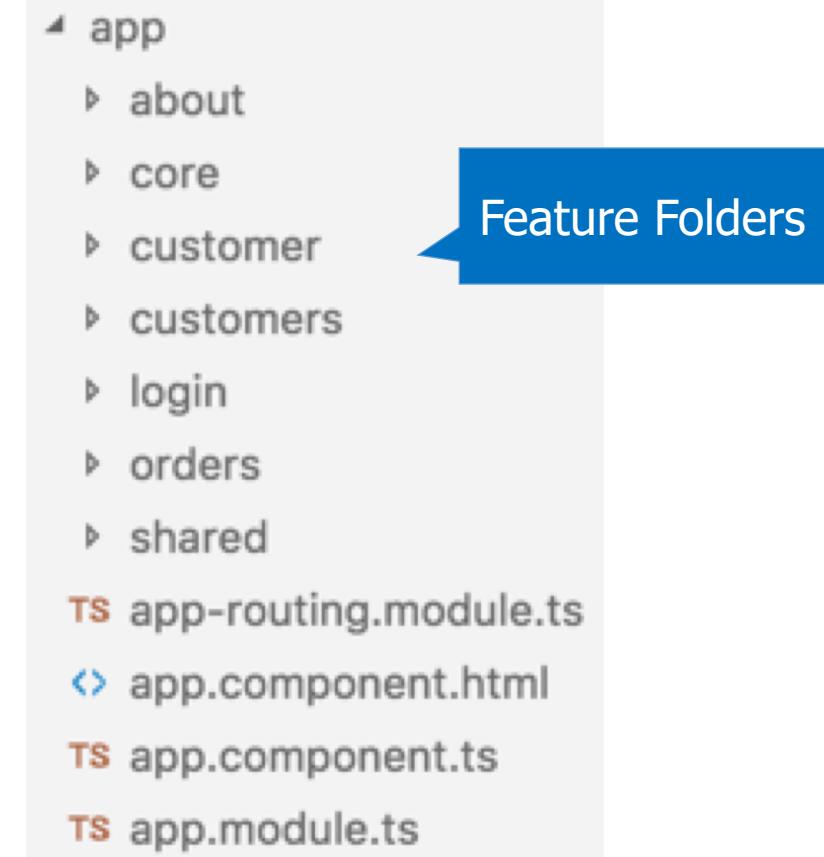
Application Structure

Application Structure Approaches

- Angular applications are generally organized using one of the following approaches:
 - **Convention** – Relies on naming conventions for folders such as "components", "directives", "services", etc.
 - **Feature** – Folders are named based upon specific features in the application
- The Customer Manager application uses the "feature" approach for organizing folders

Customer Manager Application Structure

- Application is organized using a feature-based approach
- Top-level folders are named based on the feature
- Goal is to minimize folder nesting to simplify development & maintenance



Summary

- Angular JumpStart application is built using Angular components
- Provides several different ways to view customer data
- Communicates with the server using "services" and an Http object
- Folders are organized by feature
- Demonstrates several key routing features



Lab

Exploring the Angular JumpStart Application





Angular Application Development



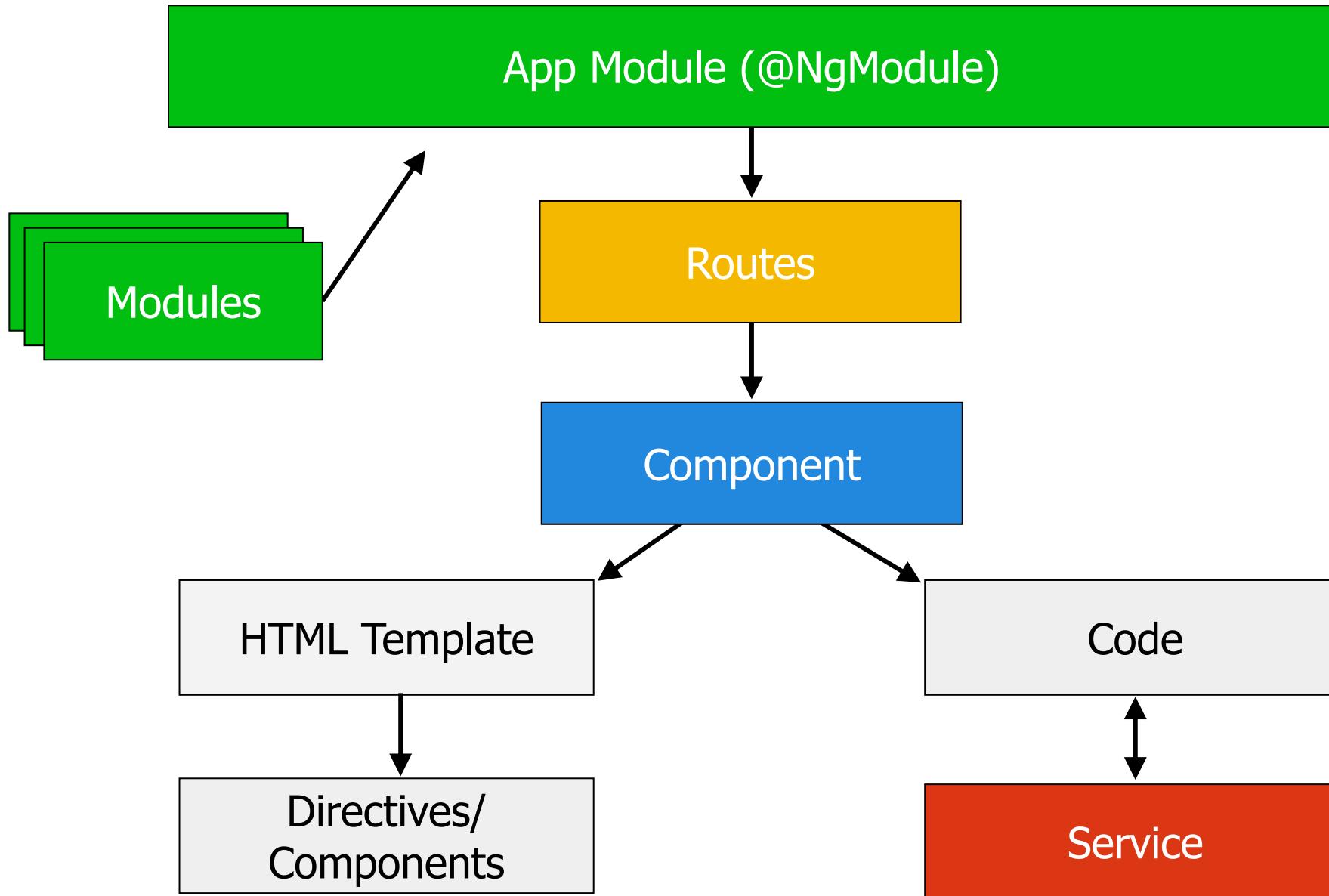
Components and Modules

Agenda

- Components Overview
- Component Lifecycle
- ES2015 Modules
- Angular Modules
- Angular Workflow



The Big Picture



Components Overview

```
ng g component [component_name]
```

What is a Component?

- A component is a reusable object



- Made up of:

Code

HTML
Template

- Has a "selector": <app-customers></app-customers>

Key Component Features

- Building block of Angular apps
- Represents a portion of the screen
- Imports functionality from modules
- Uses a `@Component` decorator to define metadata



Additional Component Features

- Handles user input
- Handles events (click, mouse events, etc.)
- Has a life-cycle
- Delegates to services



What's in a Component Class?

imports

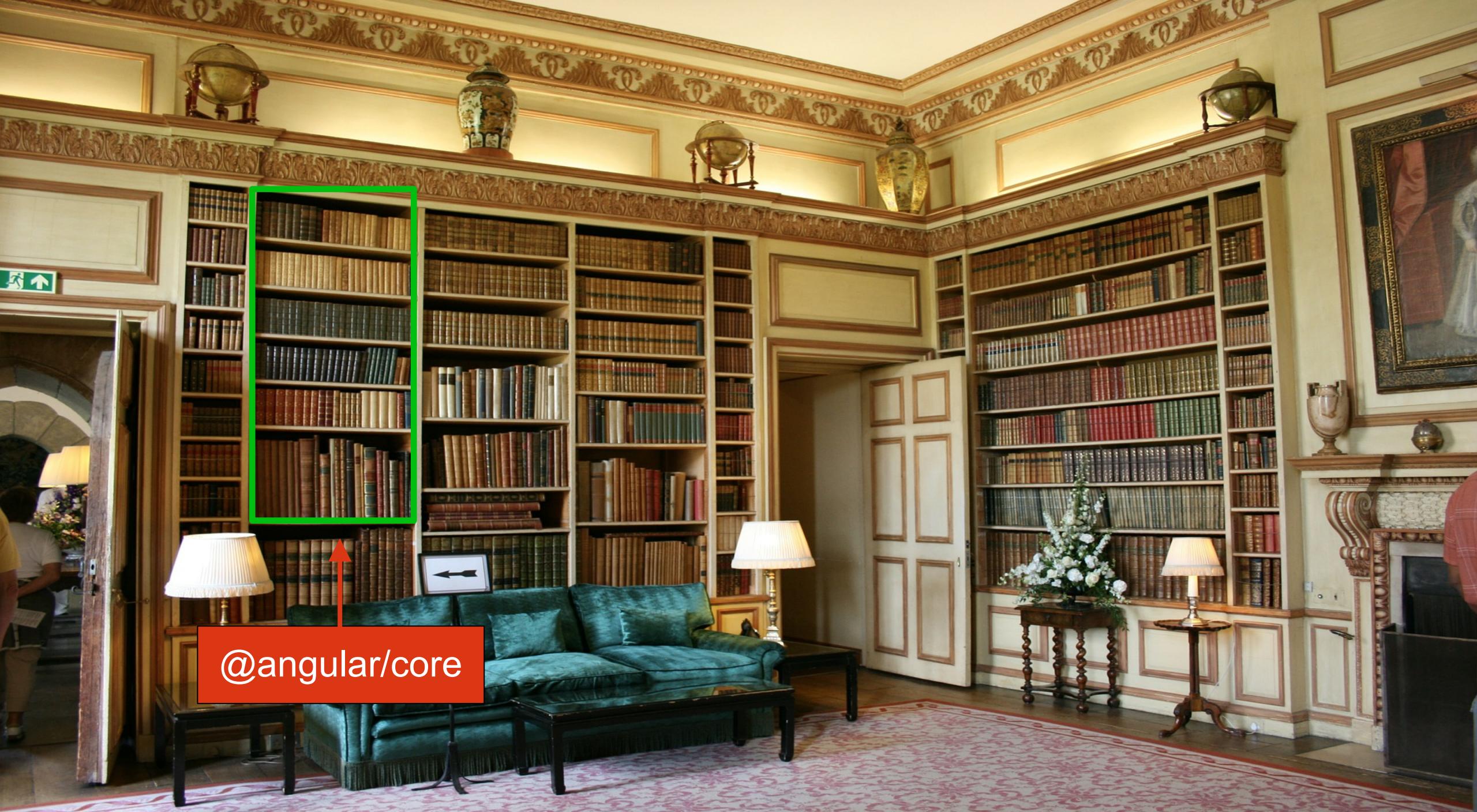
```
import { Component } from '@angular/core';
```

decorator

```
@Component({  
  selector: 'app-customers',  
  templateUrl: './customers.component.html'  
})
```

class

```
export class CustomersComponent {  
}
```



@angular/core

The @Component Decorator

- Decorators provide metadata for a component class
- `@Component` imported from **@angular/core** module
- Key properties:

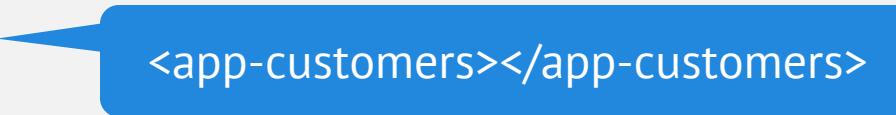
Property	Description
selector	Defines the selector that triggers instantiation of the component (ex: 'app-customers' = <app-customers></app-customers>
template	Defines the inline template used by the component
templateUrl	Defines the file template used by the component

Using the @Component Decorator

@Component defines metadata for a component

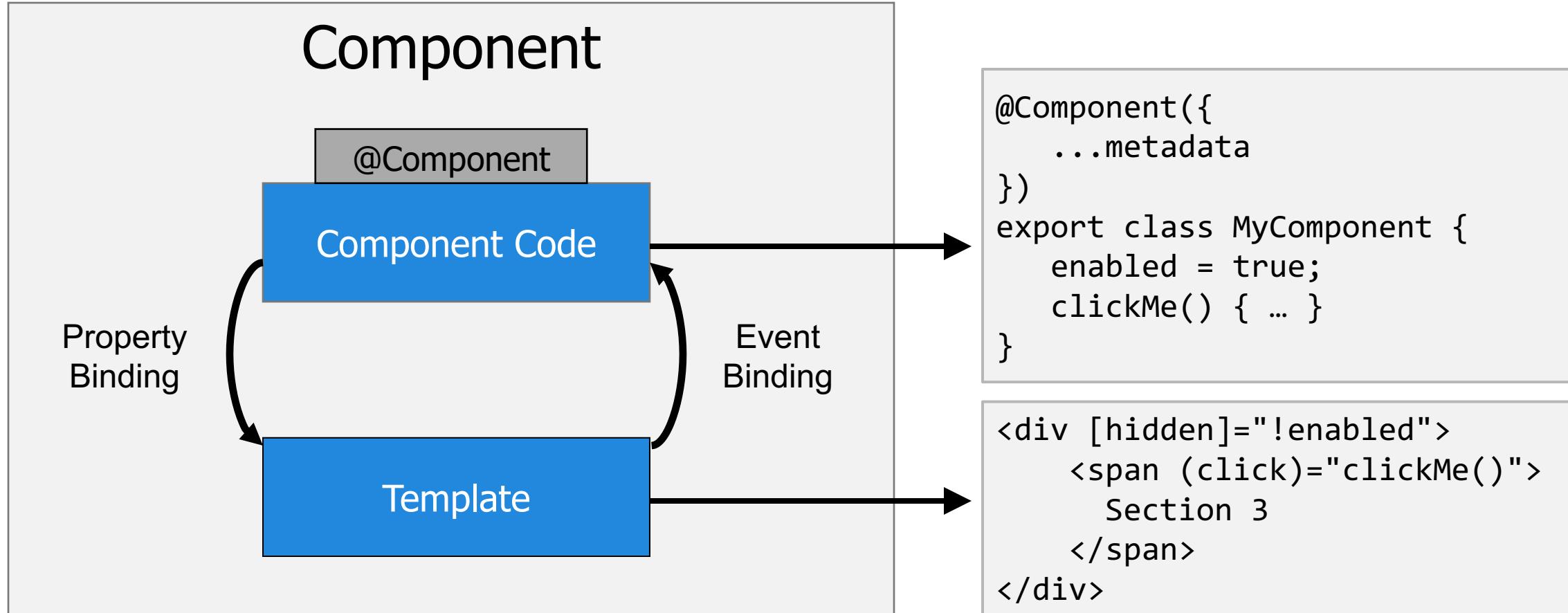
```
import { Component } from '@angular/core';

@Component({
  selector: 'app-customers',
  template: `
    <h1>Customers</h1>
  `
})
export class CustomersComponent {
```



<app-customers></app-customers>

Component Code and Templates

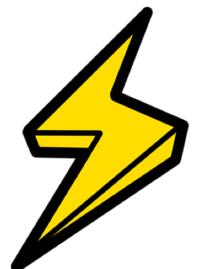


Components in Action

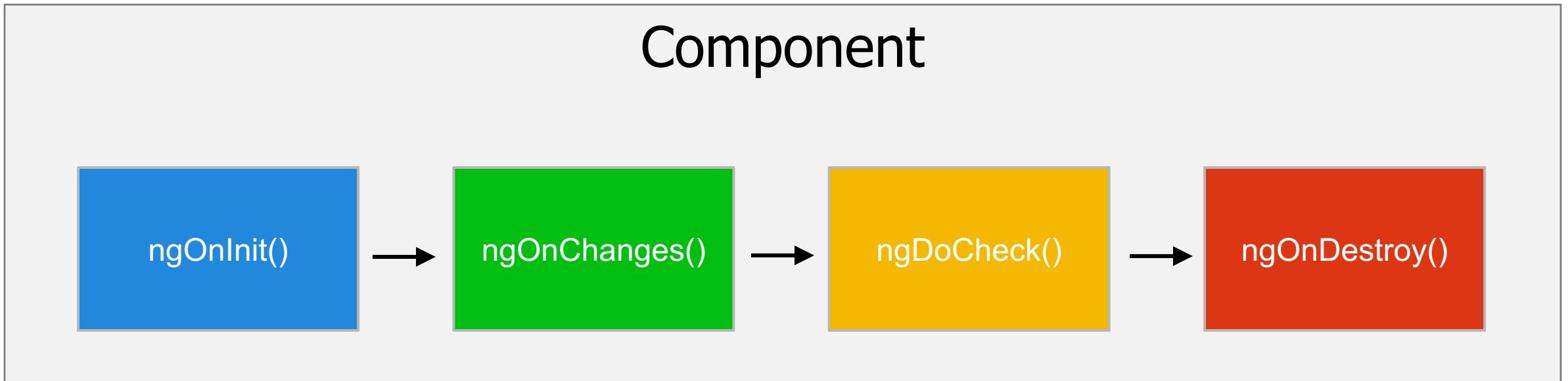
Component Lifecycle

The Component Lifecycle

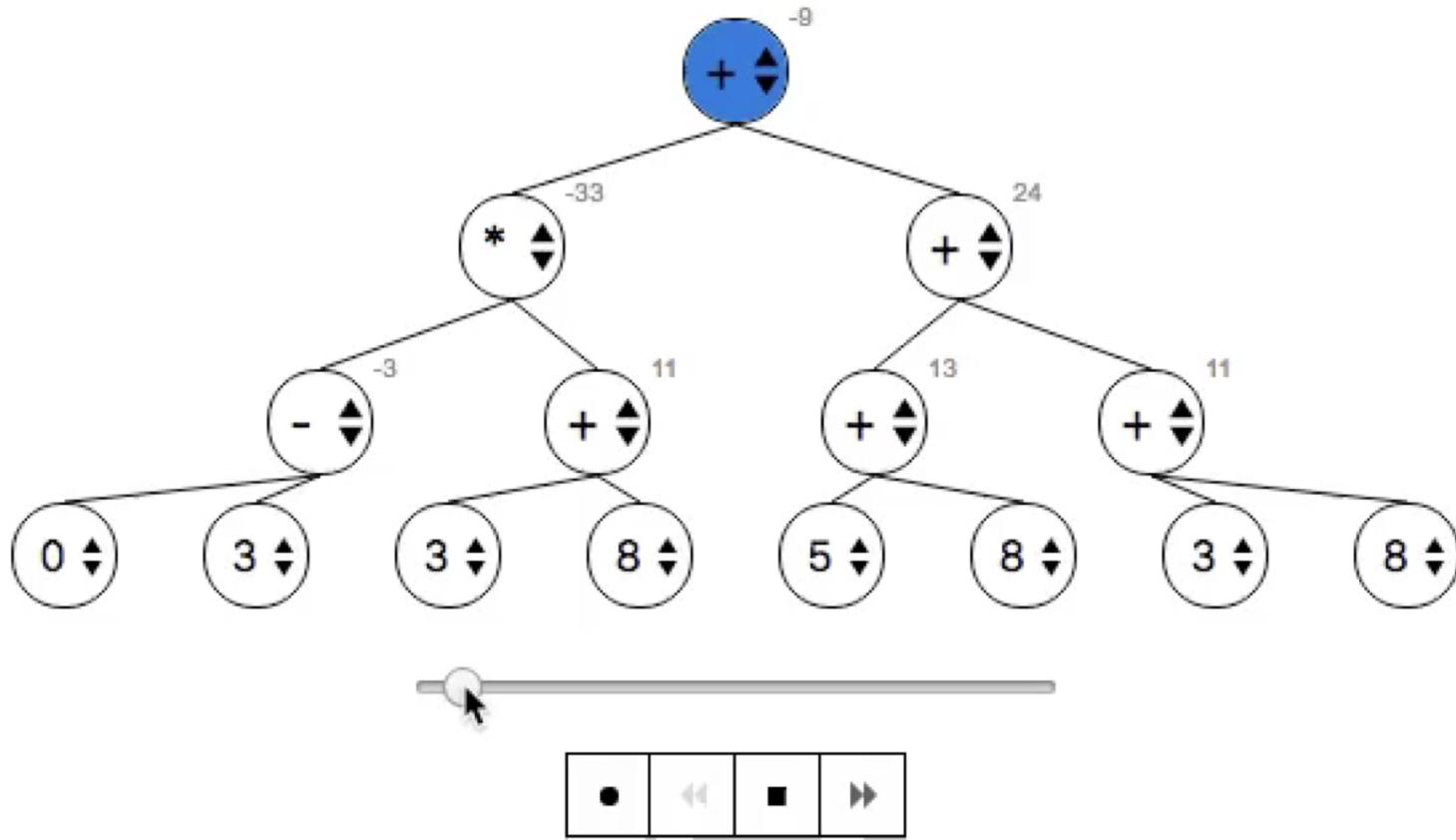
- Components have a lifecycle that is managed by Angular
- Detect and "hook" into key lifecycle moments:
 - Changes
 - Initialization
 - The component's view and child views are initialized
 - Content is projected into the view
 - Destruction



Key Component Lifecycle Hooks



<https://angular.io/guide/lifecycle-hooks>



```
-> ngDoCheck called  
ngDoCheck called  
ngDoCheck called  
ngDoCheck called
```

Monitoring OnInit and OnDestroy

Components can tie into the component lifecycle

Each lifecycle "moment" has an associated interface

```
import { Component, OnInit, OnDestroy } from '@angular/core';
@Component({
  selector: 'app-component',
  template: `...
})
export class AppComponent implements OnInit, OnDestroy {

  ngOnInit() { console.log('Component initialized'); }

  ngOnDestroy() { console.log('Component destroyed'); }
}
```

Interfaces provided by Angular

Initialize variables with values,
load data, etc. here

ES2015 Modules

What's an ES2015 Module?

- Part of the ES2015 specification
- Modules (files) act as code containers
- Run in strict mode
- Pull members out of the global scope
- Can **export** and **import**

ES2015 Modules: Exporting

Classes, functions and variables can be exported using the **export** keyword

Other modules can import DataService

data.service.js

```
export class DataService {  
    ...  
}
```

ES2015 Modules: Importing

Module members can be imported using the **import** keyword

customers.component.ts

```
import { Component } from '@angular/core';
import { DataService } from '../shared/services/data.service';
...
```

Imported module is relative to current file.

Enabling Module Loading in Browsers

- A "module loader" is needed to load ES2015 modules in the browser (although newer browsers are starting to add this functionality)
- Key module loaders include:
 - System.js
 - Webpack ("walks" imports and creates script bundles)



Angular Modules

The Role of Angular Modules

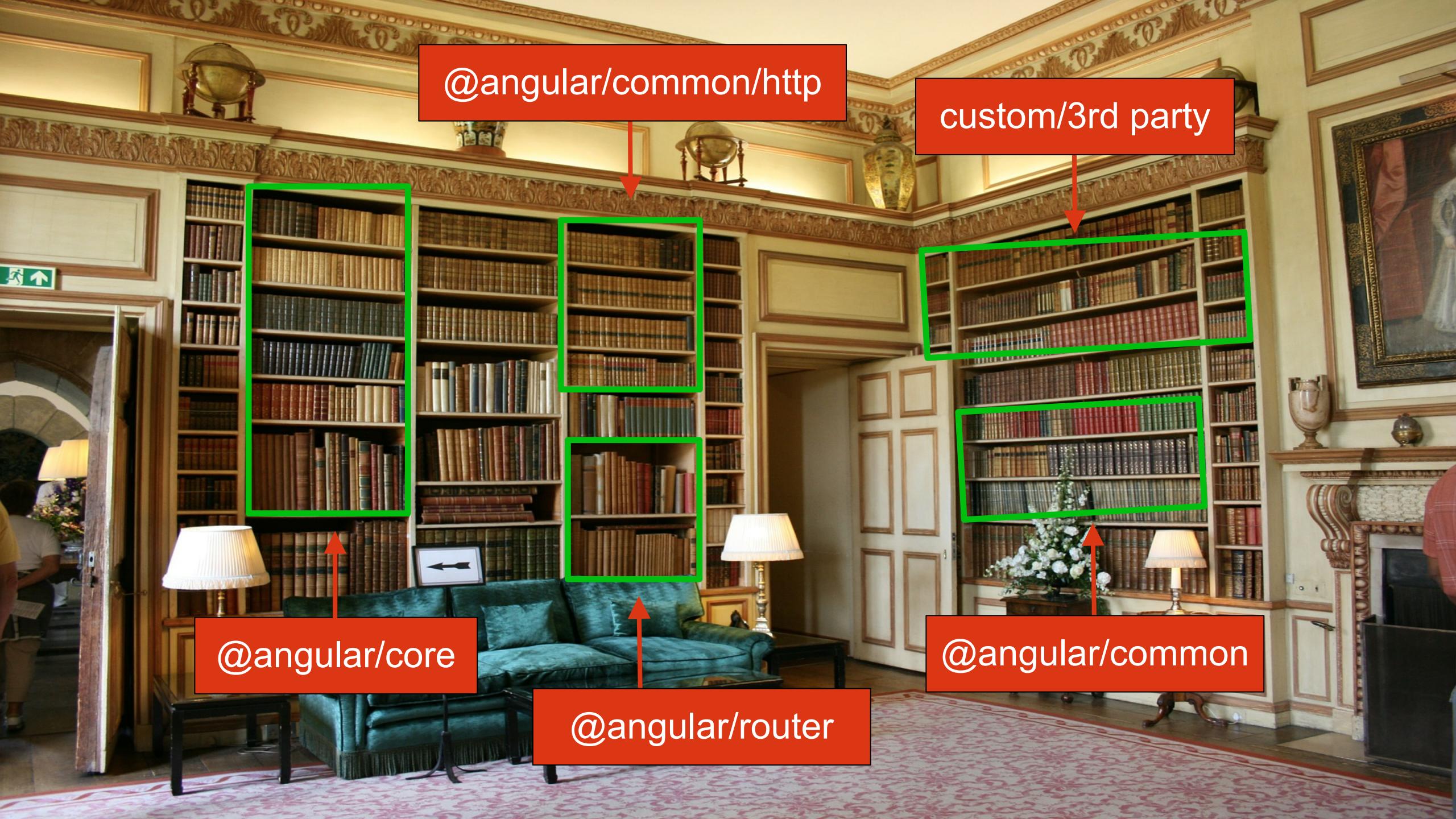
- Organize an application into "cohesive blocks of functionality"
- Group related components, directives and pipes together
- Import other modules (Angular, custom or 3rd party)
- Export functionality used by other modules
- Define service providers used in an application



The @NgModule Decorator

- `@NgModule` marks a class as a "module"
- Provides the following properties

Property	Description
bootstrap	Used to define the root component that hosts other components in the application
declarations	The view classes (components, directives, pipes) that belong to the module
exports	Subset of declarations that should be useable in the component templates of other modules.
imports	Other modules that should be imported into this module
providers	Services that this module adds to the global collection of services



Using the @NgModule Decorator

app.module.ts

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

import { AppComponent } from './app.component';
import { CustomersComponent } from './customers/customers.component';
import { DataService } from './shared/data.service';

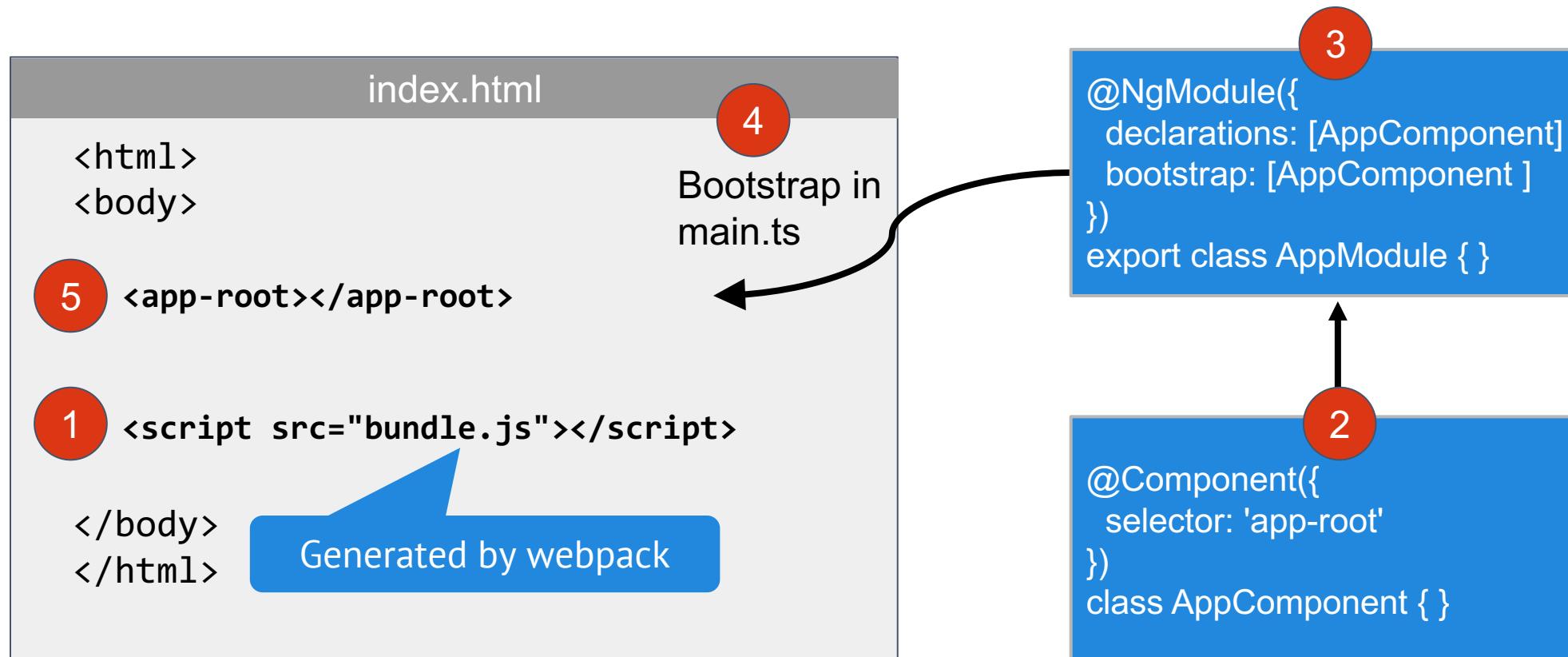
@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ AppComponent, CustomersComponent ],
  providers:    [ DataService ],
  bootstrap:   [ AppComponent ]
})
export class AppModule { }
```

Angular Workflow

The Angular Workflow

- 1 Add application scripts
- 2 Create an Angular component
- 3 Add the component into an Angular module
- 4 Pass the module to bootstrapModule()
- 5 Use the root component selector tag in a page

Getting Started with Angular



Summary

- Components are the key building blocks of Angular applications
- A component has a lifecycle that can be monitored
- Modules act as code containers
- A root component must be "bootstrapped"



Lab

Components and Modules





Angular Application Development



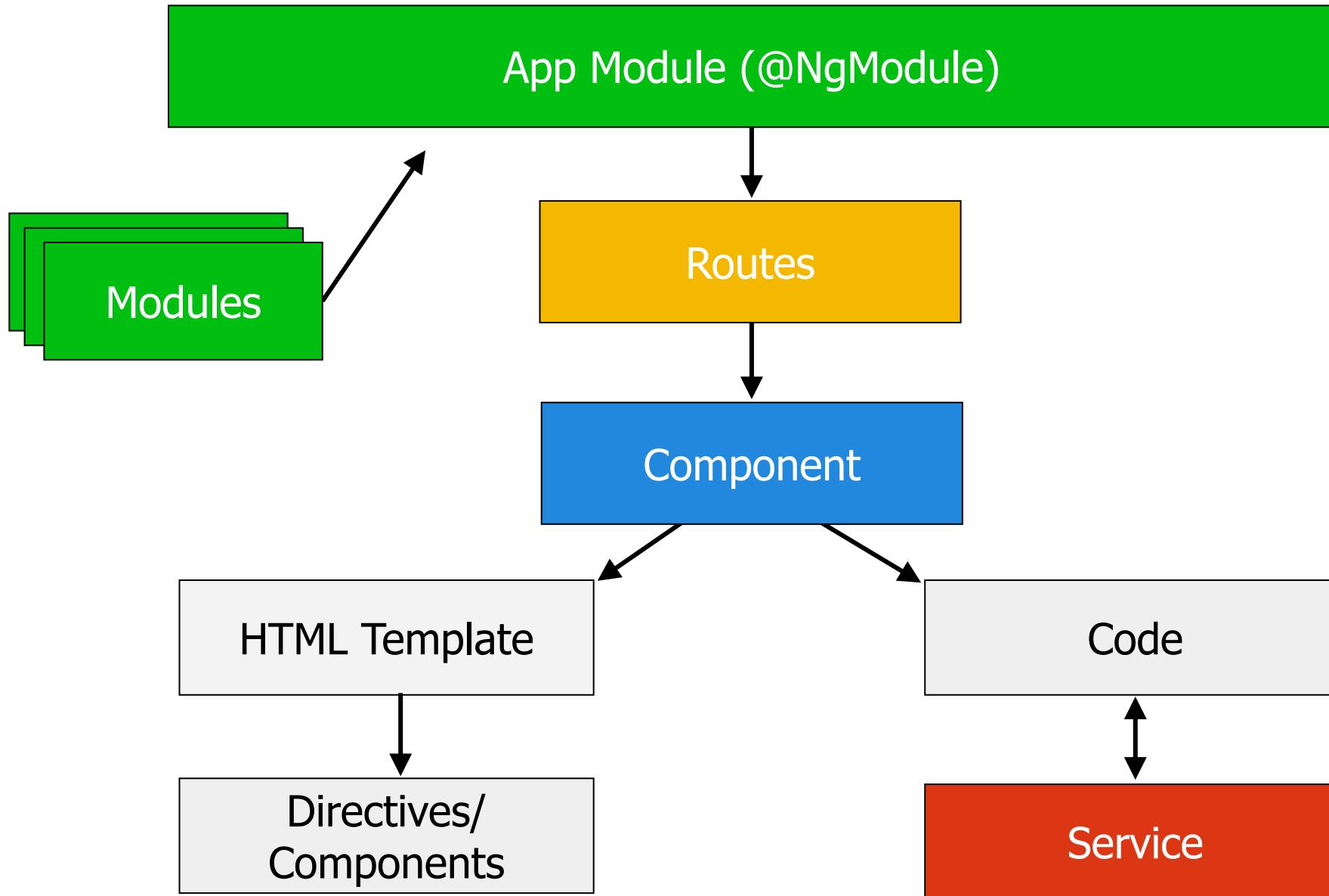
Template Expressions and Pipes

Agenda

- Interpolation and Expressions
- Working with Pipes
- Creating a Custom Pipe

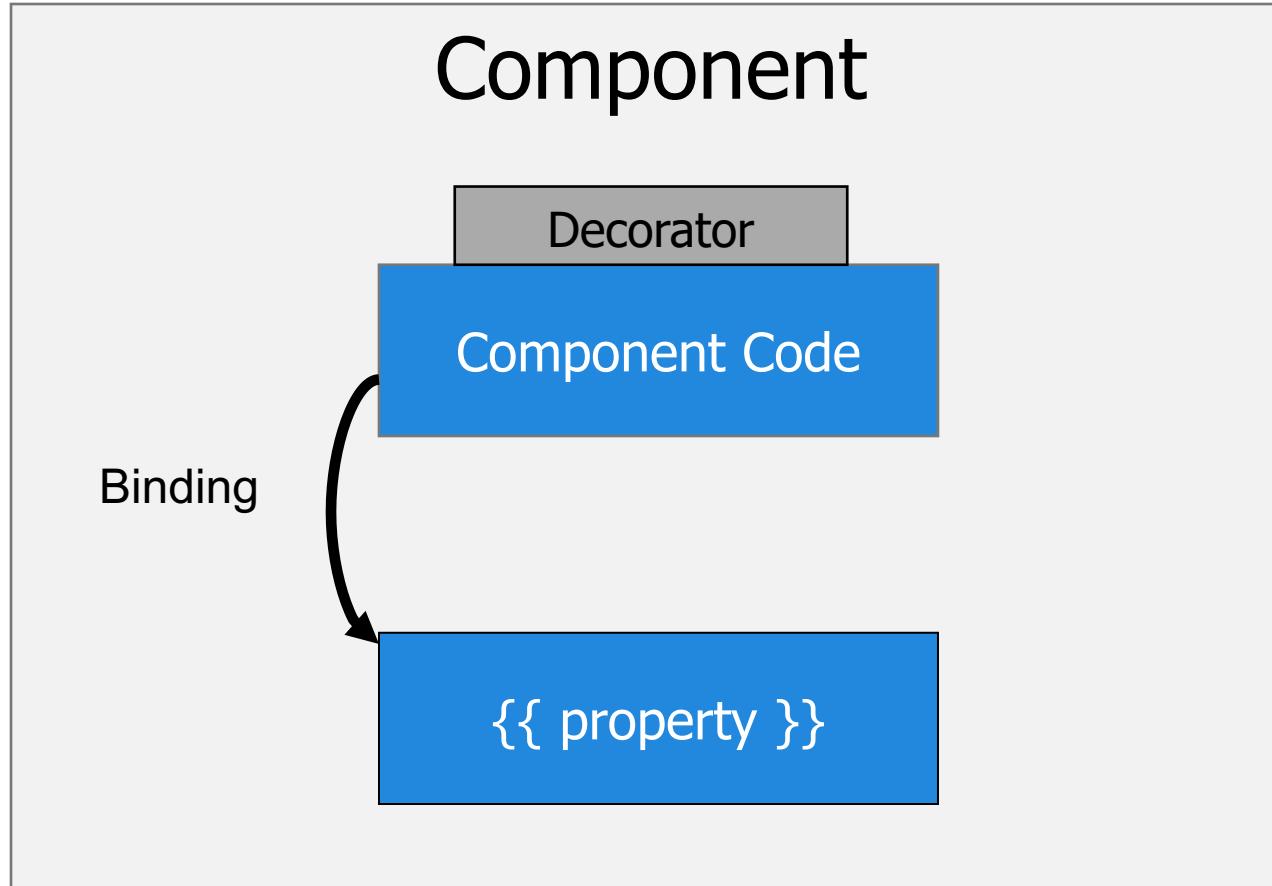


The Big Picture



Interpolation and Expressions

Binding Data to a Template



Template Data Binding Syntax

- Components bind data to templates and handle events that pass data back to the component
- Templates use "expressions"

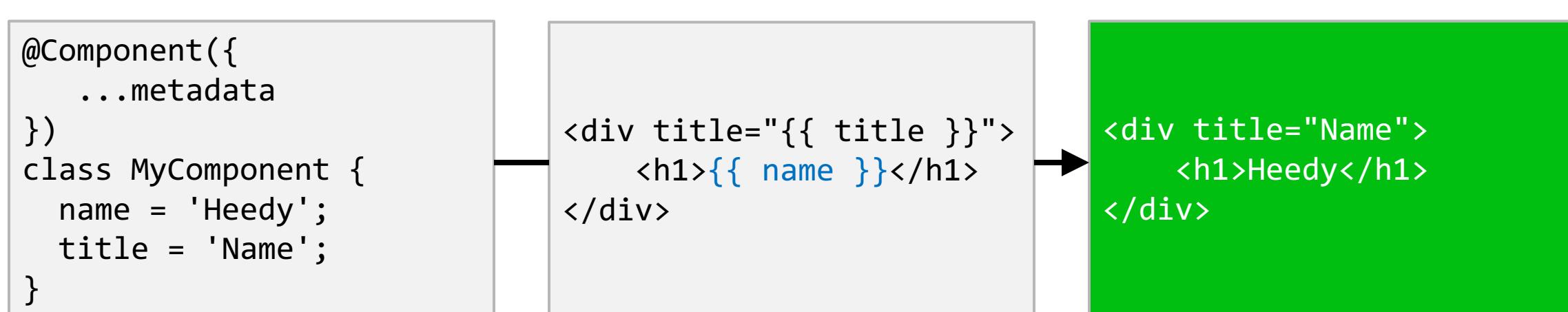
Syntax Example	Description
<code>{{ propertyName }}</code>	Bind to and display property value
<code>{{ 2 + 2 }}</code>	Template expression
<code>[target]="expression"</code>	Property binding
<code>(target)="statement"</code>	Event binding
<code>[(target)]="expression"</code>	Two-way binding

Interpolation and Template Expressions

- Interpolation is the process of binding data into a template using expressions
- Template expression examples:

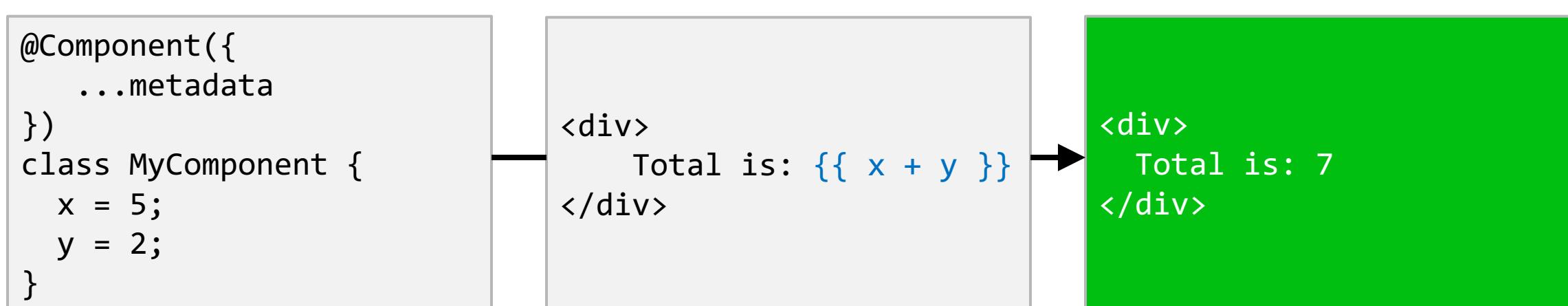
```
{{ propertyName }}
```

```
{{ x + y }}
```



Template Expressions

- Template expressions produce a value
- Key expression rules:
 - Reference component instance members only (no globals)
 - Should not change application state
 - Should be simple and fast to execute



Expressions in Action

Working with Pipes

ng g pipe pipe_name

Working with Pipes

- Angular pipes transform data:
 - Transform data (numbers, dates, currency, percentages)
 - Transform strings (lowercase, uppercase)
 - Handle async data (promises or observables)
 - Write custom pipes



Using the uppercase/lowercase Pipes

The uppercase/lowercase pipes transform string values

```
@Component({  
  selector: 'customer',  
  template: `  
    <div>{{ name | uppercase }}</div>  
    <div>{{ customerSince | date | lowercase }}</div>  
  `  
})  
export class CustomerComponent {  
  name = 'Jim';  
  customerSince: any = new Date(2014, 7, 10);  
}
```



Using the Currency Pipe

The currency pipe transforms values to a currency format

9.99 → \$9.99

```
@Component({  
  selector: 'customer',  
  template: `  
    <div>{{ price | currency:'USD':'symbol' }}</div>  
  `  
})  
export class CustomerComponent {  
  price = 9.99;  
}
```

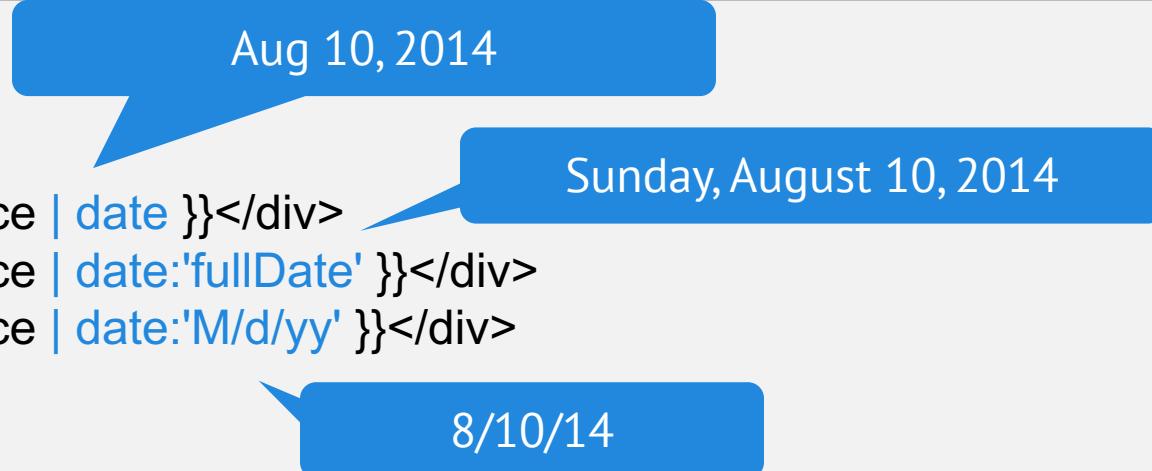


Using the date Pipe

The date pipe transforms values to date/time formats

new Date(2014, 7, 10) → August 10, 2014

```
@Component({  
  selector: 'customer',  
  template: `  
    <div>{{ customerSince | date }}</div>  
    <div>{{ customerSince | date:'fullDate' }}</div>  
    <div>{{ customerSince | date:'M/d/yy' }}</div>  
  `,  
}  
)  
export class CustomerComponent {  
  customerSince = new Date(2014, 7, 10);  
}
```



Pipes in Action

Creating a Custom Pipe

Creating a Custom Pipe

- Custom pipes can be created when data needs to be formatted in a unique way
- Create a class that uses the `@Pipe` decorator (and optional `PipeTransform` interface)
- Add a `transform()` function that accepts a value and returns a custom value



Using the @Pipe Decorator

Custom pipes can be created using the @Pipe decorator

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

@Pipe({name: 'trim'})
export class TrimPipe implements PipeTransform {
  transform(value: any) {
    if (!value) {
      return '';
    }
    return value.trim();
  }
}
```

```
@Component({
  selector: 'customer',
  template: `<h1>{{ name | trim }}</h1>`
})
export class CustomerComponent {
  name: string = ' Fred';
}
```

Declaring a Custom Pipe

Custom pipes can be used after they are added to a module

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

import { AppComponent } from './app.component';
import { DashReplacerPipe } from './dash-replacer.pipe';

@NgModule({
  imports: [ BrowserModule ],
  declarations: [ AppComponent, DashReplacerPipe ],
  bootstrap: [ AppComponent ]
})
export class AppModule {}
```

Custom Pipes in Action

Summary

- Templates use interpolation and expressions to render data to the screen
- Angular supports interpolation and expressions:
 - {{ property }}
 - {{ expression }}
- Pipes provide a way to transform data passed into a template



Lab

Interpolation, Expressions and Pipes





Angular Application Development



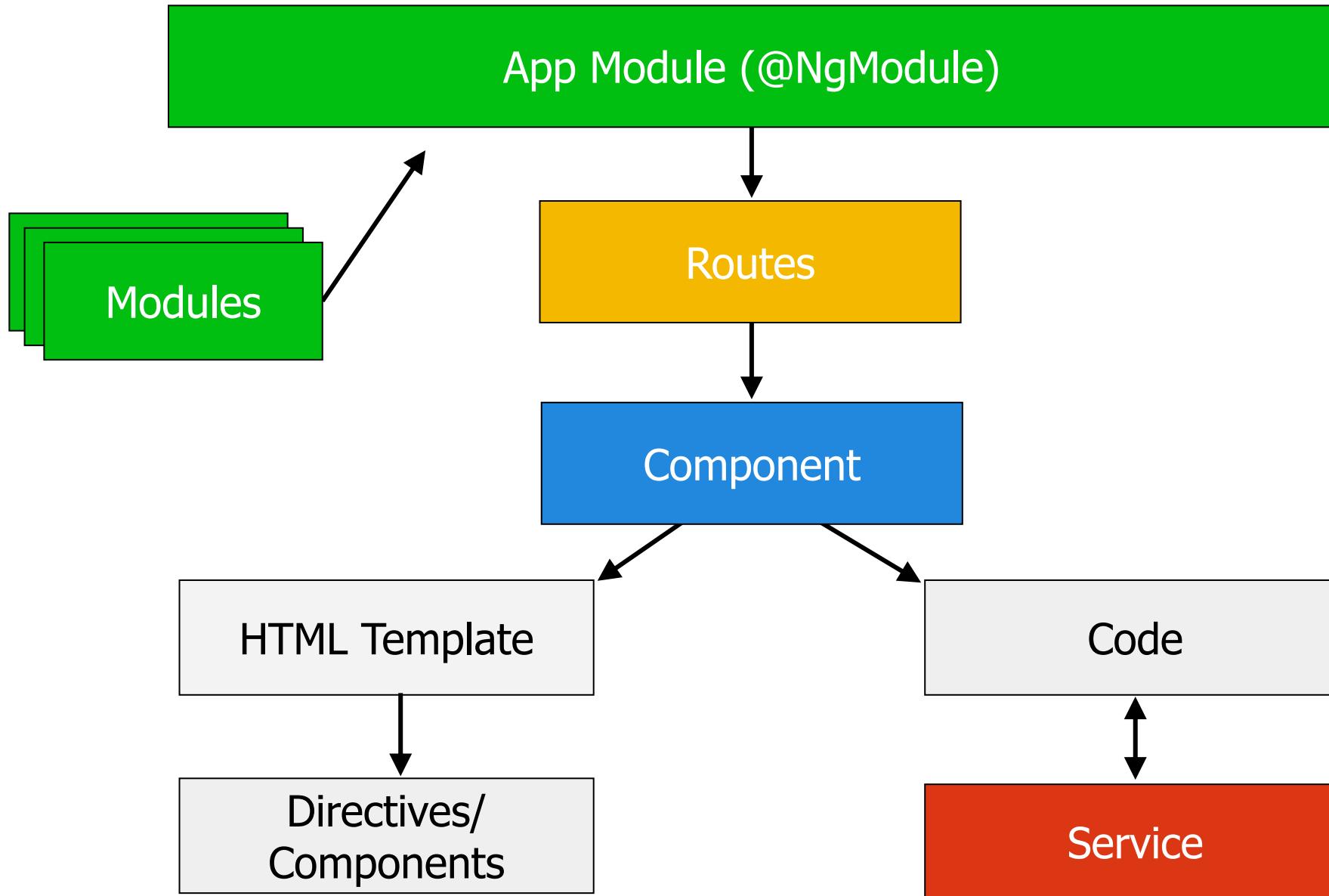
Component Properties,
Directives and Data Binding

Agenda

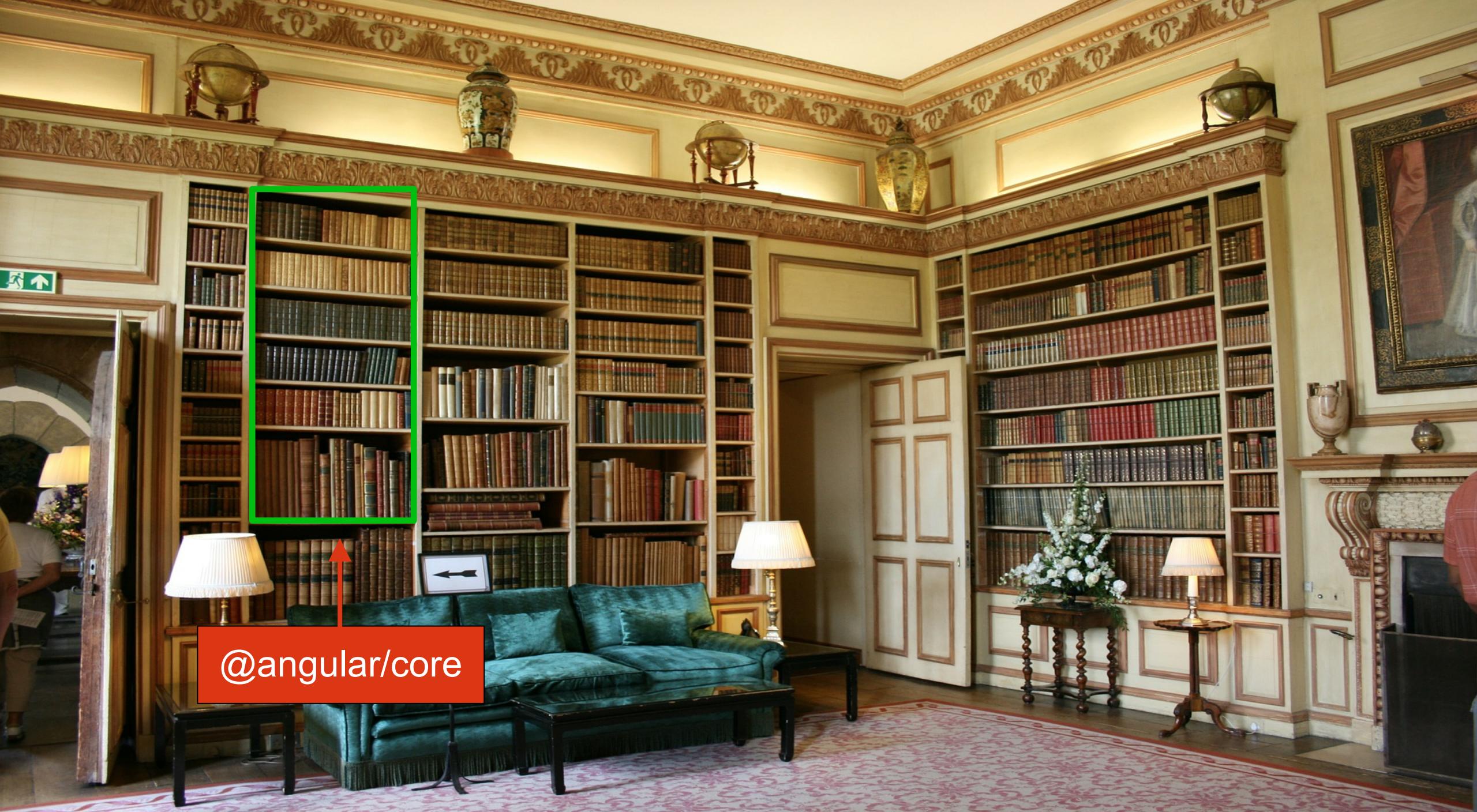
- Property and Event Binding
- Input Properties
- Output Properties
- Angular Directives
- Two-Way Data Binding
- Change Detection



The Big Picture

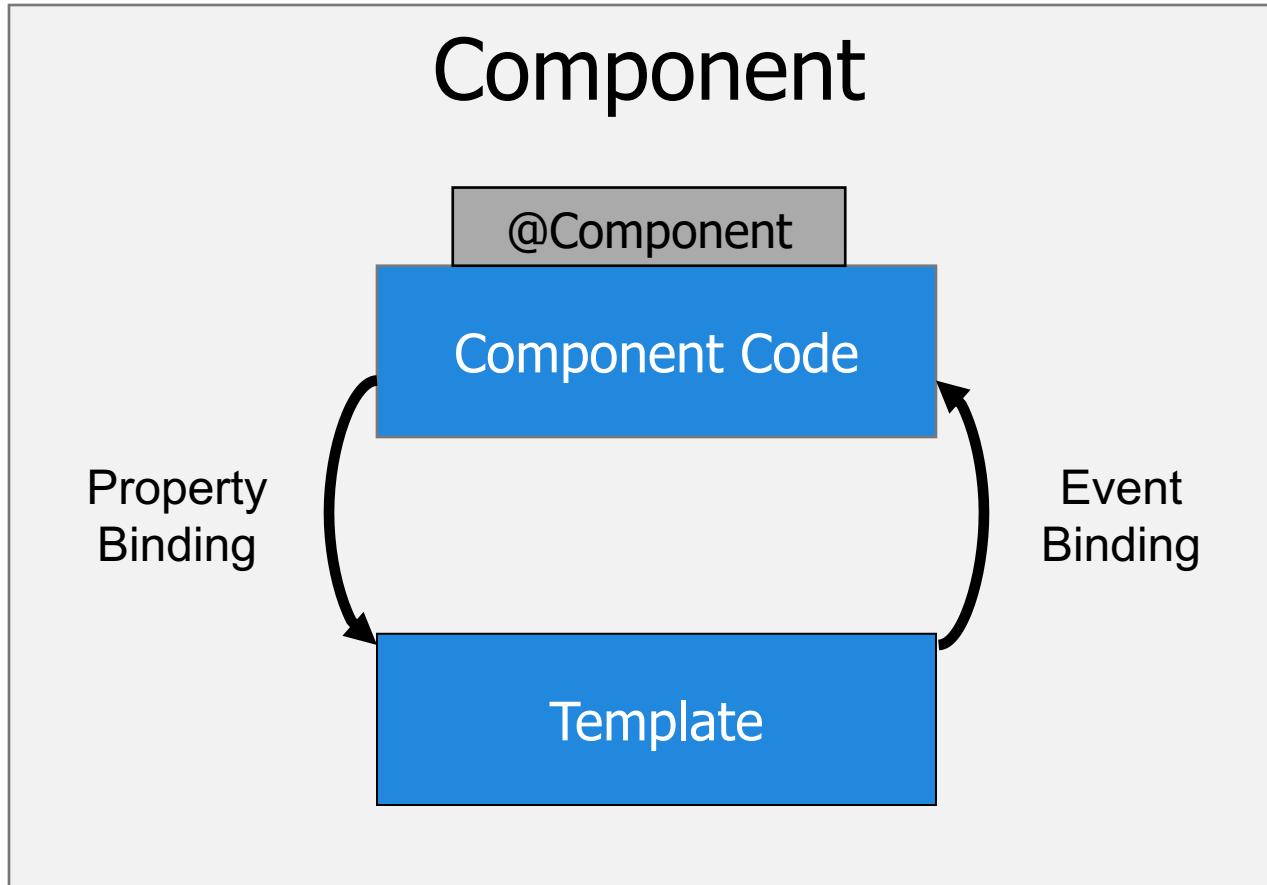


Property and Event Binding

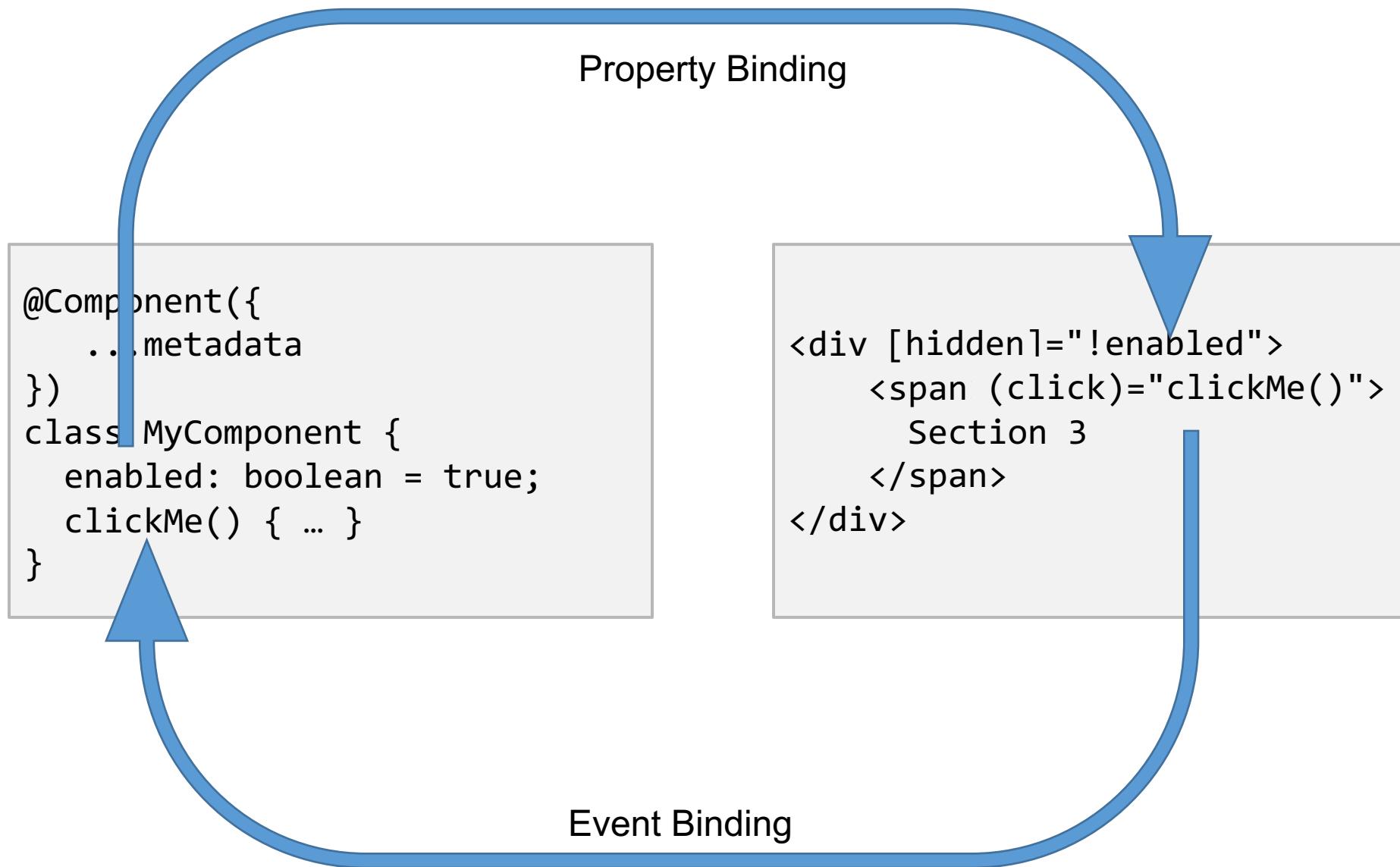


@angular/core

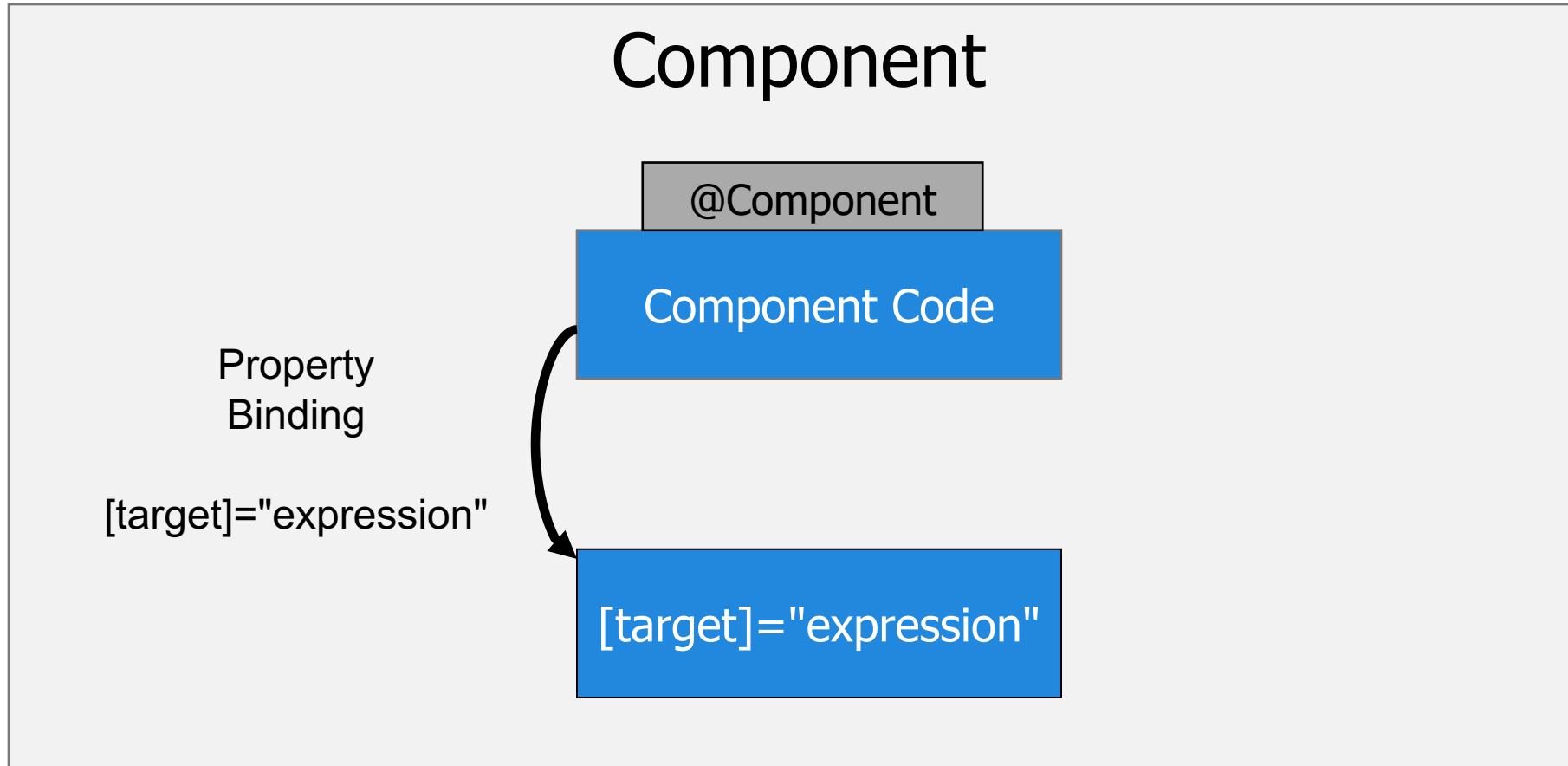
Component Templates



Property and Event Bindings



Property Binding



Property Binding Syntax

- Create one-way bindings to DOM properties using **[target]** or **bind-target** syntax

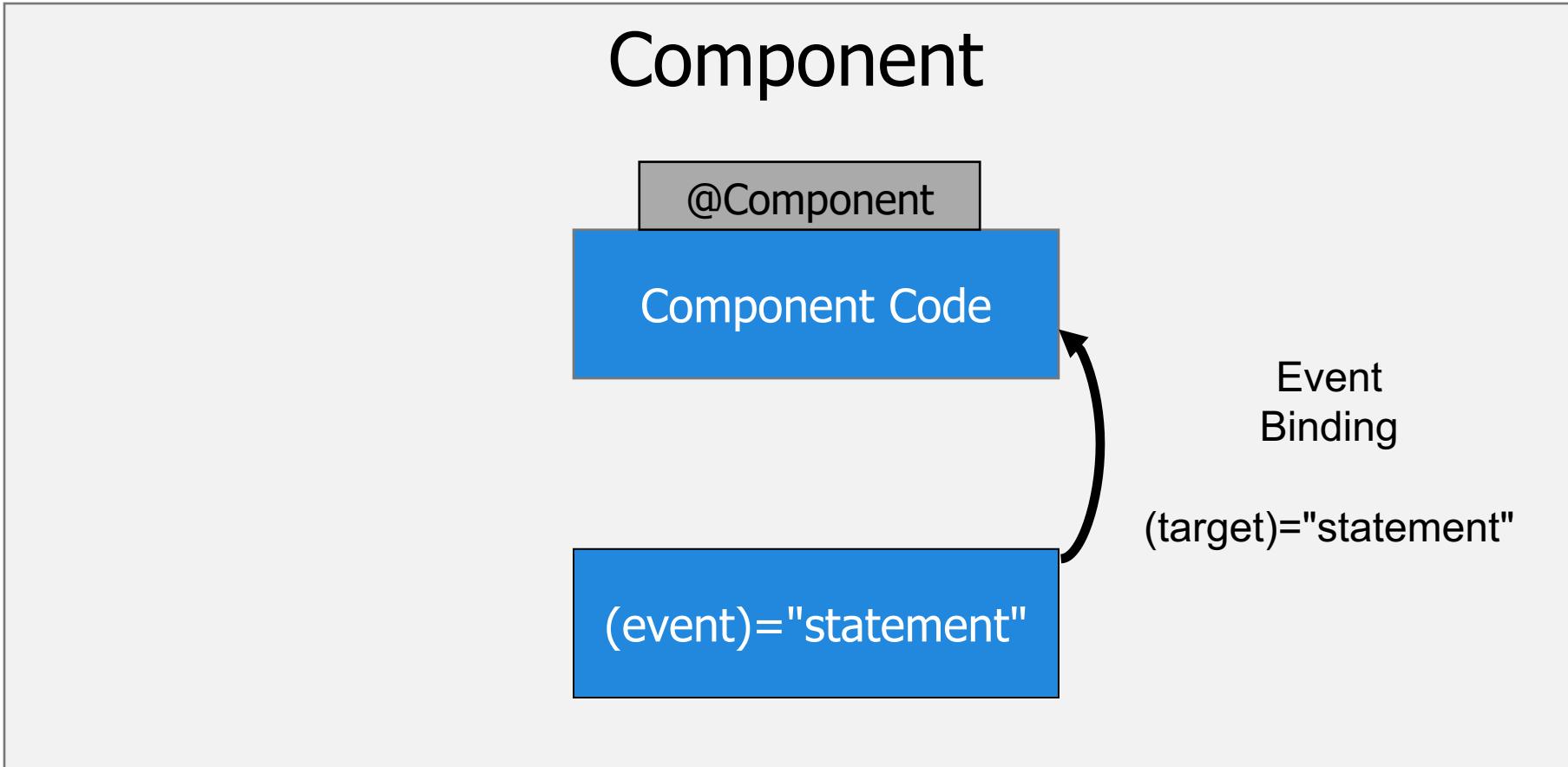
```
<button [disabled]="isDisabled">...</button>
```

```
<button bind-disabled="isDisabled">...</button>
```

Property Binding Examples

```
<img [src]="customer imagePath">  
  
<div [hidden]="!isVisible">...</div>  
  
<div [class.active]="isActive">...</div>  
  
<div [style.color]="textColor" [attr.aria-label]="text">...</div>
```

Event Binding



Event Binding Syntax

- Handle an event using the **(event)** or **on-event** syntax

```
<th (click)="sort('orderTotal')">Order Total</th>
```

```
<th on-click="sort('orderTotal')">Order Total</th>
```

Property and Event Binding in Action

Input Properties

Component Input Properties

Input properties allow the consumer of a component to pass in a value

Defined using the `@Input` decorator

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

@Component({
  selector: 'customer',
  template: '<h1>{{ name }}</h1>'
})
export class CustomerComponent {

  @Input() name: string;
}
```

Using the component:

```
<customer [name]="'James'"></customer>
```

Binding in a Component Hierarchy

```
@Component({  
  selector: 'parent',  
  template: 'parent.component.html'  
})  
export class ParentComponent {  
  fullName: string = 'Fred';  
}
```

parent.component.html

```
<child [name]="fullName"></child>
```

```
@Component({  
  selector: 'child',  
  template: '<h1>{{ name }}</h1>'  
})  
export class ChildComponent {  
  @Input() name: string;  
}
```

ChildComponent Output

```
<h1>Fred</h1>
```

Input Properties and OnChanges

The OnChanges lifecycle moment can be used to monitor Input property changes

```
import { Component, Input, OnChanges } from '@angular/core';
@Component({ ... })
export class MyComponent implements OnChanges {
  @Input() name: string;

  ngOnChanges(changes: SimpleChanges) {
    let prop = changes['name'];
    console.log(`Current: ${prop.currentValue}`);
    console.log(`Previous: ${prop.previousValue}`);
  }
}
```

Monitor input properties for changes.

Using a Set Block to Detect Input Property Changes

Input properties can have get and set blocks

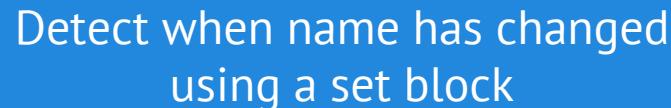
A set block can be used to detect when the property changes

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

@Component({ ... })
export class MyComponent {
  private _name: string;

  @Input() public get name(): string { return this._name; }
  public set name(value: string) { this._name = value; }

}
```



Detect when name has changed
using a set block

Input Properties in Action

Output Properties

Component Output Properties

Output properties allow the consumer of a component to be notified as a component changes

Defined using the `@Output` decorator

```
import { Component, Input, Output, EventEmitter } from '@angular/core';
@Component({
  selector: 'customer',
  template: '<div (click)="changeName()">{{ name }}</div>'
})
export class CustomerComponent {
  @Input() name: string;

  @Output() nameChanged = new EventEmitter<string>();
  changeName() {
    this.name = 'Fred';
    this.nameChanged.emit(this.name);
  }
}
```

Consumer will be notified if name
changes (more on this later!)

Binding to an Output Property

```
@Component({  
  selector: 'parent',  
  templateUrl: 'parent.component.html'  
})  
export class ParentComponent {  
  changed(name: string) { ... }  
}
```

parent.component.html

```
<child (nameChanged)="changed()"></child>
```

```
@Component({  
  selector: 'child',  
  template: '<h1>{{ name }}</h1>'  
})  
export class ChildComponent {  
  @Output() nameChanged: EventEmitter<...>;  
}
```

Output Properties in Action

Angular Directives

Angular Directives

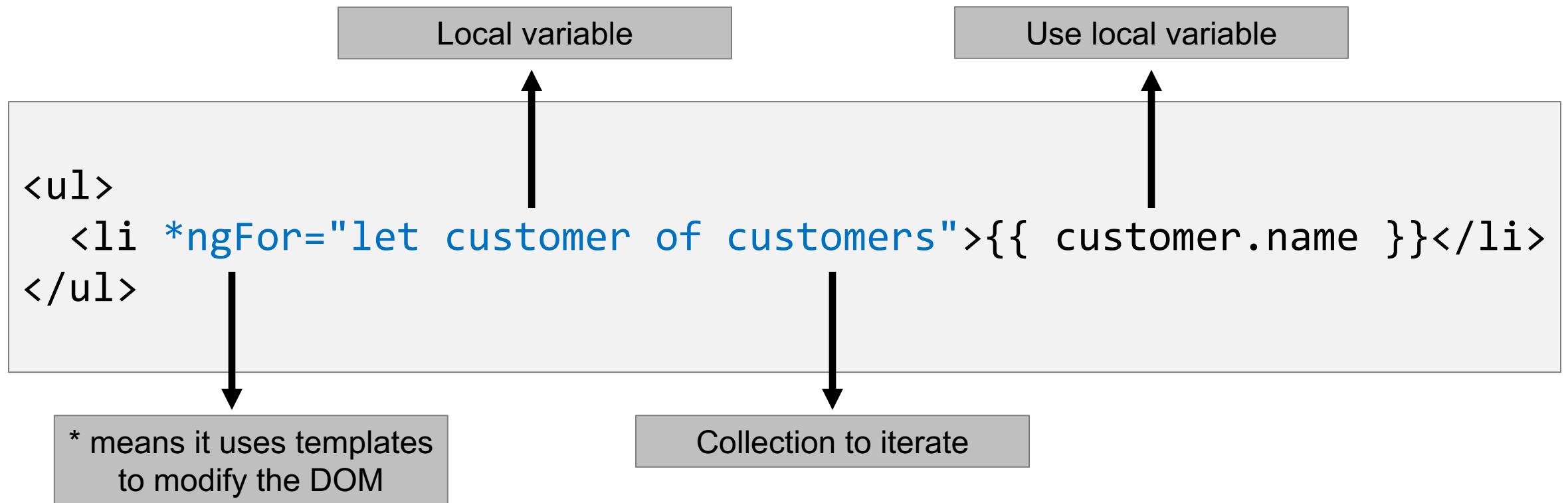
- Angular provides several key directives that can be used in templates:

Syntax Example	Description
ngFor	A repeater directive used to iterate through a collection
ngIf/ngSwitch	Adds and removes DOM sub-trees
ngClass	Adds or removes CSS classes
ngStyle	Adds or removes element styles
ngModel	Binds a model object to a form control

<https://angular.io/api?type=directive>

The ngFor Directive

- ngFor iterates through a collection and creates a template for each item

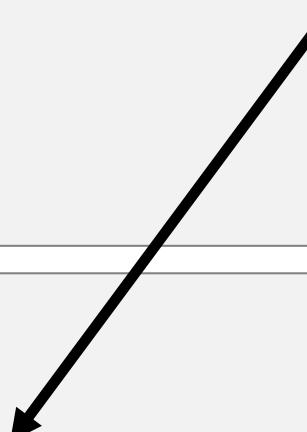


Enhancing ngFor Performance

Minimize the number of DOM updates by adding "trackBy" to ngFor in cases where items are added/removed by the user:

```
<ul>
  <li *ngFor="let customer of customers;trackBy:customerTrackBy">
    {{ customer.name }}
  </li>
</ul>
```

```
@Component({...})
export class MyComponent {
  customers: ICustomer[];
  customerTrackBy(index: number, customer: ICustomer) {
    return customer.id;
  }
}
```



The ngIf Directive

- ngIf adds an element subtree to the DOM if an expression is truthy

```
<div *ngIf="customer">{{ customer.name }}</div>
```



* means it uses templates
to modify the DOM

ngIf and else

- The ngIf directive provides support for an “else” template in Angular v4+

```
<div *ngIf="customer; else elseBlock">{{ customer.details }}</div>
<ng-template #elseBlock>No customer found</ng-template>
```

```
<div *ngIf="orders; then thenBlock else elseBlock"></div>
<ng-template #thenBlock>...Render Orders</ng-template>
<ng-template #elseBlock>No orders found</ng-template>
```

The ngClass Directive

- Standard property binding can be used to work with CSS classes:

```
<div [class.active]="enabled">{{ customer.name }}</div>
```

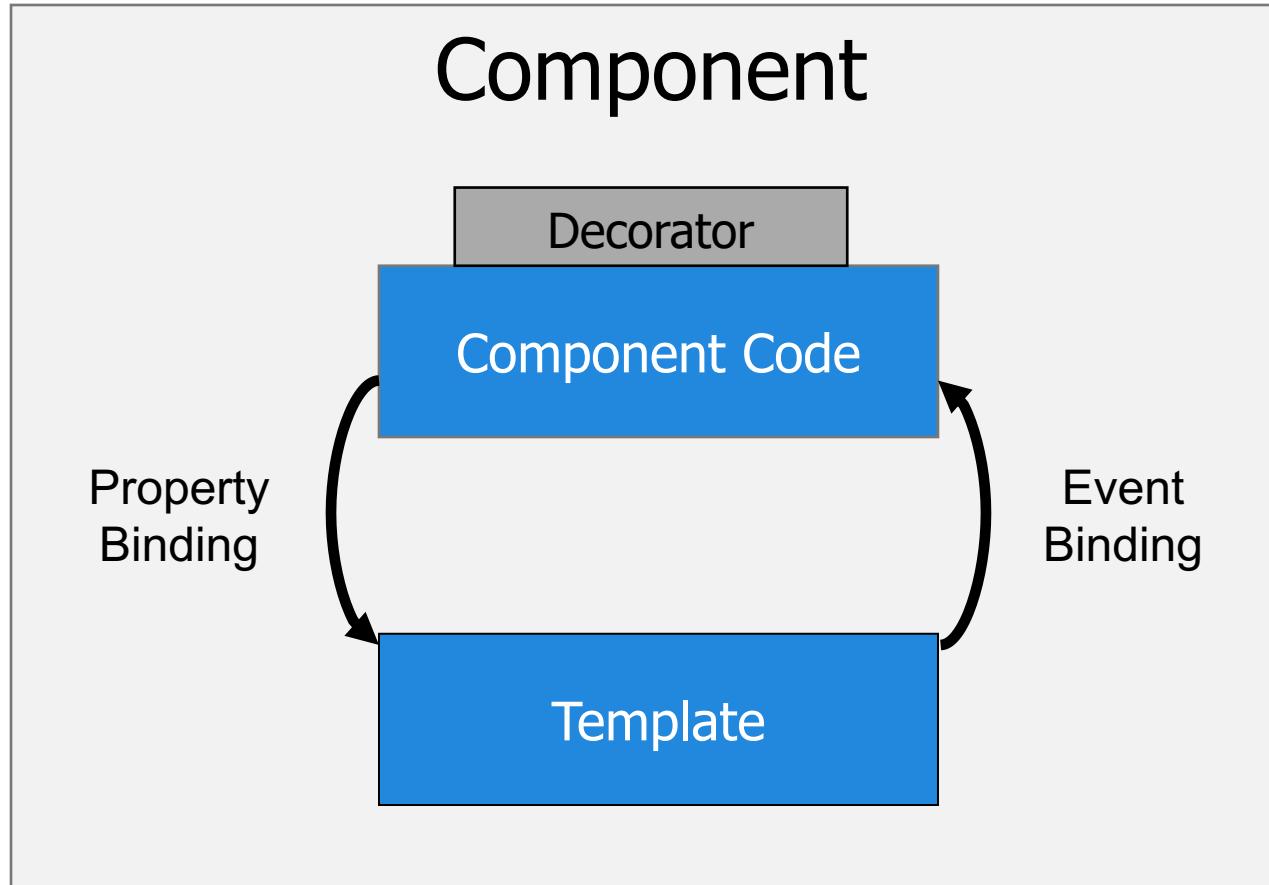
- ngClass can be used to add or remove one or more CSS classes:

```
<div [ngClass]="{active: growl.enabled, highlight: growl.focused}">  
  {{ customer.name }}  
</div>
```

Angular Directives in Action

Two-Way Binding

Two-Way Binding



Simulating Two-Way Data Binding

- Angular allows data to flow in using `@Input` properties
- Component properties are updated using events

```
<input [value]="filter" (input)="filter=$event.target.value" />
```

* The `ngModel` directive uses an `ngModel` input property (sets the `value`) and `ngModelChange` output property (listens for changes) to simulate Two-way Data Binding

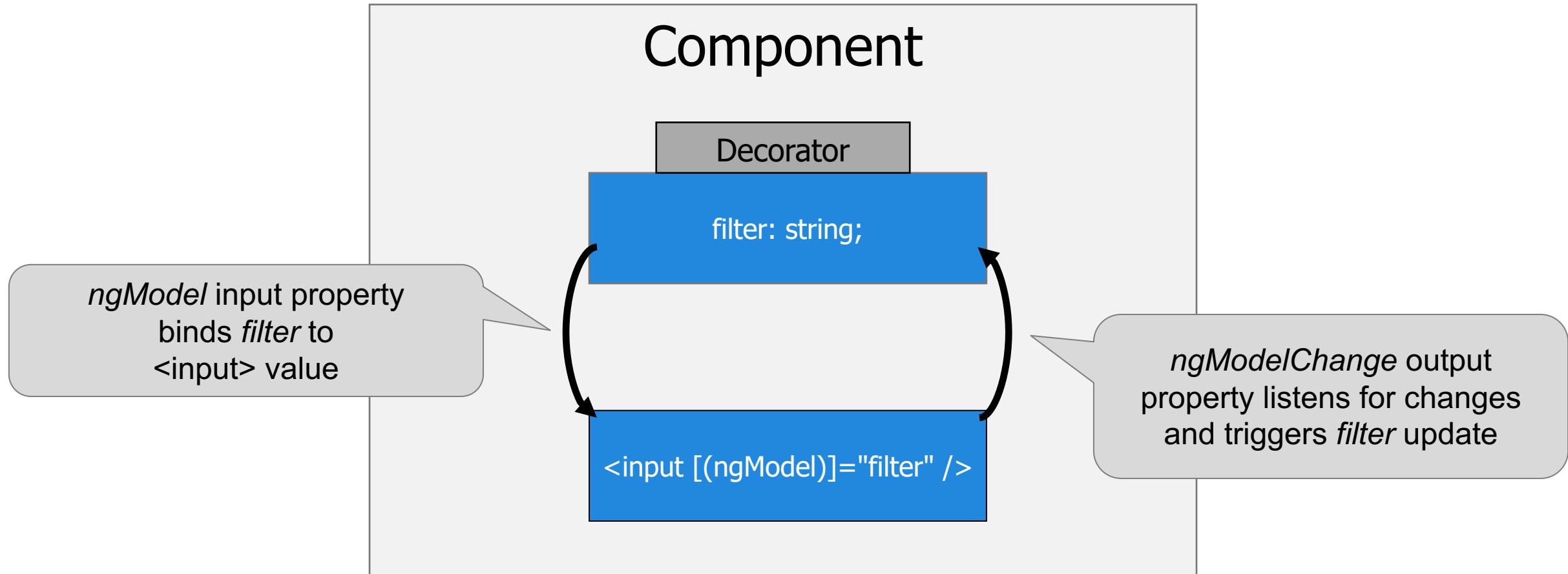
Binding with ngModel

- Create a "Two-way" binding using **[(ngModel)]** or **bindon-property** syntax
- Change triggers an event that updates the property value

```
<input type="text" [(ngModel)]="filter" />
```

```
<input type="text" bindon-ngModel="filter" />
```

How ngModel Works



Importing FormsModule

To use ngModel you must import the FormsModule

app.module.ts

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

@NgModule({
  imports: [ BrowserModule, FormsModule ],
  ...
})

export class AppModule { }
```

Creating a Custom "Two-Way" Property

A "two-way" property binding can be created by following a naming convention:

```
@Component({
  selector: 'textbox',
  template: `
    Message: <input type="text" [value]="textValue"
                  (input)="changeText($event.target.value)" />
  `,
})
export class TextboxComponent {
  @Input() textValue: string;
  @Output() textValueChange = new EventEmitter<string>();
  changeText(value: string) {
    this.textValue = value;
    this.textValueChange.emit(this.
  }
}
```

Using the component:

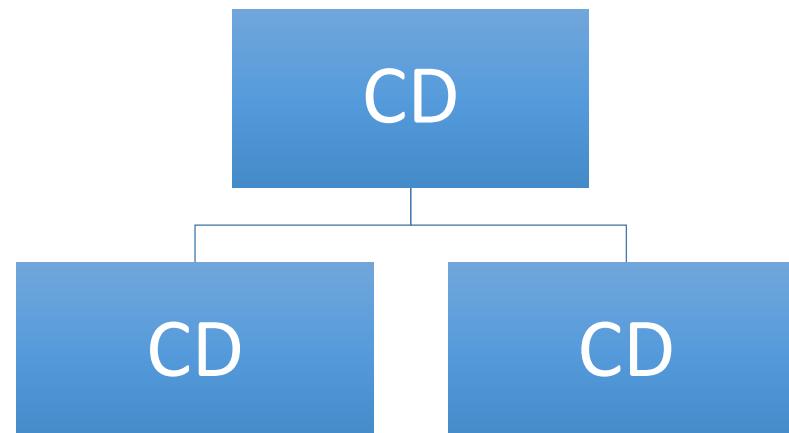
```
<textbox [(textValue)]="message"></textbox>
```

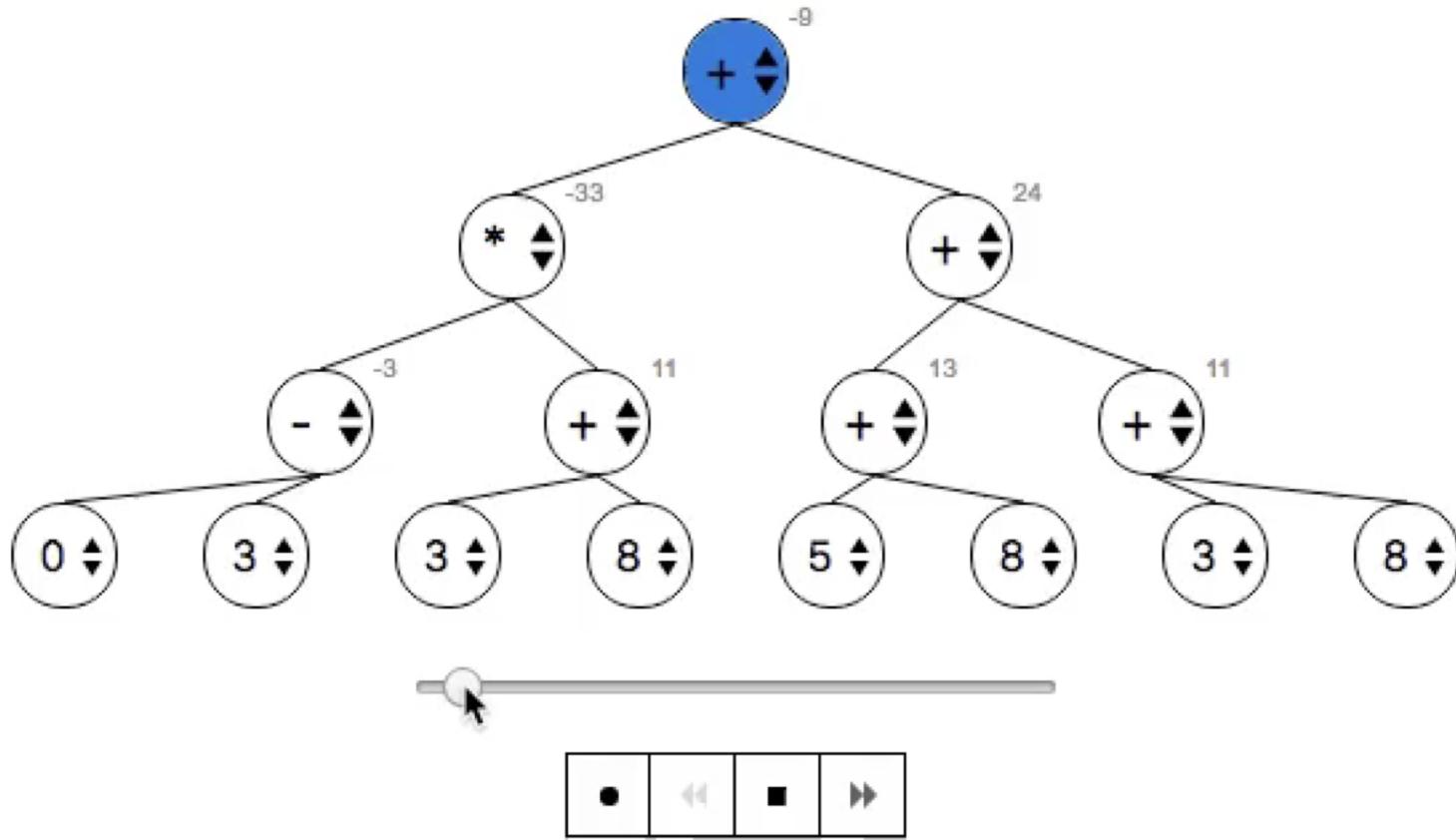
Two-Way Binding in Action

Change Detection

Change Detection

- An Angular application is a tree of components
- Each component has a change detector that is checked by default to ensure that model data is synced with the DOM
- Data flows from top to bottom (uni-directional data flow)





```
-> ngDoCheck called  
ngDoCheck called  
ngDoCheck called  
ngDoCheck called
```

Zone.js

- The "magic" behind Angular change detection
- Modifies low-level JavaScript APIs
- Monitors several key areas:
 - Events (click, submit, etc.)
 - Timers (setTimeout, setInterval)
 - Ajax/XHR
- Zones notify Angular when something changes which triggers the change detector tree

Changing the Change Detection Strategy

- Angular change detection monitors all component changes by default
- Strategy can be changed to only monitor input/output property changes

```
Import { Component, Input, ChangeDetectionStrategy } from '@angular/core';
@Component({
  ...
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class CustomerComponent {
  @Input() name: string;
}
```

Summary

- Components support input and output properties
- Angular provide several types of bindings:
 - [target]="expression"
 - (target)="statement"
 - [(ngModel)]="expression"
- Angular directives handle showing/hiding, looping through items and more



Lab

Working with Component Properties and Angular Directives





Angular Application Development



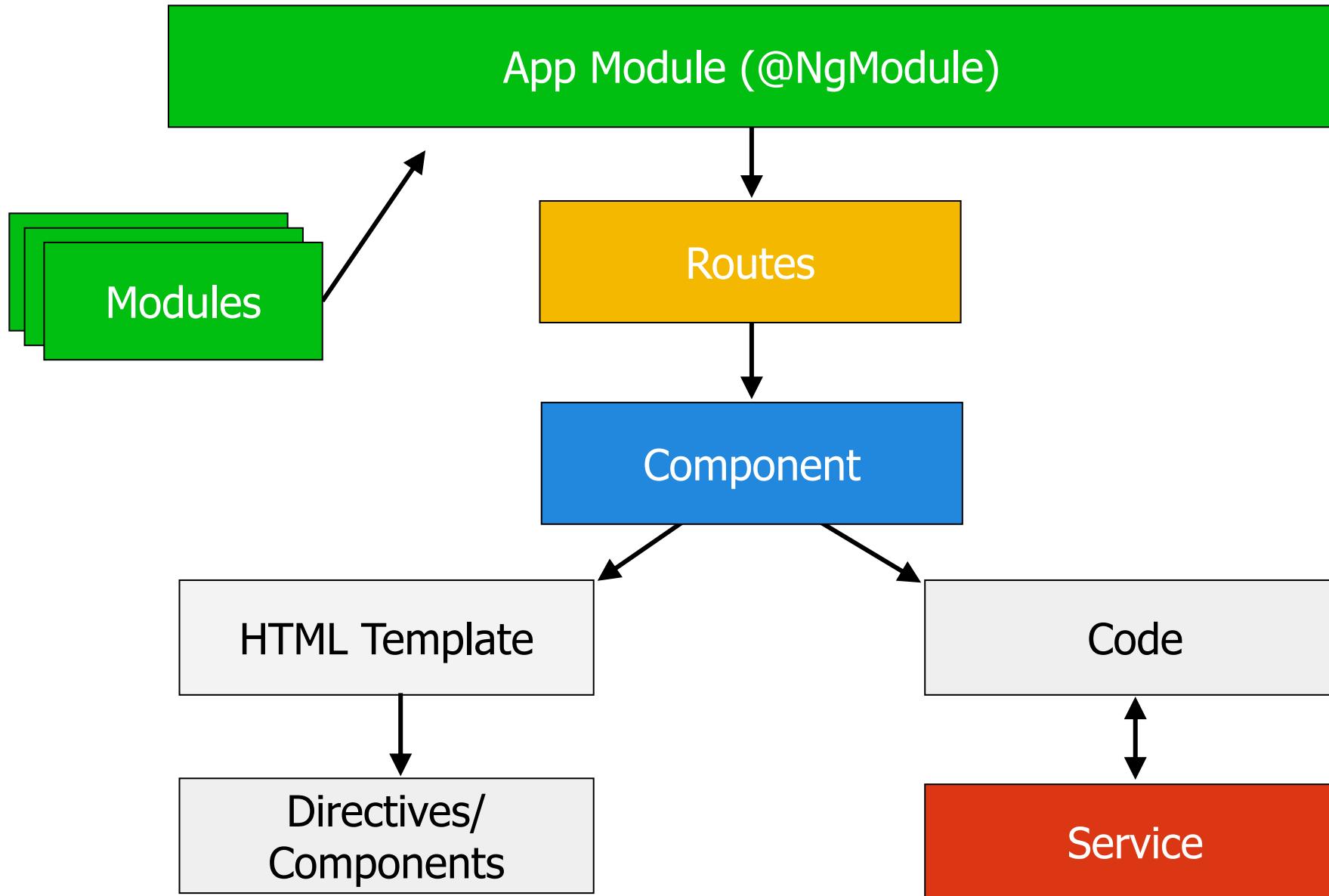
Services, Providers and HttpClient

Agenda

- Services Overview
- Injectors and Providers
- @Injectable Decorator
- Promises and Observables
- Calling RESTful Services with HttpClient
- Subscribing to an Observable



The Big Picture



Services Overview

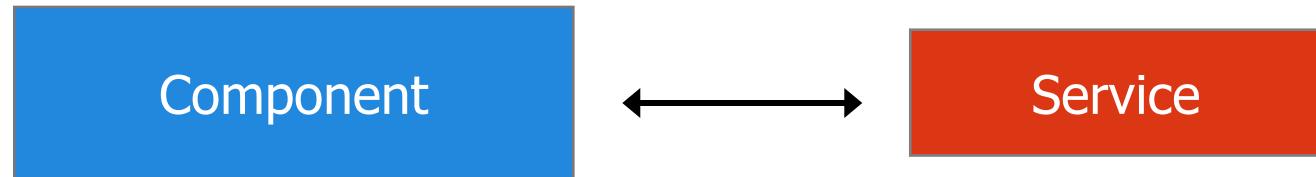
ng g service [service-name]

Services Overview

- Components rely on services to provide key functionality

- Service examples:

- Validation
- Logging
- Message bus
- Data retrieval
- Calculations



- Can be shared between a component and its child components

Steps to Create a Service



Service Class

- A service class provides values, functions and additional features to components and/or other services

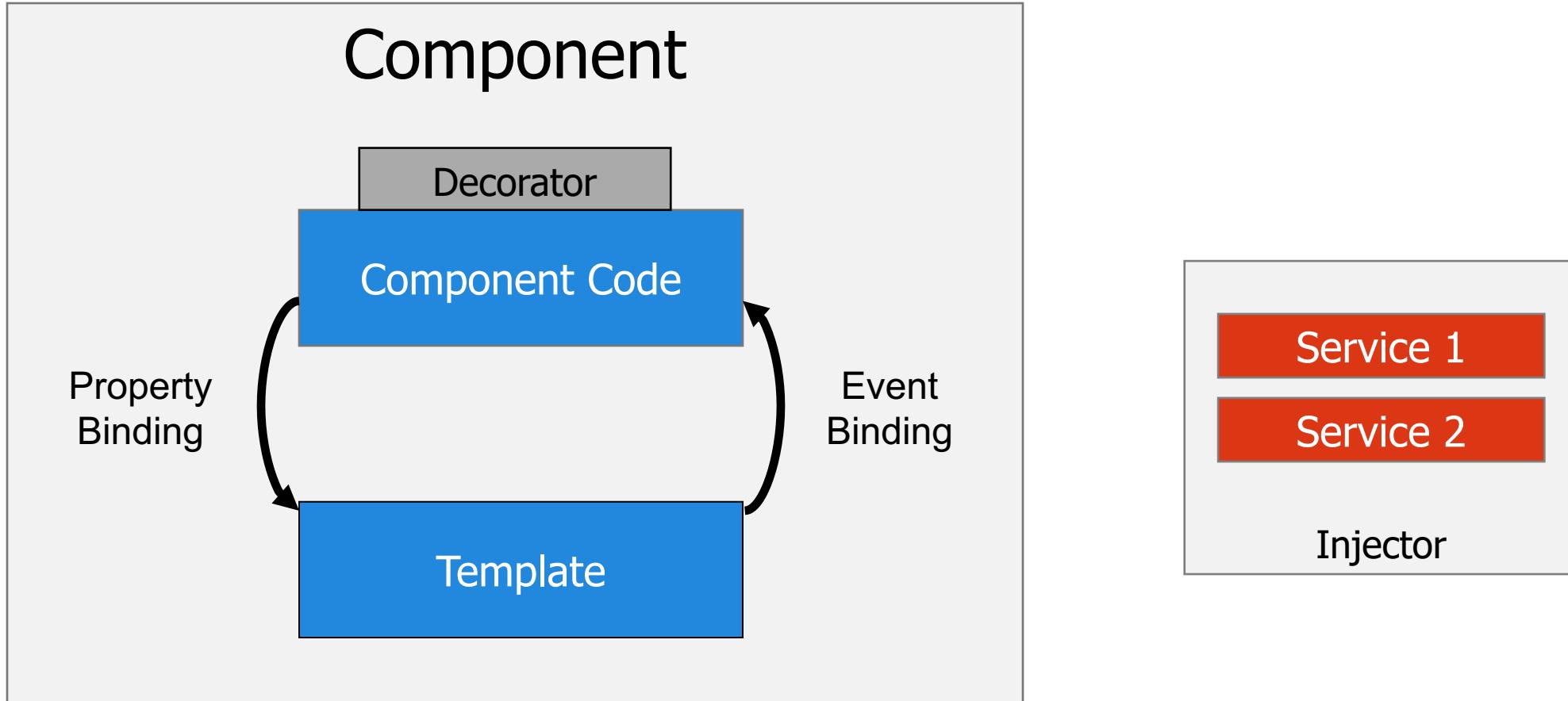
```
import { Dependency1 } from './something';

//Metadata (@decorator)
export class DataService {

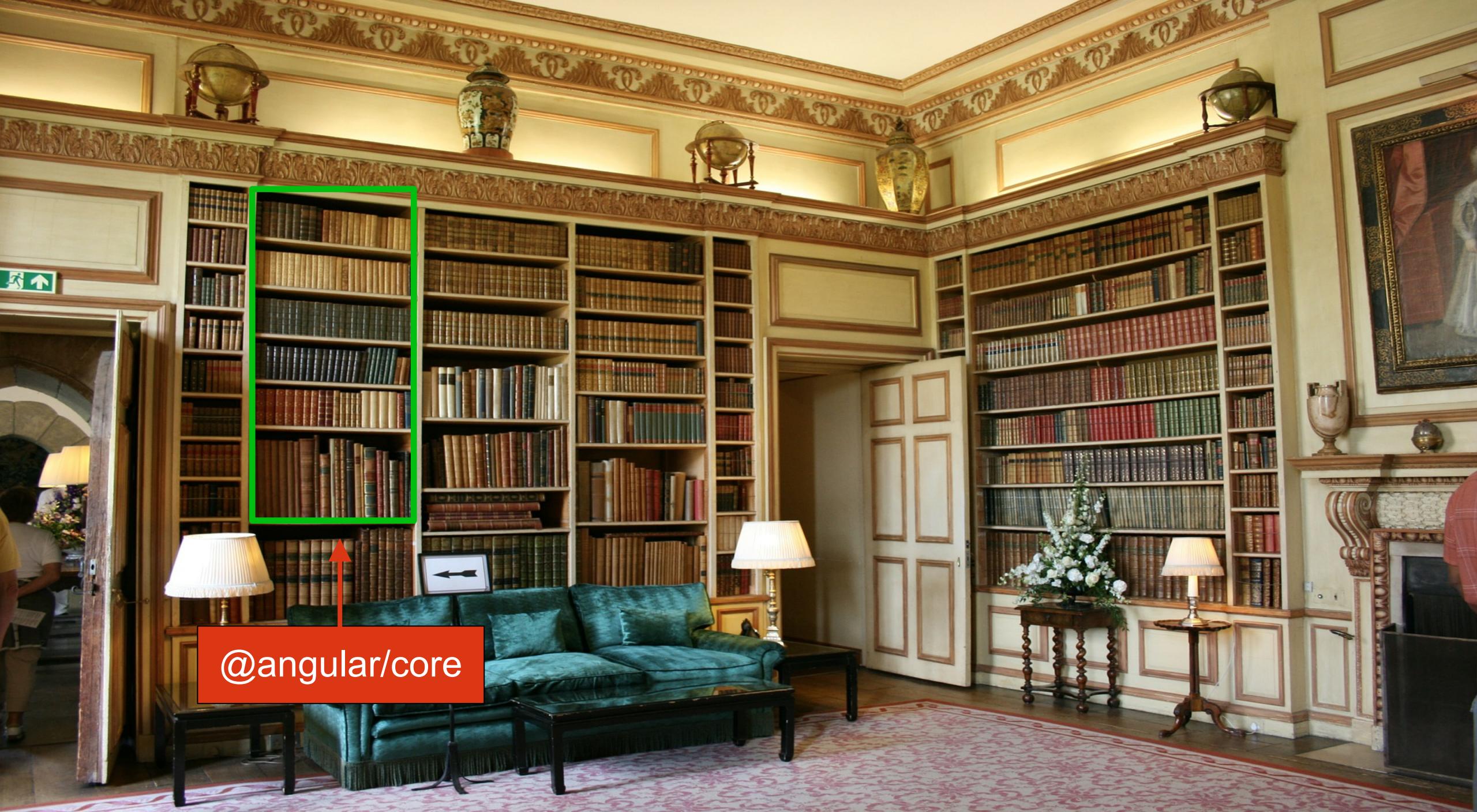
    constructor(private dep1: Dependency1) { }

    getProjectName(): string {
        return this.dep1.getProject().name;
    }
}
```

Injecting Services



Injectors and Providers



@angular/core

Dependency Injection

Provide an object with the dependencies it requires at runtime.

Dependency Injection and Injectors

- Angular uses Dependency Injection
- Services are "injected" into a component's constructor using an *Injector* at runtime rather than being "hard coded"
- Injector automatically resolves constructor dependencies

Component

```
constructor(private myService: MyService)
```

MyService

Injectors and Service Containers

- An *Injector* maintains a container of service instances (singletons)

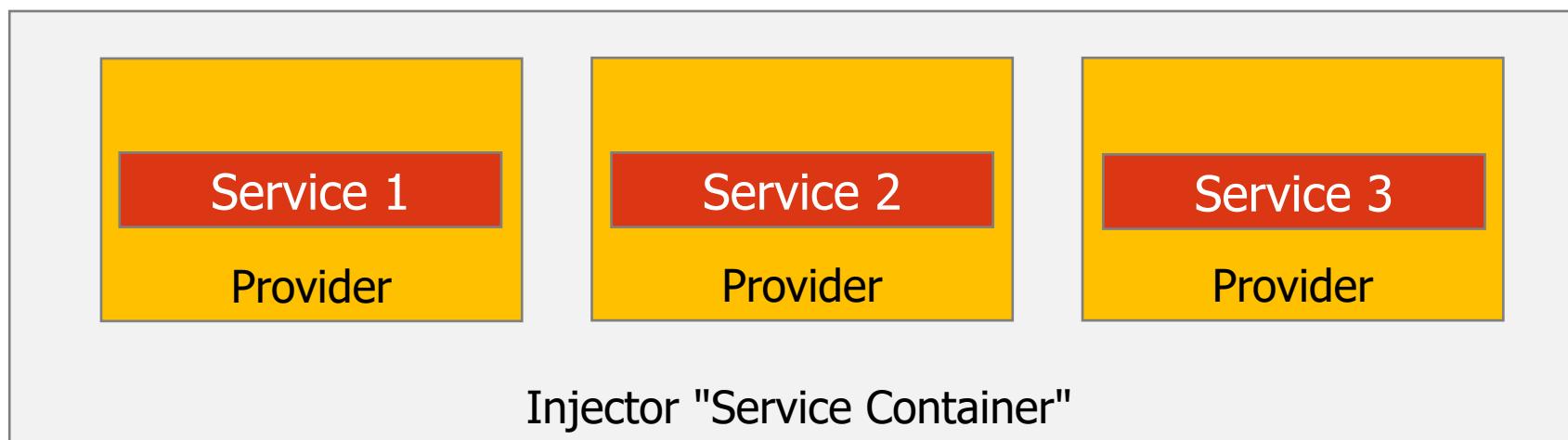


- Future requests for an existing service cause the Injector to return it from the container

<https://angular.io/guide/hierarchical-dependency-injection>

Injectors Rely on Providers

- An Injector relies on a **provider** to create or return a service
- A provider is a "recipe" for creating a service
- If a service doesn't have a provider defined then an error occurs during injection



Registering Providers in a Module

Service providers can be defined in an Angular module (or within the service itself with v6+)

app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/http';
import { DataService } from './shared/data.service';

@NgModule({
  imports: [ BrowserModule, HttpClientModule ],
  providers: [ DataService ]
})
export class AppModule { }
```

Providers Array: DataService
available everywhere

Registering Providers at the Component Level

- Providers can be registered in a component using the providers property
- Will override providers defined in a module and create a "child injector"

```
import { DataService } from './shared/services/data.service';

@Component({
  ...
  providers: [DataService]
})
export class AppComponent {
  constructor(private dataService: DataService) { }
}
```

Service available to component
and child components

Injected at runtime

The @Injectable Decorator

The @Injectable Decorator

- Services can have other services injected into them
- `@Injectable` decorator allows a service class to have dependencies "injected" into its constructor
- Inject MyService into DataService's constructor:



Using the @Injectable Decorator

- A service that has dependencies "injected" into its constructor must use the `@Injectable` decorator

```
import { Injectable } from '@angular/core';
import { OtherService } from './other.service';

@Injectable()
export class DataService {
  constructor(private otherService: OtherService) { }
}
```



Injected into DataService at
Runtime

Defining a Provider using `@Injectable`

- Angular 6+ allows a service's provider to be defined using the `@Injectable` decorator's `providedIn` property

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

@Injectable({
  providedIn: 'root'
})
export class DataService {
  constructor(private otherService: OtherService) { }
}
```

Service's provider will be part of
the root injector container

Promises and Observables

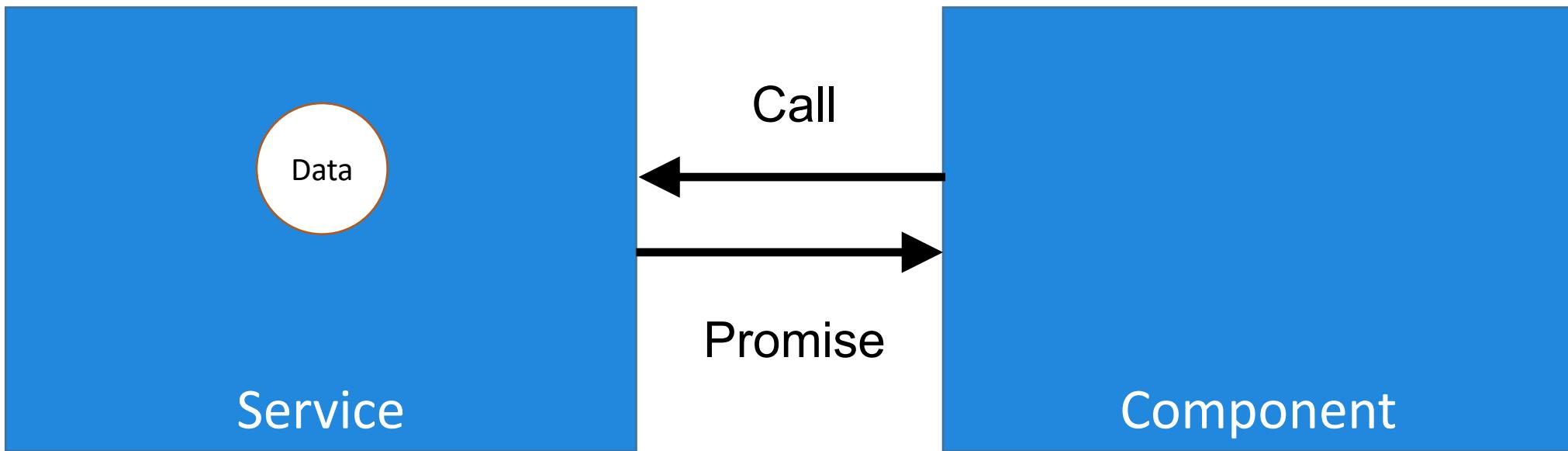
Promises and Observables

- Services that perform asynchronous operations can use Promises or Observables
- Promise:
 - An operation that hasn't completed yet, but is expected in the future
 - Used with async/deferred operations
 - Can be hooked to a callback
- Observable:
 - An object that can be “subscribed” to by other objects
 - Can return multiple values over time – an async data stream
 - Event based

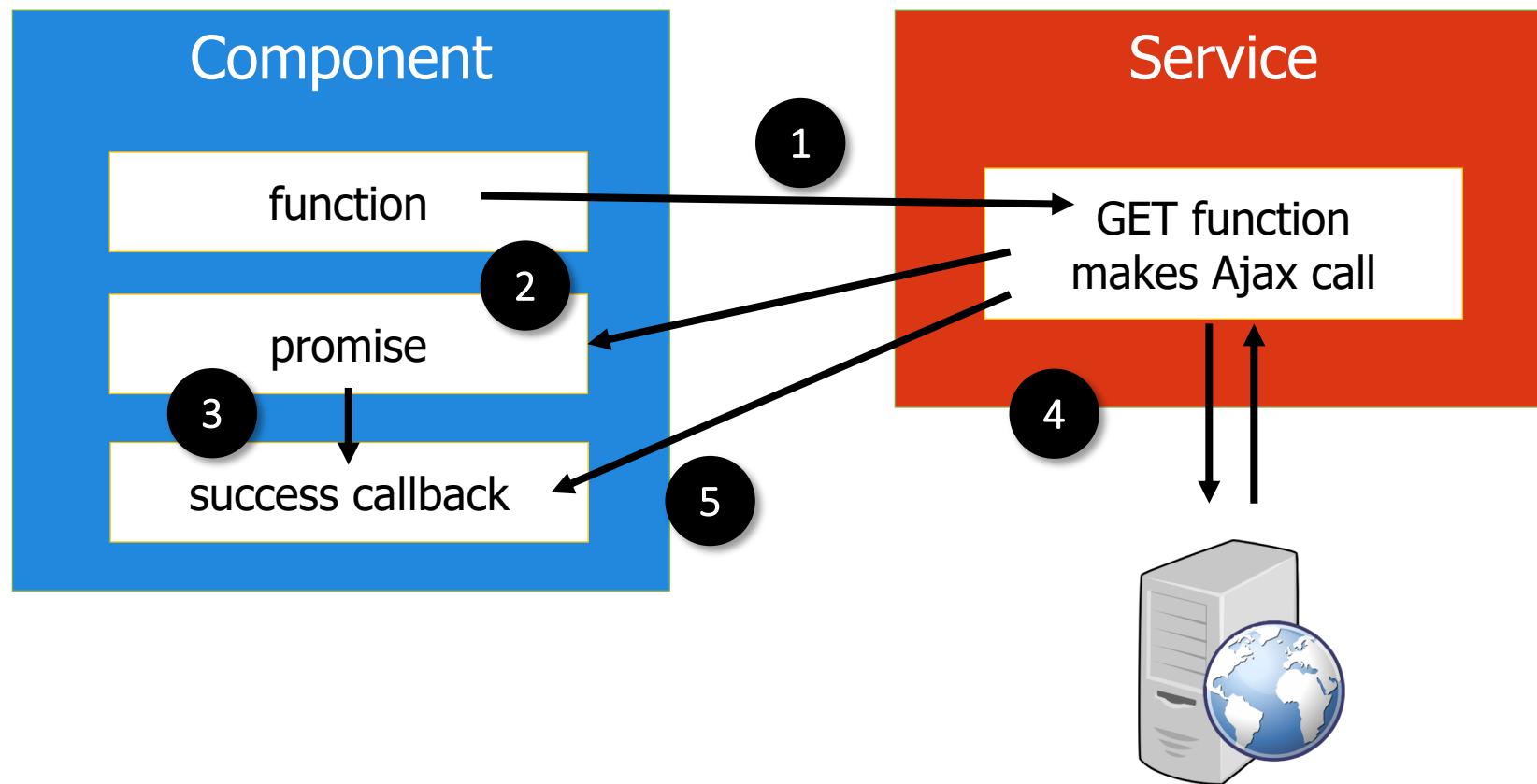
Observables versus Promises

Promise	Observable
Returns a single value	Can return multiple values over time (array)
Cannot cancel	Can cancel
	Supports standard array functions (map, filter, reduce, etc.)
	Can be converted to a promise

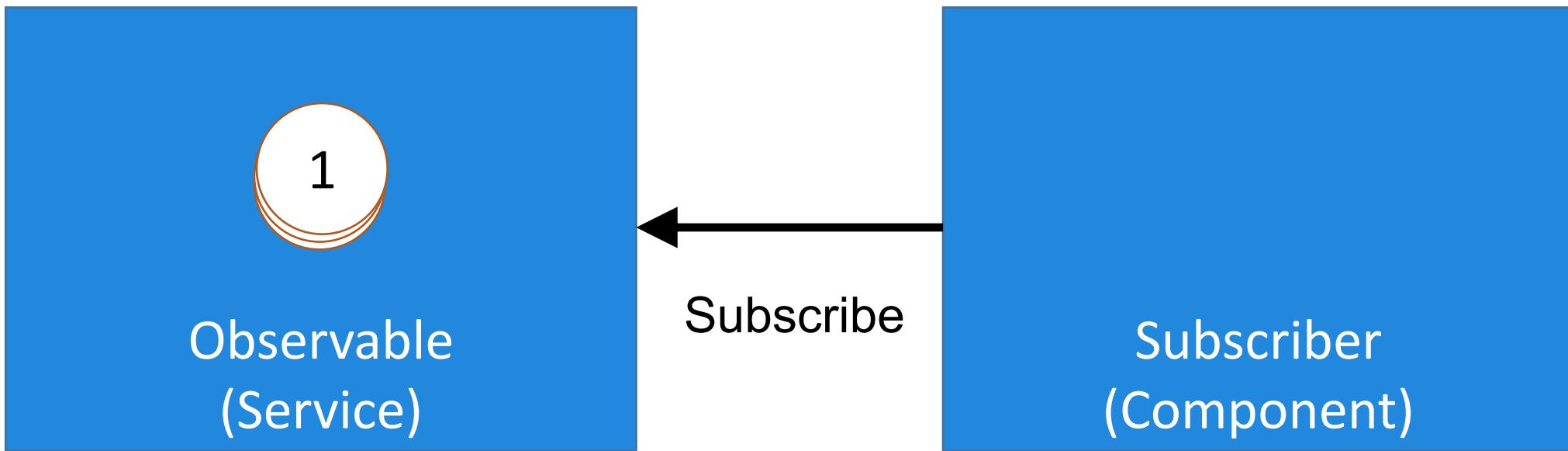
Promises Overview



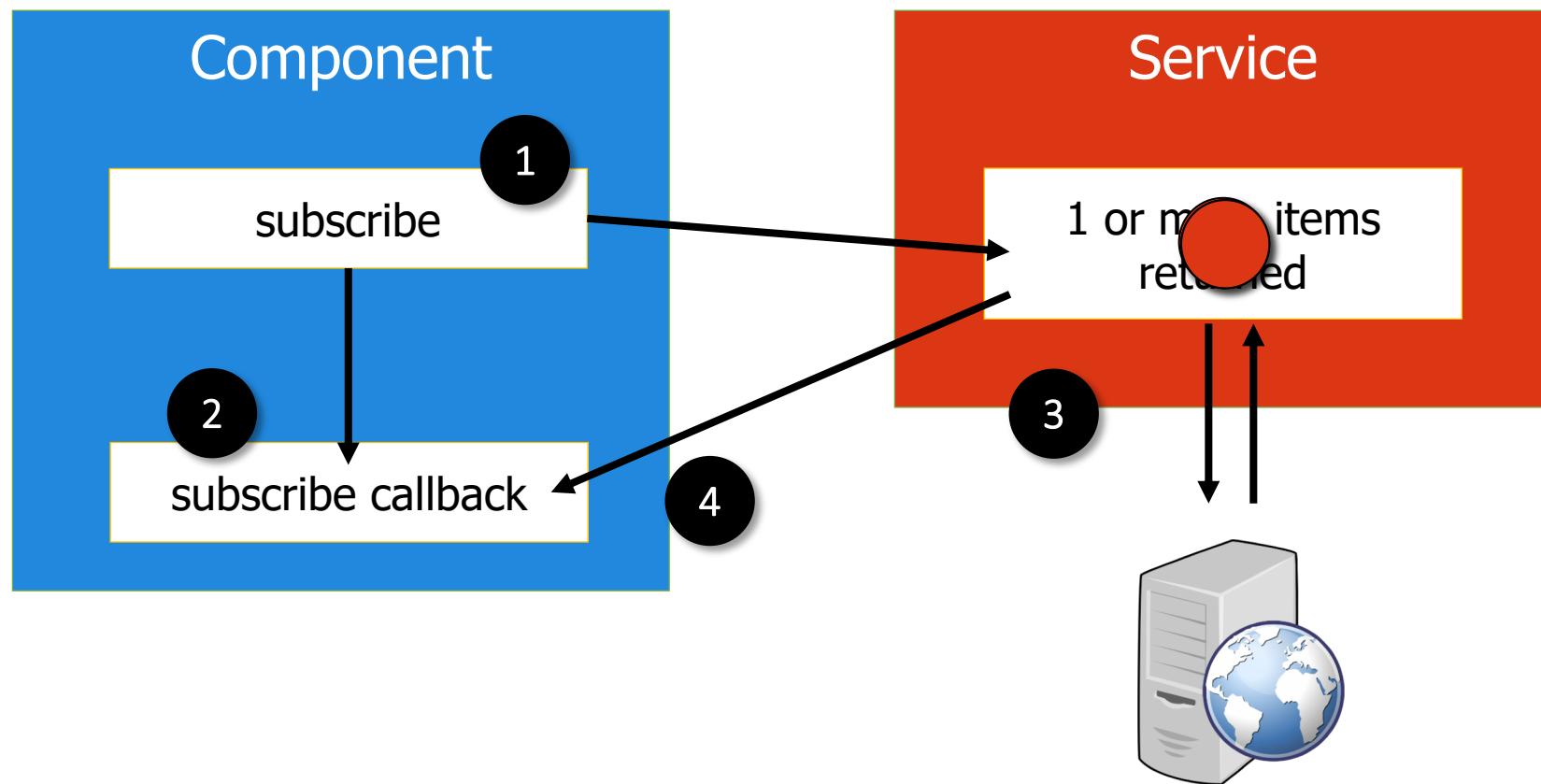
Promises in Action



Observables Overview



Observables in Action



Calling RESTful Services with HttpClient

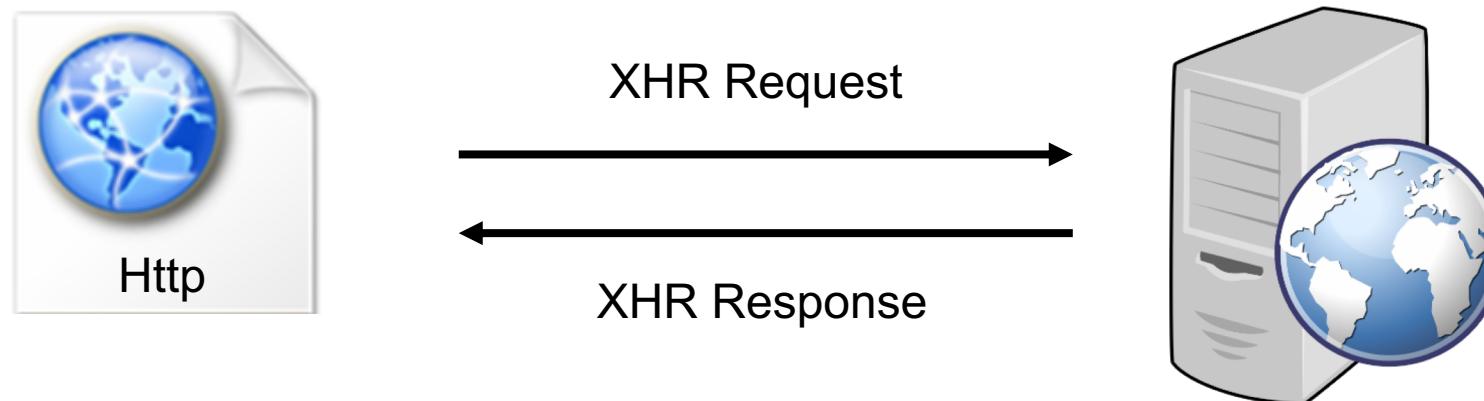


@angular/common/http

RxJS

RESTful Services and Http

- Components rely on services to communicate with the server
- `@angular/common/http` module has an *HttpClient* class that makes XMLHttpRequest (Ajax) calls to the server
- Async calls resolved using RxJS observables or promises



HttpClient Functions

- HttpClient can call a server using GET, POST, PUT, DELETE, PATCH and HEAD methods

Function	Description
get()	Performs a GET request
post()	Perform a POST request (insert)
put()	Perform a PUT request (update)
patch()	Perform a PATCH request (normally a partial update)
delete()	Perform a DELETE request
head()	Perform a HEAD request (retrieve metadata only)

Steps to use HttpClient

- 1 Import HttpClientModule into an Angular module
- 2 Import RxJS functionality
- 3 Import HttpClient from @angular/common/http
- 4 Inject HttpClient into a service

Importing HttpClientModule

To use HttpClient you must import the HttpClientModule. Angular 4.3+ is required.

app.module.ts

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

@NgModule({
  imports: [ BrowserModule, HttpClientModule ],
  ...
})
export class AppModule { }
```

Import HttpClientModule so its providers are available

Using get<T>()

- Use HttpClient.get<T>() to call the server and retrieve data

```
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { map, catchError } from 'rxjs/operators';
import { ICustomer } from '../shared/interfaces';
```

HttpClient and RxJS
functionality imported

```
@Injectable()
export class DataService {
  baseUrl: string = '/api/customers';
  customers: ICustomer[];
  constructor(private http: HttpClient) { }
  getCustomers(): Observable {
    return this.http.get<ICustomer[]>(this.baseUrl)
      .pipe(
        map((customers) => this.customers = customers),
        catchError(this.handleError)
      );
}
```

Http injected into DataService

Pipe the response into RxJS
operators

Using post<T>()

- Use HttpClient.post<T>() to send data to the server:

```
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { tap, map, catchError } from 'rxjs/operators';
import { ICustomerResponse, ICustomer } from '../shared/interfaces';

@Injectable()
export class DataService {
    constructor(private http: HttpClient) { }

    insertCustomer(customer: ICustomer) : Observable<ICustomer> {
        return this.http.post<ICustomerResponse>(this.baseUrl, customer)
            .pipe(
                tap(res => console.log('Post status ' + res.status)),
                map(res => res.customer),
                catchError(this.handleError)
            );
    }
}
```

Subscribing to an Observable

Subscribing to DataService Observables

Subscribing to an RxJS Observable returned from DataService

```
import { DataService } from './shared/services/data.service';
@Component({
  ...
})
export class CustomersComponent {
  customers: ICustomer[];
  filteredCustomers: ICustomer[];
  constructor(private dataService: DataService) { }
  ngOnInit() {
    this.dataService.getCustomers()
      .subscribe((customers: ICustomer[]) => {
        this.customers = this.filteredCustomers = customers;
      });
  }
}
```

Subscribe to Observable

Using the async Pipe

async pipe works with a Promise or Observable property

Allows data to be rendered asynchronously

```
import { DataService } from './shared/services/data.service';
@Component({
  ...
  template: `<div *ngFor="let cust of customers$ | async">{{cust.name}}</div>`
})
export class CustomersComponent {
  constructor(private DataService: DataService) { }
  ngOnInit() {
    this.customers$ = this.dataService.getCustomers();
  }
}
```

Binding will wait until
data is available

Returns an observable

Services in the Angular JumpStart App

Summary

- Dependency injection is used throughout Angular
- An injector maintains a container of services
- Injectors rely on *providers* ("recipes") to create and return services
 - `@Injectable` allows a service to be injected into another service
 - `Http` can be used to call RESTful services and return Observables



Lab

Working with Services, Providers and Http



Bonus: Using Http (Angular < 4.3)

Steps to use Http (if using Angular < 4.3)

- 1 Import HttpClientModule into an Angular module
- 2 Import RxJS functionality
- 3 Import Http from @angular/http
- 4 Inject Http into a service

Importing HttpModule

To use Http you must import the HttpModule

app.module.ts

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

@NgModule({
  imports: [ BrowserModule, HttpClientModule ],
  ...
})
export class AppModule { }
```

Import HttpClientModule so its providers are available

Http Class Functions

- Http class can call a server using GET, POST, PUT, DELETE, PATCH and HEAD methods

Function	Description
get()	Performs a GET request
post()	Perform a POST request (insert)
put()	Perform a PUT request (update)
patch()	Perform a PATCH request (normally a partial update)
delete()	Perform a DELETE request
head()	Perform a HEAD request (retrieve metadata only)

Using get()

- Use Http.get() to call the server and retrieve data

```
import { Http, Response } from '@angular/http';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/catch'

@Injectable()
export class DataService {
  baseUrl: string = '/api/customers';
  constructor(private http: Http) { }
  getCustomers(): Observable<ICustomer[]> {
    return this.http.get(this.baseUrl)
      .map((res: Response) => res.json())
      .catch(this.handleError);
  }
}
```

Http and RxJS functionality imported

Http injected into DataService

Using post()

- Use Http.post() to send data to the server:

```
@Injectable()
export class DataService {
  baseUrl: string = '/api/customers';
  constructor(private http: Http) { }

  insertCustomer(customer: ICustomer) : Observable<ICustomer> {
    return this.http.post(this.baseUrl, customer)
      .map((res: Response) => res.json())
      .catch(this.handleError);
  }
}
```



Angular Application Development



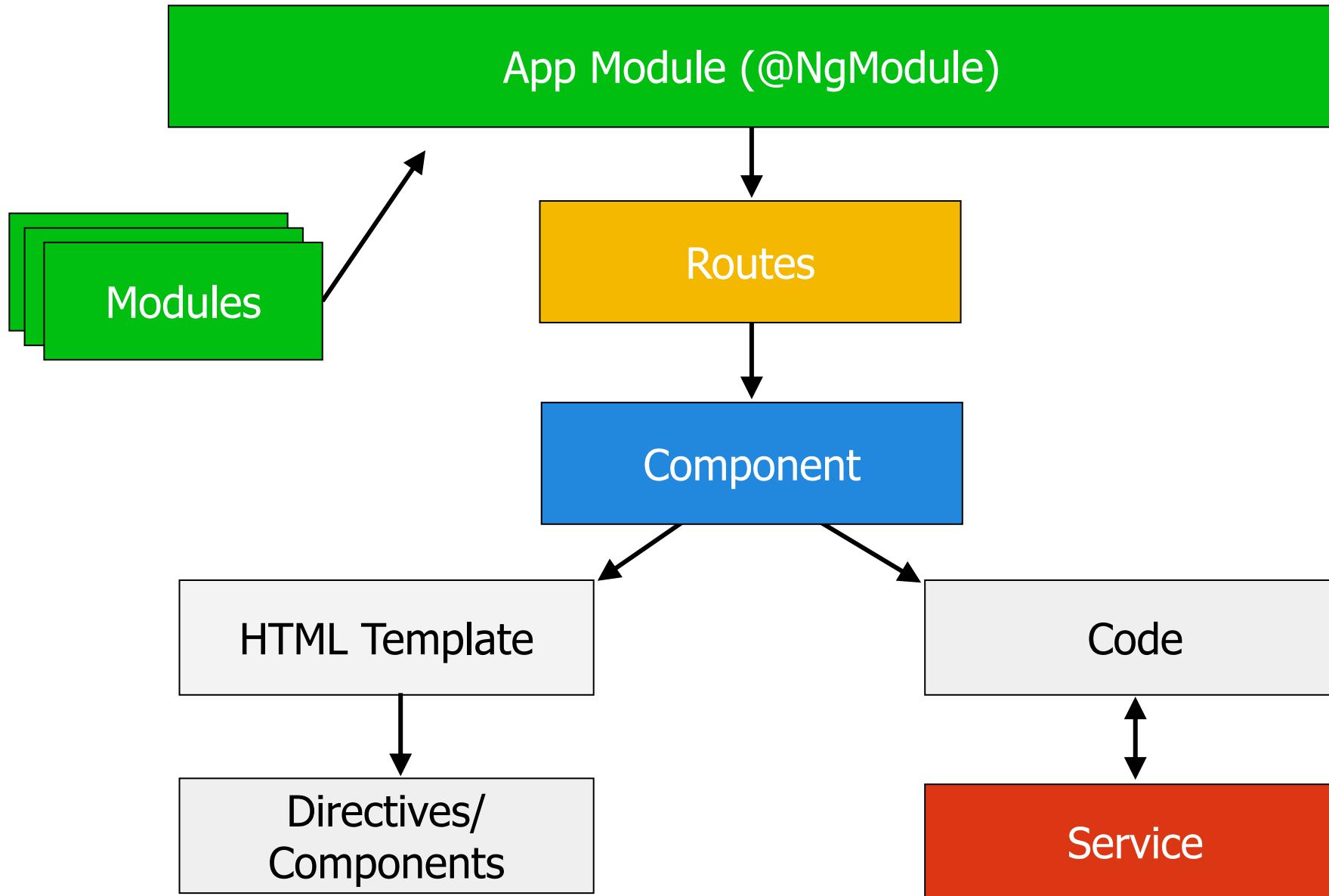
Routing

Agenda

- Routing Overview
- Routing Steps
- The Router Service
- Route Parameters
- Child Routes



The Big Picture

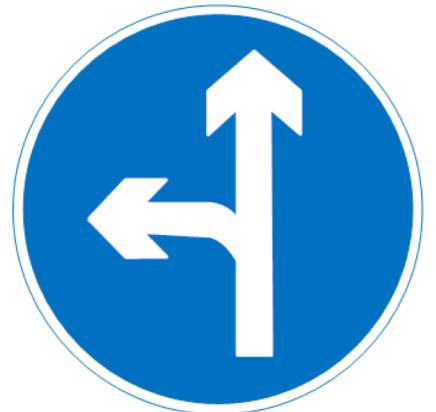


Routing Overview

```
ng new my-app --routing
```

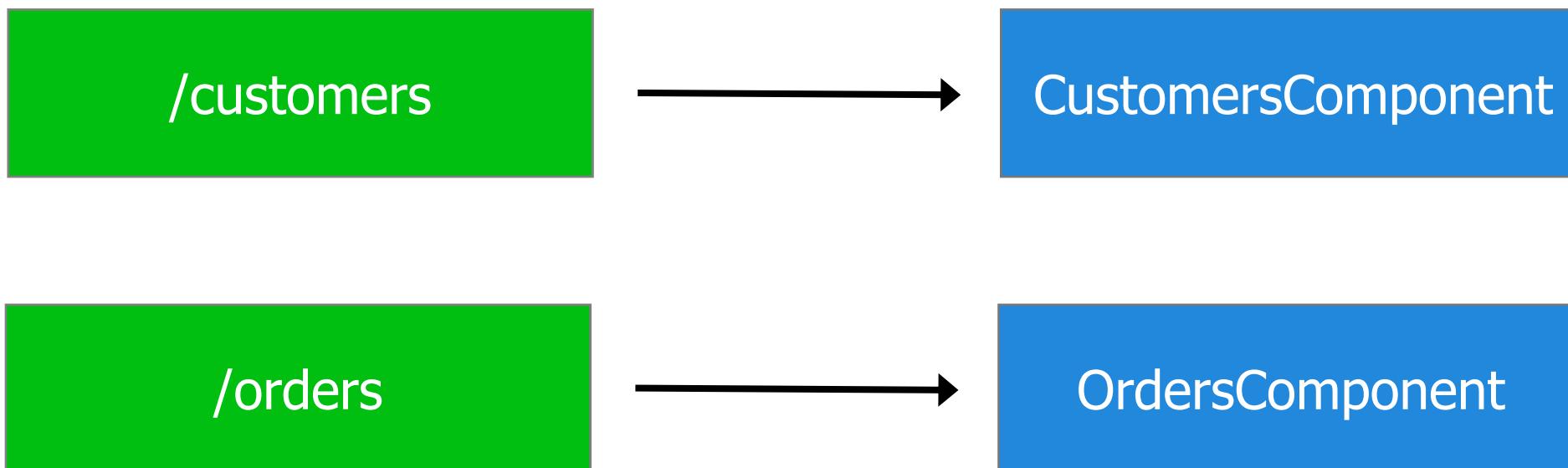
Routing Overview

- Routes are virtual paths that can be used to load components into a screen
- Use the **@angular/router** module
- **RouterModule** and the **Router** service play key roles

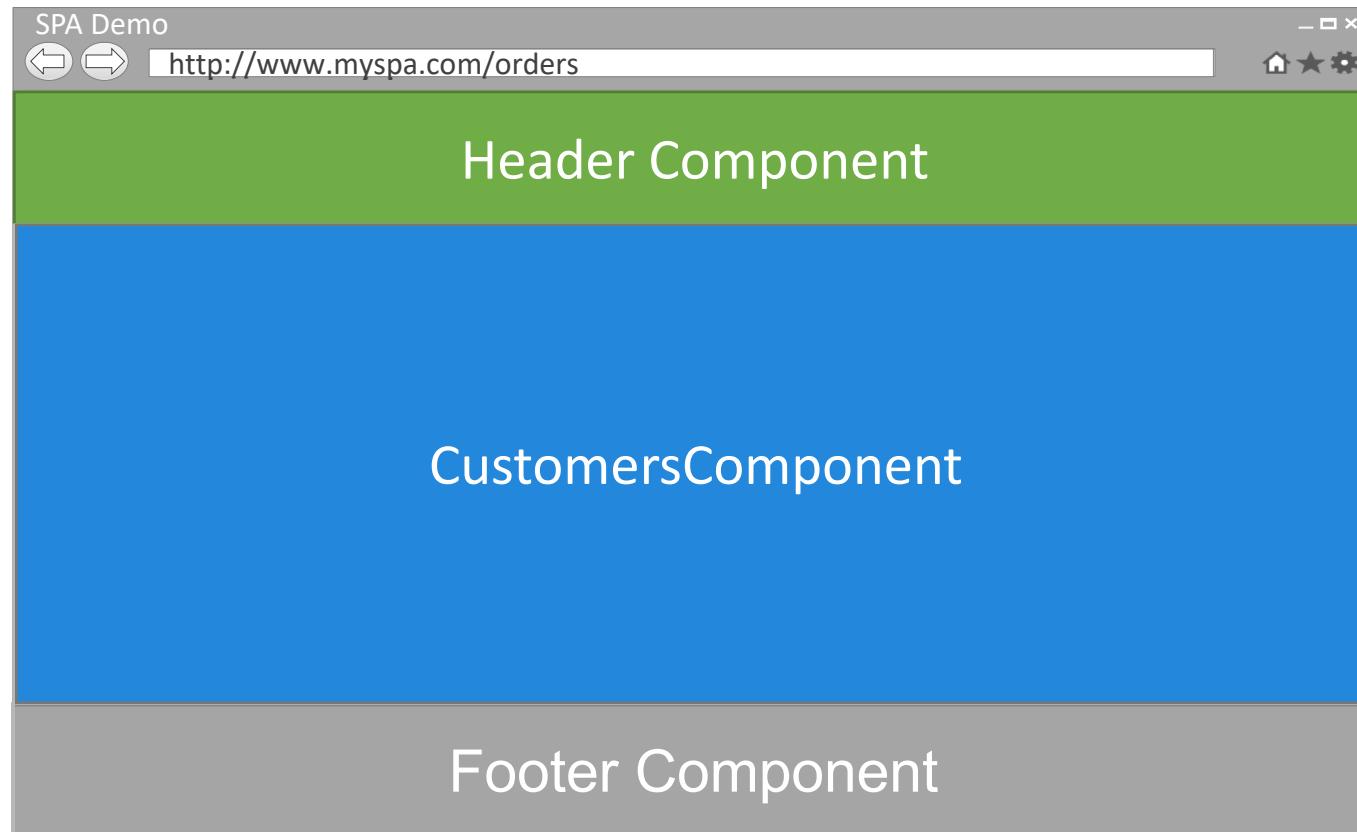


Routes and Components

Routes determine which component to load:



Changing Component Views



Routing Steps

Steps to Use Routing

- 1 Add a <base> Element
- 2 Define routes
- 3 Pass routes to the AppModule
- 4 Add a <router-outlet>
- 5 Use the routerLink directive

Add a <base> Element

- Router supports history.pushState
- Allows paths like http://yourdomain.com/customers to be used
- The <base> element needs to be set for it to work properly

index.html

```
<html>
<head>
  <base href="/">
</head>
<body>

</body>
</html>
```

Steps to Use Routing

- 1 Add a <base> Element
- 2 **Define routes**
- 3 Pass routes to the AppModule
- 4 Add a <router-outlet>
- 5 Use the routerLink directive

Import Routes and RouterModule

- RouterModule provides access to routing features
- Routes help us declare or route definitions

app.routing.ts

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

Define Routes

- Define the route's path
- Set the target component for the route

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

const routes: Routes = [
  { path: '', pathMatch: 'full', redirectTo: '/customers', },
  { path: 'customers', component: CustomersComponent },
  { path: 'orders/:id', component: CustomerOrdersComponent },
  { path: '**', component: NotFoundComponent},
];
```

Creating a Custom Routing Module

Routes can be created inside of a custom routing module that is imported into a feature or root module

app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
...

const routes: Routes = [
  { path: '', pathMatch: 'full', redirectTo: '/customers', },
  { path: 'customers', component: CustomersComponent },
  { path: 'orders/:id', component: CustomerOrdersComponent },
  { path: '**', component: NotFoundComponent }
];
@NgModule({
  imports: [ RouterModule.forRoot(routes) ],
  exports: [ RouterModule ]
})
export class AppRoutingModule {}
```

Add routes into module

Steps to Use Routing

- 1 Add a <base> Element
- 2 Define routes
- 3 **Pass routes to the AppModule**
- 4 Add a <router-outlet>
- 5 Use the routerLink directive

Import Routes into AppModule

Import routes into AppModule

app.module.ts

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

import { AppComponent } from './app.component';
import { AppRoutingModule } from './app-routing.module';

@NgModule({
  imports: [ BrowserModule, AppRoutingModule ],
  declarations: [ AppComponent, ...other components go here... ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Steps to Use Routing

- 1 Add a <base> Element
- 2 Define routes
- 3 Pass routes to the AppModule
- 4 **Add a <router-outlet>**
- 5 Use the routerLink directive

Add a <router-outlet>

Components are loaded into the <**router-outlet**> area of a template

app.component.ts

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

@Component({
  selector: 'app-container',
  template: `<router-outlet></router-outlet>`
})
export class AppComponent {
  constructor() { }
}
```

Steps to Use Routing

- 1 Add a <base> Element
- 2 Define routes
- 3 Pass routes to the AppModule
- 4 Add a <router-outlet>
- 5 **Use the routerLink directive**

Using the routerLink Directive

- The routerLink directive can be used to add links to routes
- Defines the route path and any route parameter data

customers.component.ts

```
@Component({
  selector: 'customers',
  templateUrl: 'customers.component.html'
})
export class CustomersComponent { }
```

customers.component.html

```
<a routerLink="/customers">
  Customers
</a>

<a [routerLink]=["/orders", customer.id]>
  {{ customer.firstName }} {{ customer.lastName }}
</a>
```

The Router Service

The Router Service

- The Router service handles mapping URLs to components
- Holds a reference to the <router-outlet>
- Provides functions such as navigate() to go to different routes
- Supports child routes, route parameters and more



Navigating with the Router Service

Routes can be navigated to in code using Router's **navigate()** function:

```
import { Router } from '@angular/router';
...

@Component({
  selector: 'customer-edit',
  templateUrl: 'customer-edit.component.html'
})
export class CustomerEditComponent implements OnInit {
  constructor(private router: Router, private dataService: DataService) { }
  deleteCustomer() {
    this.dataService.deleteCustomer(this.customer.id)
      .subscribe((status: boolean) => {
        this.router.navigate(['/customers']);
      });
  }
}
```

Navigate to "customers"
root route

Route Parameters

Accessing Route Parameters

- Route parameters can be accessed using the ActivatedRoute object
- Can return a route parameter value using an **observable** or by grabbing a **snapshot** of the static value

Route Parameter

`http://localhost:3000/customers/6`

Defining Route Parameters

Route parameters are defined on a route using the : character

```
const appRoutes: Routes = [  
  { path: 'orders/:id', component: CustomerOrdersComponent}  
];
```

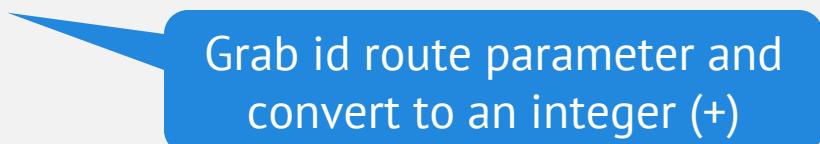
Example: /orders/2

```
<a [routerLink]="['/orders', customer.id]">  
  {{ customer.firstName }}  
</a>
```

Accessing a Route Parameter Snapshot

customer-orders.component.ts

```
@Component({  
  ...  
})  
export class CustomerOrdersComponent implements OnInit {  
  
  constructor(private route: ActivatedRoute, private dataService: DataService) { }  
  
  ngOnInit() {  
    const id = +this.route.snapshot.paramMap.get('id');  
    //Call dataService  
  }  
}
```



Grab id route parameter and convert to an integer (+)

<http://localhost:3000/customers/6>

Observable Route Parameters

customer-edit.component.ts

```
@Component({  
  ...  
})  
export class CustomerOrdersComponent implements OnInit {  
  constructor(private route: ActivatedRoute,  
             private dataService: DataService) { }  
  
  ngOnInit() {  
    this.route.paramMap.subscribe(params => {  
      const id = +params.get('id');  
      //call dataService  
    });  
  }  
}
```

ActivatedRoute **params** property
is an observable

Grab id route parameter and
convert to an integer (+)

<http://localhost:4200/customers/6>

Routing in Action

Child Routes

Child Routes in Action

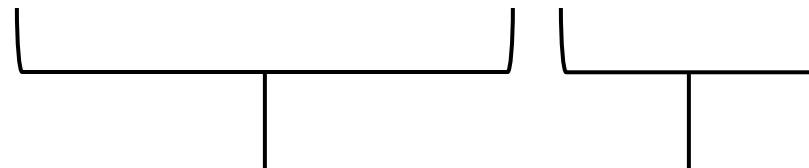
The screenshot shows a web browser window titled "Angular 2 TypeScript App" at "localhost:3000/customers". The title bar includes standard OS X icons and a user name "Dan". The main content area has a blue header bar with a "Customer Manager" icon and the text "Customer Manager". Below this is a sub-header "Customers" with a person icon. There are two navigation options: "Card View" (selected) and "List View". To the right is a "Filter:" input field with an empty placeholder. The main area displays a grid of 12 customer cards, arranged in three rows of four. Each card contains a customer's name, profile picture, address, and a "View Orders" link.

Name	Address	Action
Ted James	Phoenix, Arizona	View Orders
Michelle Thompson	Los Angeles, California	View Orders
Zed Bishop	Las Vegas, Nevada	View Orders
Tina Adams	Seattle, Washington	View Orders
Igor Minar	Chandler, Arizona	View Orders
Brad Green	Mountain View, California	View Orders
Misko Hevery	Mountain View, California	View Orders
Heedy Wahlin	Chandler, Arizona	View Orders
John Papa	Orlando, Florida	View Orders
Tonya Smith	Atlanta, Georgia	View Orders
Ward Bell	San Francisco, California	View Orders

Created by Dan Wahlin

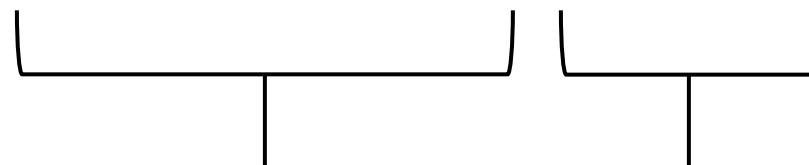
Child Route Structure

`http://localhost/customers/1/orders`



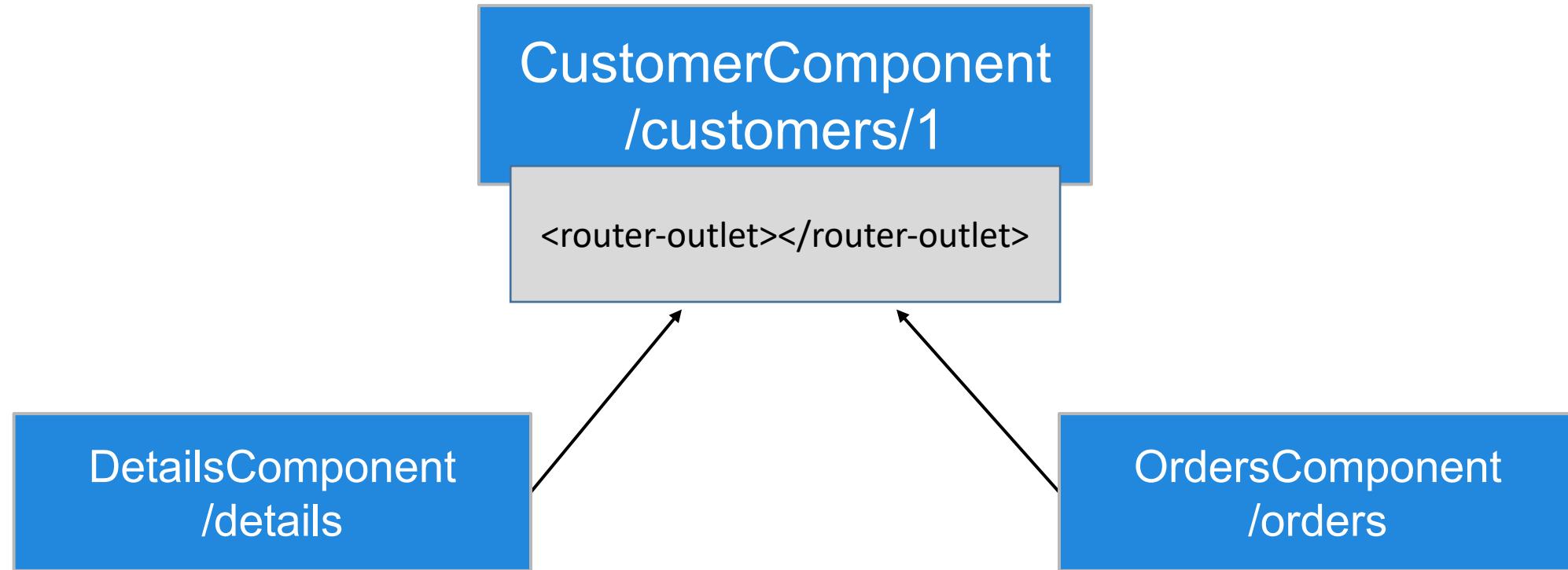
Parent Route Child Route

`http://localhost/customers/1/details`



Parent Route Child Route

Parent and Child Routes



<http://localhost/customers/1/details>

<http://localhost/customers/1/orders>

Parent and Child Routes

Child routes can be defined using the `children` property:

customer.routes.ts

```
import { Routes } from '@angular/router';
import { CustomerComponent } from './customer.component';
import { CustomerOrdersComponent } from './customerOrders.component';
import { CustomerDetailsComponent } from './customerDetails.component';
import { CustomerEditComponent } from './customerEdit.component';

const appRoutes: Routes = [
  {
    path: 'customers/:id', component: CustomerComponent,
    children: [
      { path: 'orders', component: CustomerOrdersComponent },
      { path: 'details', component: CustomerDetailsComponent },
      { path: 'edit', component: CustomerEditComponent }
    ]
  }
];
```

Parent Route

Child Routes

Linking to a Child Route

RouterLink can be used to link to a child route

```
customer.component.html
```

```
<a routerLink="details"  
     routerLinkActive="active">  
  Customer Details  
</a>
```

Route is relative to the
parent route

Add "active" CSS class when
"details" route is active

Accessing Parent Route Parameters

The router's **parent** property can be used to access a parameter value defined in a parent route

details.component.ts

```
@Component({
  selector: 'details-component'
})
export class DetailsComponent implements OnInit {

  constructor(private route: ActivatedRoute) { }

  ngOnInit() {
    const id = +this.route.parent.snapshot.paramMap.get('id');
  }
}
```

Access parent route param using
snapshot (or via observable)

<http://localhost/customer/6/details>

Child Routes in Action

Summary

- Routing is provided by the `@angular/router` module
- Routes are defined and passed to `RouterModule.forRoot()` or `RouterModule.forChild()`
- Components defined in a route are loaded into a view's `<router-outlet>` area
- Child routes provide a way to define nested routes



Lab

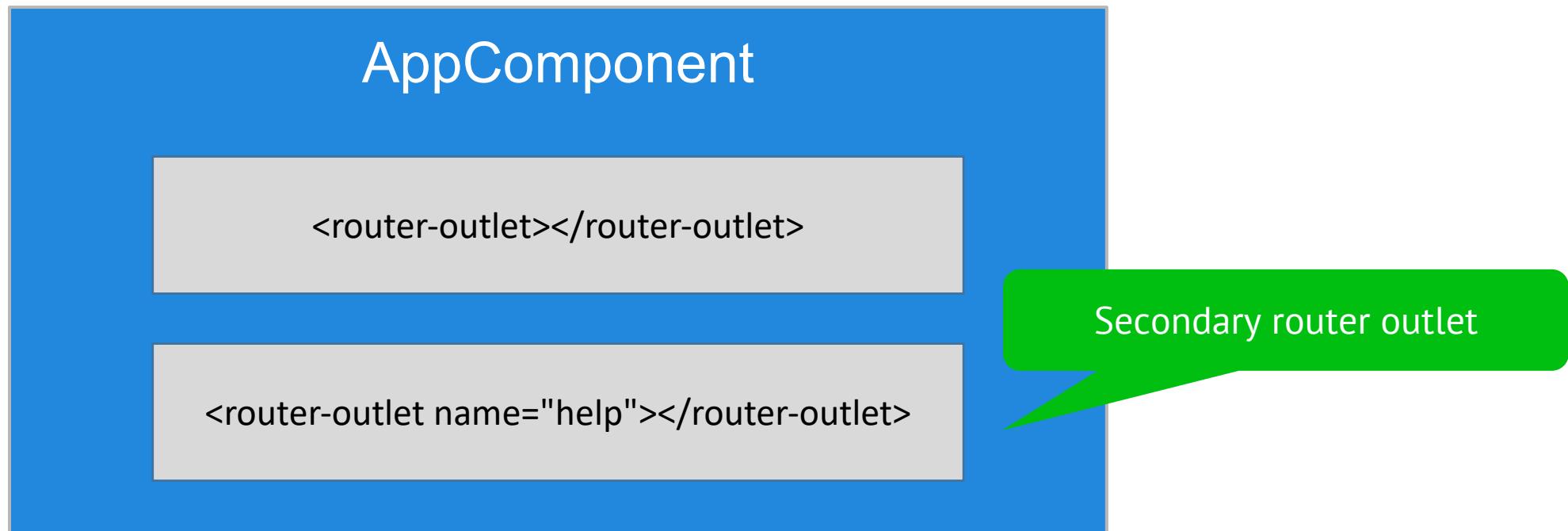
Working with Routing



Bonus - Secondary Routes

Secondary Routes

- Secondary routes are created by giving a name to a router outlet
- Allows the router to load multiple components at the same time



Loading a Secondary Route

Secondary routes are defined using an "outlet" property

app-routing.module.ts

```
...
{ path: 'customers', component: CustomersComponent },
{ path: 'helptext', component: HelpComponent, outlet: 'help' },
...
```

Name of router outlet where component will be loaded

header.component.html

```
<a [routerLink]="[{ outlets: { help: ['helptext'] } }]>View Customers Help</a>
```

Loading a Primary and Secondary Route

Primary and secondary routes can be loaded simultaneously

app-routing.module.ts

```
...
{ path: 'customers', component: CustomersComponent },
{ path: 'helptext', component: HelpComponent, outlet: 'help' },
...
```

header.component.html

```
<a [routerLink]="[{ outlets: { primary: ['customers'], help: ['helptext'] }}]">Customers + Help</a>
```

Load primary route +
secondary route

Clearing a Secondary Route

Clear a secondary route by passing null:

header.component.ts

```
this.router.navigate([{ outlets: { help: null }}]);
```



Angular Application Development



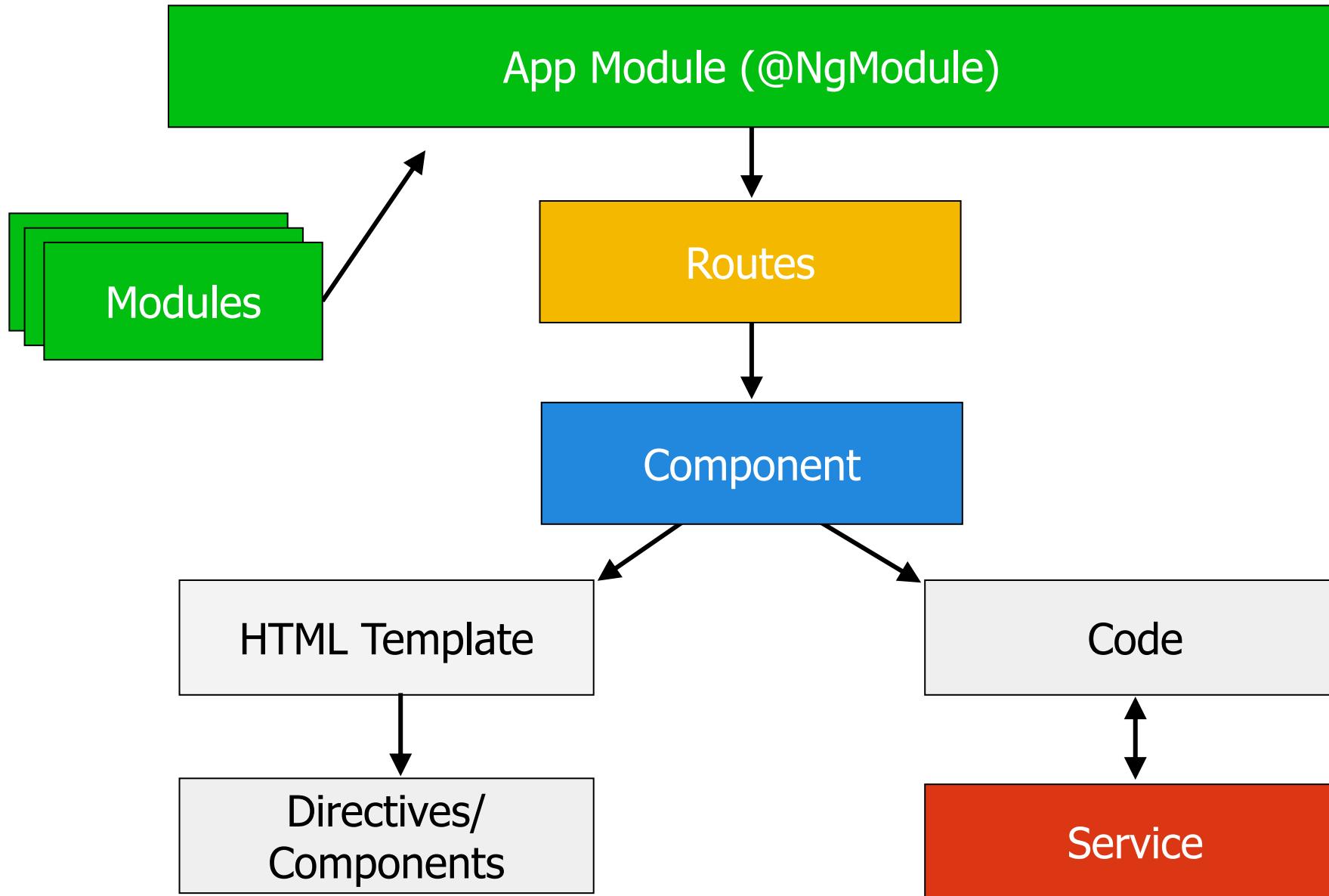
Route Guards and Lazy Loading

Agenda

- Route Guards
- Creating and Using Guards
- Lazy Loading



The Big Picture



Route Guards

Introduction to Route Guards

- Route Guards are used to intercept navigation to or from a route
- Key route guard scenarios:
 - User isn't allowed to navigate to a root or child route (not authorized)
 - Prompt user before leaving a route due to unsaved changes
 - Ensure data is valid before leaving a route in a multi-part form
 - Determine if a feature module can be loaded asynchronously

Built-In Route Guards

CanActivate

CanActivate
Child

CanDeactivate

CanLoad

Route Guard Details

CanActivate – Determines if a route can be "activated" and navigated to by the user

CanActivateChild – Determines if a child route can be "activated" and navigated to by the user

CanDeactivate – Determines if a user can navigate away from a route

CanLoad – Determines if a feature module can be loaded asynchronously

Resolve – Not technically a guard, but can be used to preload data before navigating to a route and showing the component

Creating and Using Guards

CanActivate Route Guard

CanActivate can be used to check if a user can navigate to a route

can-activate.guard.ts

```
import { Injectable } from '@angular/core';
import { CanActivate, Router, ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router';
import { AuthService } from '../core/services/auth.service';

@Injectable()
export class CanActivateGuard implements CanActivate {
  constructor(private authService: AuthService, private router: Router) { }

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {
    if (this.authService.isAuthenticated) { return true; }

    this.authService.redirectUrl = state.url;
    this.router.navigate(['/login']);
    return false;
  }
}
```

Determines if user can navigate to the route where the guard is applied

Applying CanActivate to a Route

Adding CanActivateGuard to a route using the **canActivate** property:

customer-routing.module.ts

```
const routes: Routes = [
  {
    path: '',
    component: CustomerComponent,
    children: [
      { path: 'edit',
        component: CustomerEditComponent,
        canActivate: [ CanActivateGuard ]
      }
    ]
  }
];
@NgModule({
  imports: [ RouterModule.forChild(routes) ],
  exports: [ RouterModule ],
  providers: [ CanActivateGuard ]
})
export class CustomerRoutingModule { }
```

Associate guard with target route

Ensure guard has a provider so it can
be used at runtime.

CanDeactivate Route Guard

CanDeactivate can be used to check if a user can leave a route/component

can-deactivate.guard.ts

```
import { Injectable } from '@angular/core';
import { CanDeactivate, ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router';
import { Observable } from 'rxjs/Observable';
import { CustomerEditComponent } from './customer-edit.component';

@Injectable()
export class CanDeactivateGuard implements CanDeactivate<CustomerEditComponent> {

  canDeactivate(component: CustomerEditComponent, route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {
    console.log(`CustomerId: ${route.parent.params['id']} URL: ${state.url}`);
    //Check with component to see if we're able to deactivate
    return component.canDeactivate();
  }
}
```

Check with component to see if the user
should be prompted before leaving.

Component's canDeactivate() Function

The component's canDeactivate() function determines if a user will be prompted before navigating to a different route

customer-edit.component.ts

```
canDeactivate(): Promise<boolean> | boolean {
  if (!this.customerForm.dirty) { return true; }

  //Dirty show display modal dialog to user to confirm leaving
  const modalContent: IModalContent = {
    header: 'Lose Unsaved Changes?',
    body: 'You have unsaved changes! Would you like to leave the page and lose them?',
    cancelButtonText: 'Cancel',
    OKButtonText: 'Leave'
  }
  return this.modalService.show(modalContent);
}
```

If no unsaved changes then allow user to navigate to a different route

If there are unsaved changes then prompt the user (returns a promise)

Applying CanDeactivate to a Route

Adding CanDeactivateGuard to a route using the **canDeactivate** property:

customer-routing.module.ts

```
const routes: Routes = [
  {
    path: '',
    component: CustomerComponent,
    children: [
      { path: 'edit',
        component: CustomerEditComponent,
        canActivate: [ CanActivateGuard ],
        canDeactivate: [ CanDeactivateGuard ]
      }
    ]
  }
];
@NgModule({
  imports: [ RouterModule.forChild(routes) ],
  exports: [ RouterModule ],
  providers: [ CanActivateGuard, CanDeactivateGuard ]
})
export class CustomerRoutingModule { }
```

Associate guard with target route

Ensure guard has a provider so it can be used at runtime.

Route Guards in Action

Lazy Loading

Route Loading

Angular supports two ways to load routes:

Eager Loaded Routes

Lazy Loaded Routes

Lazy Loaded Routes

- Lazy loading can be used to load Angular features asynchronously:
 - Load as the feature is requested (when the user clicks a link for instance)
 - Load in the background as the application first loads
- Use lazy loading anytime you'd like load a feature "on demand" or preload it in the background

Lazy Loading and the loadChildren Property

A lazy loaded route can be created using the loadChildren property

app-routing.module.ts

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

const app_routes: Routes = [
  { path: '', pathMatch:'full', redirectTo: '/customers' },
  { path: 'customers/:id', loadChildren: 'app/customer/customer.module#CustomerModule' },
  { path: '**', pathMatch:'full', redirectTo: '/customers' }
];

@NgModule({
  imports: [ RouterModule.forRoot(app_routes) ],
  exports: [ RouterModule ]
})
export class AppRoutingModule { }
```

Load a customer feature on demand as
a user navigates to target route

Defining a Lazy Loading Strategy

PreloadAllModules strategy can asynchronously "preload" feature modules in the background as an application loads

app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes, PreloadAllModules } from '@angular/router';
import { PreloadModulesStrategy } from './coreestrategies/preload-modules.strategy';

const app_routes: Routes = [
  { path: '', pathMatch:'full', redirectTo: '/customers' },
  { path: 'customers/:id', loadChildren: 'app/customer/customer.module#CustomerModule' },
  { path: '**', pathMatch:'full', redirectTo: '/customers' }
];
@NgModule({
  imports: [ RouterModule.forRoot(app_routes, { preloadingStrategy: PreloadAllModules }) ],
  exports: [ RouterModule ]
})
export class AppRoutingModule { }
```

Lazy loaded modules will be
asynchronously loaded in the background

Creating a Custom Loading Strategy

Custom preloading strategies can be created to load specific modules in the background

preload-modules.strategy.ts

```
import { Injectable } from '@angular/core';
import { PreloadingStrategy, Route } from '@angular/router';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/of';

@Injectable()
export class PreloadModulesStrategy implements PreloadingStrategy {
  preload(route: Route, load: () => Observable<any>): Observable<any> {
    if (route.data && route.data['preload']) {
      console.log('Preloaded: ' + route.path);
      return load();
    } else {
      return Observable.of(null);
    }
  }
}
```

```
const routes: Routes = [
  {
    path: 'customers/:id',
    loadChildren: 'customer.module#CustomerModule',
    data: { preload: true }
  }
];
```

Lazy Loading in Action

Summary

- Route guards can be applied to routes:
 - CanActivate
 - CanActivateChild
 - CanDeactivate
 - CanLoad
- Feature modules can be loaded asynchronously in the background using lazy loading



Lab

Route Guards and Lazy Loading





Angular Application Development



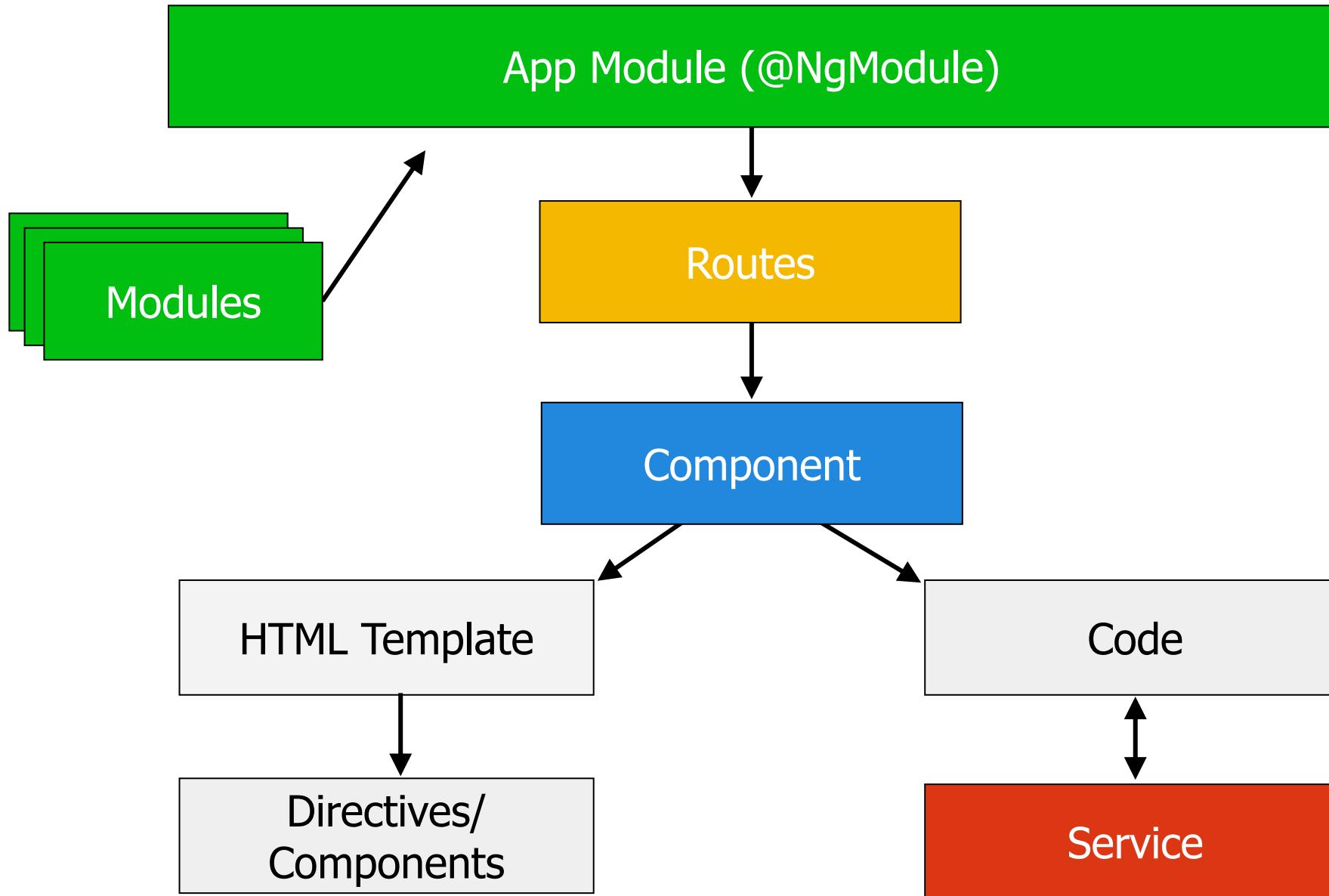
Template and Reactive Forms

Agenda

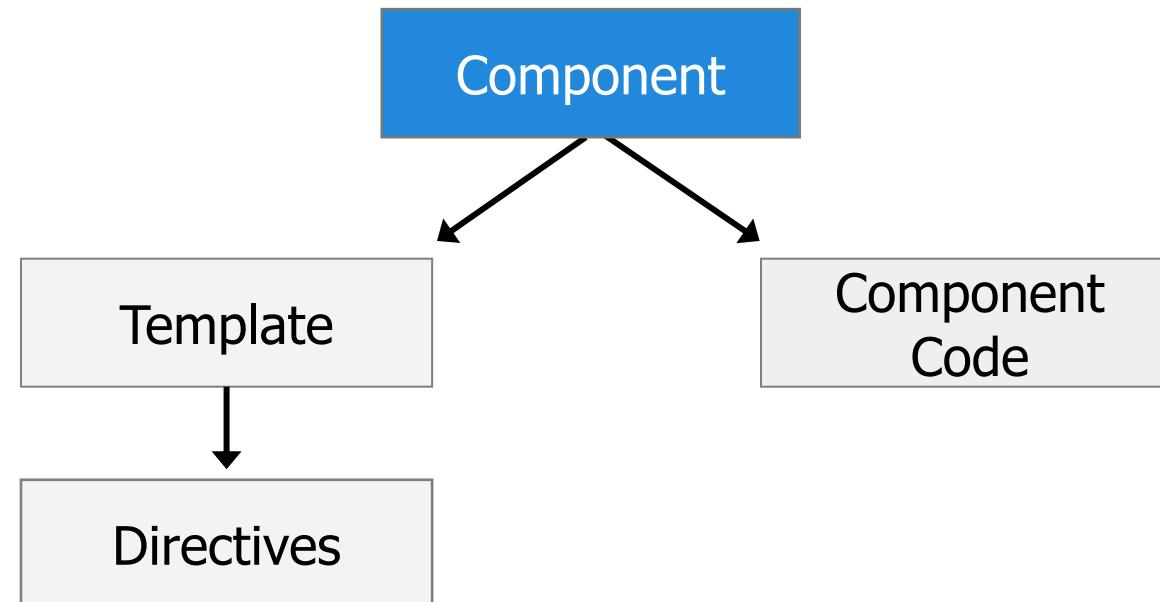
- Forms Overview
- Template-Driven Forms
- Reactive Forms
- Managing Form Control Styles



The Big Picture



Components and Forms



Forms Overview

Forms Overview

- Angular forms features:
 - Data binding
 - Validation
 - Change tracking
 - Error handling
- Form options:
 - Template-driven forms (declarative)
 - Reactive Forms (imperative)

Forms in Action

Edit Customer

Name

Last Name

Address

City

State

[Cancel](#)

[Update](#)

Key Forms Players

Template-Driven Forms

FormsModule

NgForm

NgModel

NgSubmit

Reactive Forms

ReactiveForms
Module

FormGroup

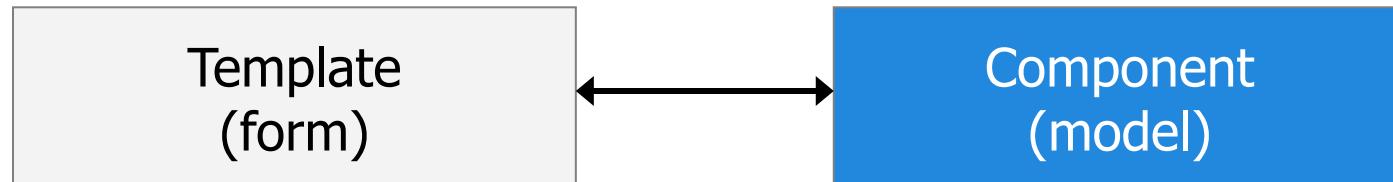
FormControlName

FormBuilder

Template-Driven Forms

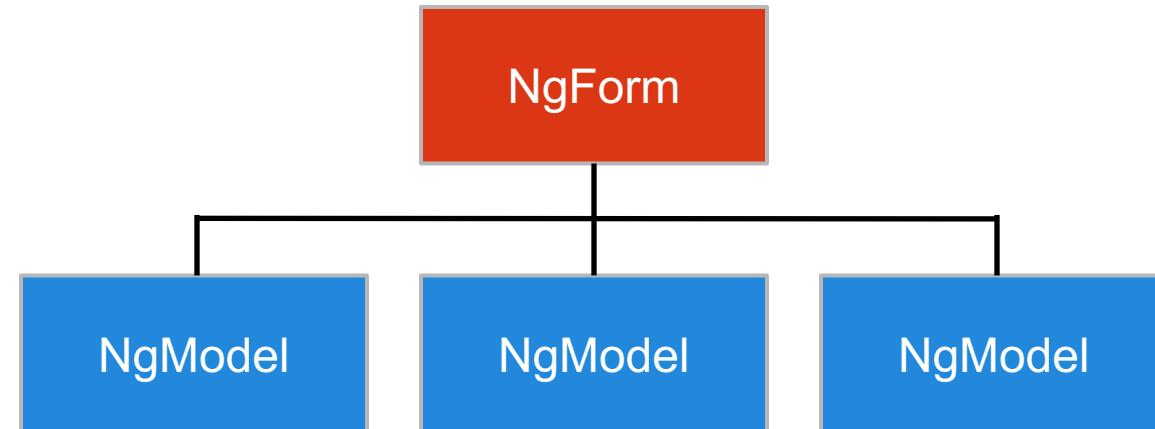
What are Template-Driven Forms?

- A component template is used to create a form and validate data provided by a model object (declarative approach)
- Two-way data binding, form control state and validation support are provided by using directives in the template



Template-Drive Form Hierarchy

- Angular template-driven forms rely on a hierarchy to handle data binding, validation and more
- NgForm acts as the parent for NgModel children



Key Template-Driven Form Directives

- Angular provides several key template-driven form directives:

Directive	Description
ngForm	Added to form elements that need to use the Angular form system
ngSubmit	Event triggered during a form submission
ngModel	Binds a model object to a form control. Provides change state, validation and CSS class support.
ngModelGroup	Allows one or more controls using ngModel to be grouped together.

Importing FormsModule

To get started using template-driven forms import **FormsModule**:

app.module.ts

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { AppComponent} from './app.component';

@NgModule({
  imports:      [ BrowserModule, FormsModule ],
  declarations: [ AppComponent ],
  bootstrap:   [ AppComponent ]
})
export class AppModule { }
```

Import template-driven
forms module

Using ngForm and ngModel

- ngForm and ngModel directives work together to provide change state and validation functionality:
 - Check if form/controls are dirty/pristine
 - Check if form/controls are valid/invalid

```
<form #myForm="ngForm" (ngSubmit)="onSubmit()">  
  <input type="text" name="city" #city="ngModel" [(ngModel)]="city" />  
  ...  
</form>
```

Get to instance of form

Register control with
ngForm instance

Steps to Create Template-Driven Forms

- 1 Add ngForm and ngSubmit to the <form> element
- 2 Add the name attribute and ngModel directive to form controls
- 3 Add a local template variable to form controls
- 4 Add validation directives
- 5 Show/Hide validation errors

Add ngForm and ngSubmit

Start by adding ngForm and ngSubmit to the form element

form.component.html

```
<form #nameForm="ngForm" (ngSubmit)="onSubmit()">  
  ...  
</form>
```

form.component.ts

```
@Component({...})  
export class FormComponent {  
  customer: ICustomer;  
  onSubmit() {  
    ...  
  }  
}
```

Steps to Create Template-Driven Forms

- 1 Add `ngForm` and `ngSubmit` to the `<form>` element
- 2 Add the `name` attribute and `ngModel` directive to form controls
- 3 Add a local template variable to form controls
- 4 Add validation directives
- 5 Show/Hide validation errors

Add the *name* Attribute and ngModel Directive

- Add the name attribute and ngModel directive to form controls to get change state, validation and CSS functionality

form.component.html

```
<form #nameForm="ngForm" (ngSubmit)="onSubmit()">
  Name: <input type="text" name="firstName" [(ngModel)]="customer.firstName" />
  <button type="submit">Submit</button>
</form>
```

Steps to Create Template-Driven Forms

- 1 Add `ngForm` and `ngSubmit` to the `<form>` element
- 2 Add the `name` attribute and `ngModel` directive to form controls
- 3 Add a local template variable to form controls
- 4 Add validation directives
- 5 Show/Hide validation errors

Add a Local Template Variable

- Add a local template variable to form controls to register it with **ngForm**
- Must reference **ngModel** to access validation and change information

form.component.html

```
<form #nameForm="ngForm" (ngSubmit)="onSubmit()">
  Name: <input type="text" name="firstName" [(ngModel)]="customer.firstName"
             #firstName="ngModel" />
  <button type="submit">Submit</submit>
</form>
```

Steps to Create Template-Driven Forms

- 1 Add `ngForm` and `ngSubmit` to the `<form>` element
- 2 Add the `name` attribute and `ngModel` directive to form controls
- 3 Add a local template variable to form controls
- 4 Add validation directives
- 5 Show/Hide validation errors

Add Validation Directives

- Angular includes *required*, *minlength*, *maxlength* and *pattern* validation directives

form.component.html

```
<form #nameForm="ngForm" (ngSubmit)="onSubmit()">
  Name: <input type="text" name="firstName" [(ngModel)]="customer.firstName"
              #firstName="ngModel" required />
  <button type="submit">Submit</submit>
</form>
```

Steps to Create Template-Driven Forms

- 1 Add `ngForm` and `ngSubmit` to the `<form>` element
- 2 Add the `name` attribute and `ngModel` directive to form controls
- 3 Add a local template variable to form controls
- 4 Add validation directives
- 5 Show/Hide validation errors

Show/Hide Validation Errors

- Use the formControlName object to access the state of the target control and determine if it's valid

form.component.html

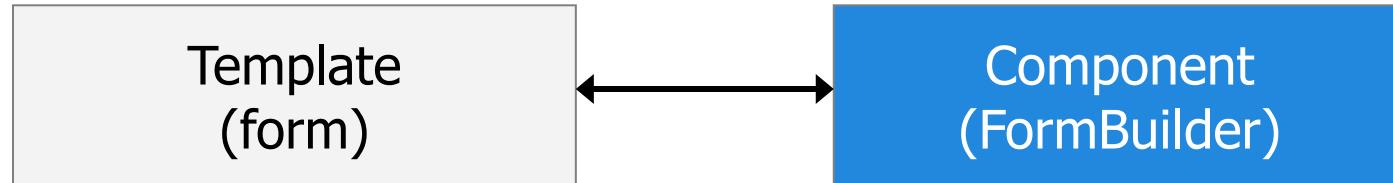
```
<form #nameForm="ngForm" (ngSubmit)="onSubmit()">
  Name: <input type="text" name="firstName" [(ngModel)]="customer.firstName"
               #firstName="ngModel" required />

  <span [hidden]="firstName.valid || firstName.pristine">
    Name is invalid
  </span>
  <button type="submit" [disabled]="!nameForm.valid">Submit</button>
</form>
```

Reactive Forms

What are Reactive Forms?

- Component code defines controls and validators that are used in a form (imperative approach)
- Relies on the FormBuilder service to create controls and validators and organize them into one or more groups



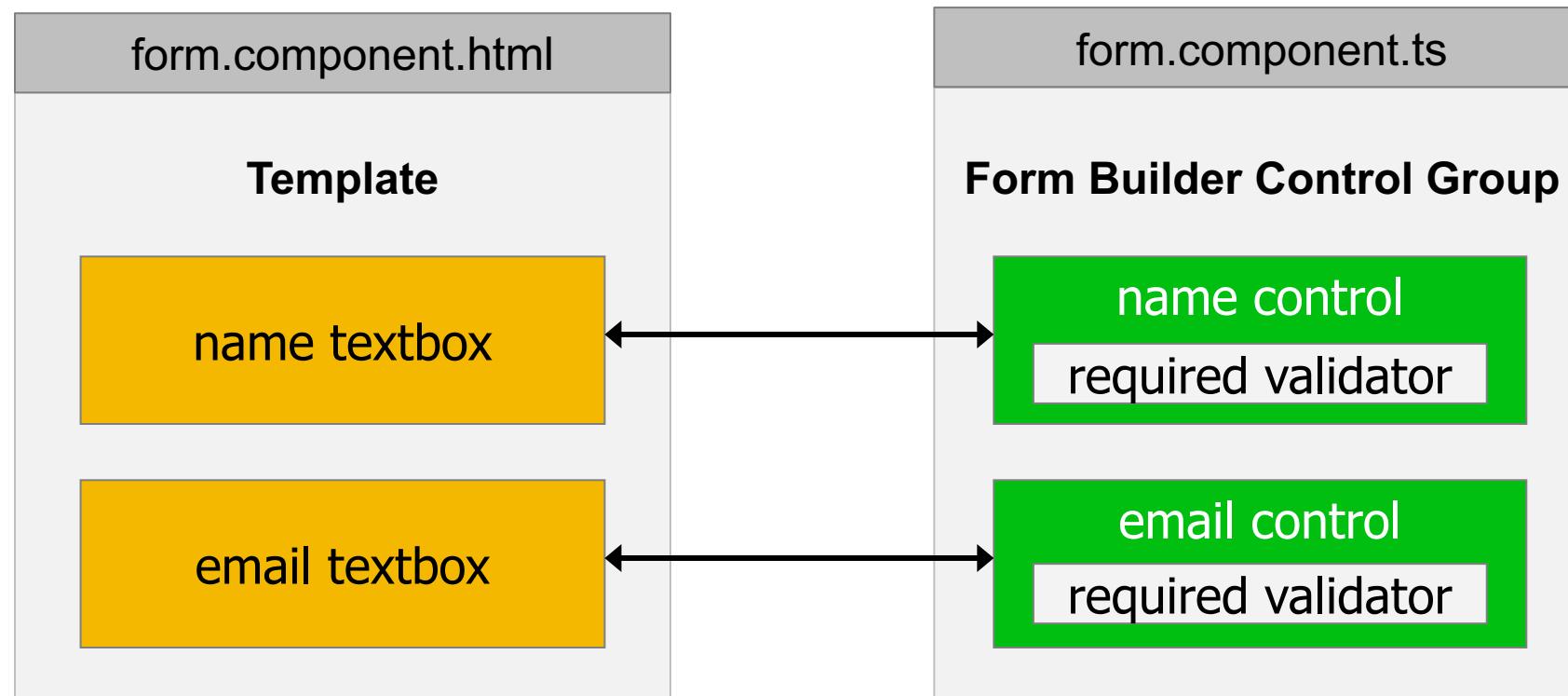
Key Model-Driven Form Directives

- Angular provides several key model-driven form directives:

Directive	Description
formGroup	Connects a FormGroup to a form
FormControlName	Connects a model with a form control

Reactive Forms Overview

- Form controls and validators are defined in component code
- Controls are bound to form input controls



Importing ReactiveFormsModule

To get started using reactive forms import **ReactiveFormsModule**:

app.module.ts

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { ReactiveFormsModule } from '@angular/forms';
import { AppComponent} from './app.component';

@NgModule({
  imports:      [ BrowserModule, ReactiveFormsModule ],
  declarations: [ AppComponent ],
  bootstrap:   [ AppComponent ]
})
export class AppModule { }
```

Import reactive forms
module

Creating a Form Group with FormBuilder

- FormBuilder is used to create one or more form control groups
- A control group defines controls
- Controls can have the following:
 - A default/initial value
 - One or more validators

Create a control group that can be bound to a form

```
this.form = this.formBuilder.group({  
  name: ['Jane', Validators.required],  
  alterEgo: ['Super Hero', Validators.required]  
});
```

Form Control Validators

- Angular provides several built-in validators:
 - required
 - minlength
 - maxlength
 - pattern
- Custom validators can be created and used

```
this.form = this.formBuilder.group({  
  name: [this.model.name, [Validators.required, Validators.maxLength(25)]],  
  alterEgo: [this.model.alterEgo, Validators.required]  
});
```

Steps to Create Reactive Forms

- 1 Create a FormGroup using FormBuilder
- 2 Add the formGroup directive to the form
- 3 Add formControlName to each form input control
- 4 Show/Hide validation errors

Create a FormGroup using FormBuilder

- FormBuilder provides a group() function that can be used to create a control group

form.component.ts

```
@Component({ selector: 'model-driven-form' })
export class ModelFormComponent implements OnInit {
  form: FormGroup;
  constructor(private formBuilder: FormBuilder) { }
  ngOnInit() {
    this.model = new Hero(18, 'Dr IQ', 'Really Smart', 'Chuck Overstreet', 'iq@superhero.com');
    this.form = this.formBuilder.group({
      name: [this.model.name, Validators.required],
      alterEgo: [this.model.alterEgo, Validators.required],
      email: [this.model.email, [Validators.required, ValidationService.emailValidator]],
      power: [this.model.power, Validators.required]
    });
  }
}
```

Steps to Create Reactive Forms

- 1 Create a FormGroup using FormBuilder
- 2 Add the `formGroup` directive to the form
- 3 Add the `FormControlName` directive to each form input
- 4 Show/Hide validation errors

Add the formGroup Directive

- Once a control group has been created, bind the form to it using the `formGroup` directive

form.component.html

```
<form [formGroup]="form" (ngSubmit)="onSubmit()">  
  ...  
</form>
```

FormGroup property defined in component

Steps to Create Reactive Forms

- 1 Create a FormGroup using FormBuilder
- 2 Add the formGroup directive to the form
- 3 Add the formControlName directive to each form input
- 4 Show/Hide validation errors

Add the formControlName directive

- Add formControlName to each form control to bind it to the respective "control" in the form group

form.component.html

```
<form [formGroup]="form" (ngSubmit)="onSubmit()">
  Name:      <input type="text" formControlName="name" />
  Alter Ego: <input type="text" formControlName="alterEgo" />
  Hero Email: <input type="email" formControlName="email" />
  Power:     <select formControlName="power">
    <option *ngFor="let p of powers" [value]="p">{{p}}</option>
  </select>
  ...
</form>
```

Steps to Create Reactive Forms

- 1 Create a FormGroup using FormBuilder
- 2 Add the formGroup directive to the form
- 3 Add the formControlName directive to each form input
- 4 Show/Hide validation errors

Show/Hide Validation Errors

- Use the FormGroup object to access controls and check validity

form.component.html

```
<form [formGroup]="form" (ngSubmit)="onSubmit()">
  Name:      <input type="text" formControlName="name" />
              <div [hidden]="form.controls.name.valid">Name is required</div>

  Alter Ego: <input type="text" formControlName="alterEgo" />
              <div [hidden]="form.controls.alterEgo.valid">
                Alter Ego is required
              </div>
  ...
</form>
```

Managing Form Control Styles

Managing Form Control Styles

- Using the `ngModel` or `formControlName` directives causes CSS classes to dynamically be added to form controls:

Name

Alter Ego

Hero Power

Submit

CSS Control Styles

- CSS classes added to form controls include:
 - ng-untouched
 - ng-touched
 - ng-pristine
 - ng-dirty
 - ng-valid
 - ng-invalid

CSS classes added by
forms directives

```
<input class="form-control ng-dirty ng-invalid ng-touched" name="name" required type="text" ng-reflect-model="ng-reflect-name="name">
```

Using Angular Form CSS Styles

- Example of defining styles for form control CSS classes

styles.css

```
.ng-invalid {  
    border-left: 5px solid #a94442;  
}  
  
.ng-valid {  
    border-left: 5px solid #42A948;  
}
```

Name

Alter Ego

Hero Power

Submit

Forms in Action

Summary

- Angular provides two options for building forms:
 - Template-Driven (declarative)
 - Reactive (imperative)
- `ngForm` and `formGroup` directives play a key role with forms
- `ngModel` can be used for "two-way" data binding
- `FormBuilder` can be used to create model-driven control groups
- `ngModel` and `formControlName` handle adding/removing CSS styles to form elements



Lab

Template and Reactive Forms





Angular Application Development



TypeScript JumpStart
(Bonus)

Agenda

- Introduction to TypeScript
- Types, Keywords and Hierarchy
- Classes, Properties and Functions
- Interfaces
- Namespaces and Modules
- Compiling TypeScript



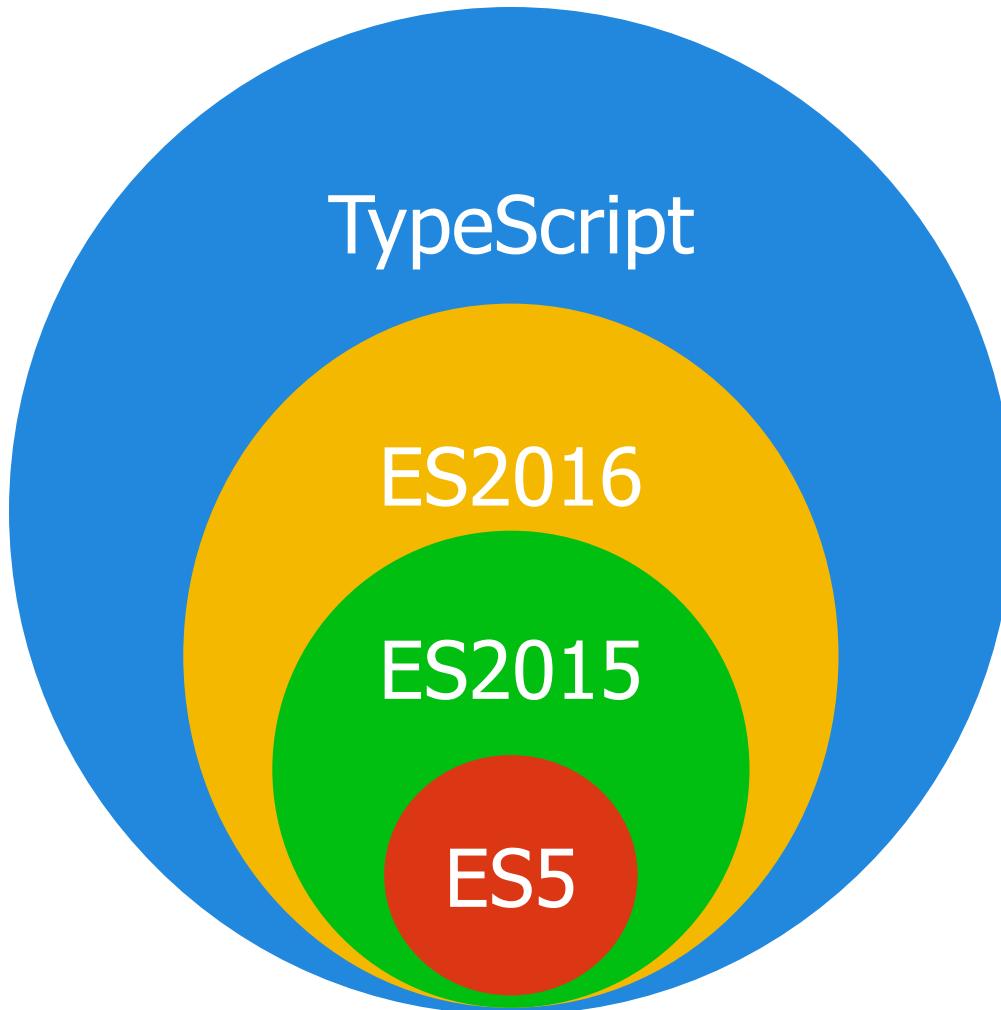
Introduction to TypeScript

What is TypeScript?

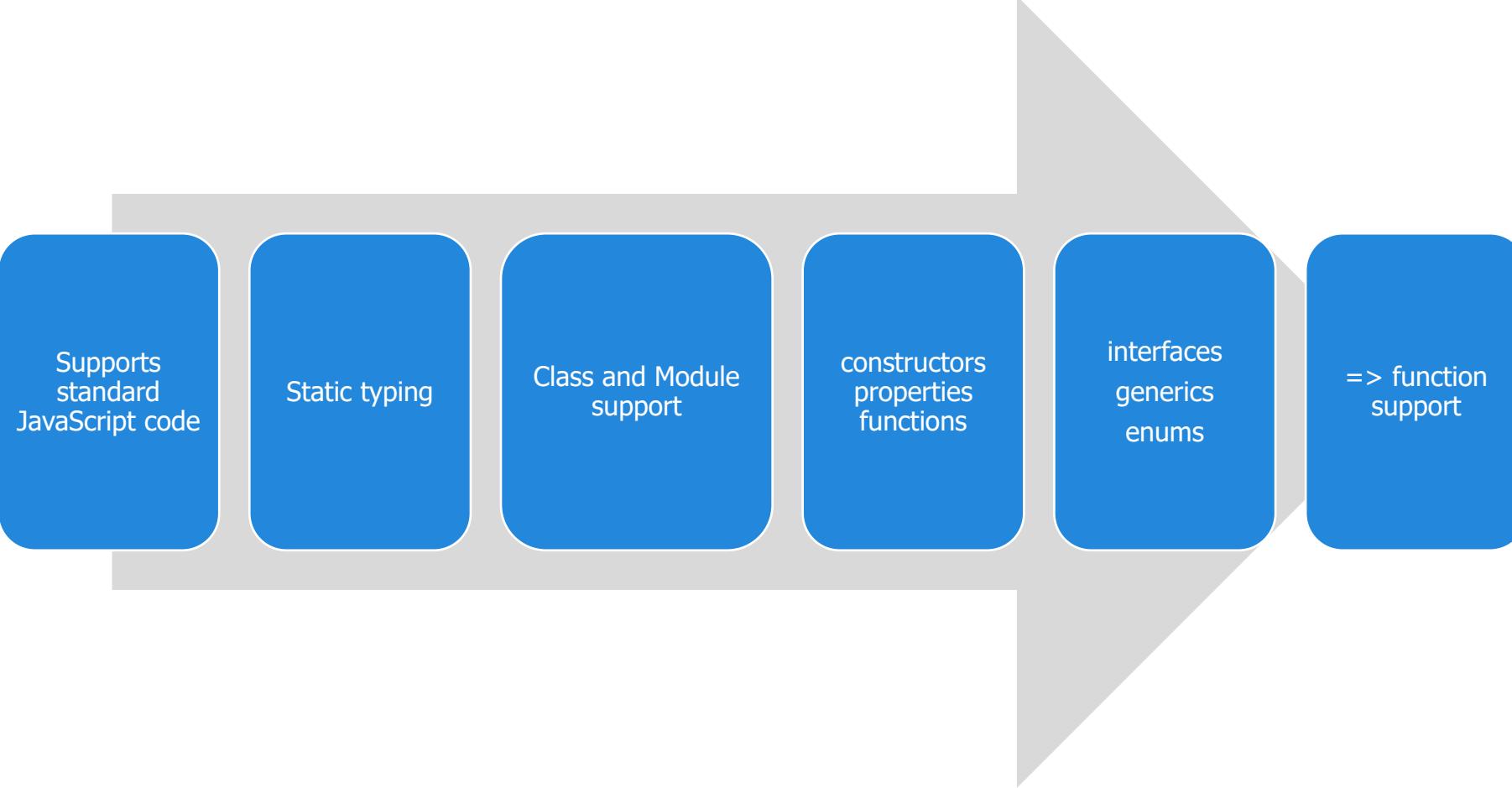
"TypeScript is a typed superset of JavaScript that compiles to plain JavaScript." ~ typescriptlang.org



TypeScript – A Superset of JavaScript



Key TypeScript Features



Supports standard JavaScript code

Static typing

Class and Module support

constructors
properties
functions

interfaces
generics
enums

=> function support

TypeScript → JavaScript

TypeScript

```
class Greeter {  
    greeting: string;  
    constructor (message: string) {  
        this.greeting = message;  
    }  
    greet() {  
        return "Hello, " + this.greeting;  
    }  
}
```

JavaScript

```
var Greeter = (function () {  
    function Greeter(message) {  
        this.greeting = message;  
    }  
    Greeter.prototype.greet = function () {  
        return "Hello, " + this.greeting;  
    };  
    return Greeter;  
})();
```



TypeScript is compiled/transpiled to JavaScript

Using the TypeScript Playground

Types, Keywords and Hierarchy

Types in TypeScript

```
var age: number = 2;
var score: number = 98.25;
var rating = 98.25;

var hasData: boolean = true;
var isReady = true;

var firstName: string = 'Scott';
var lastName = 'Johnson';

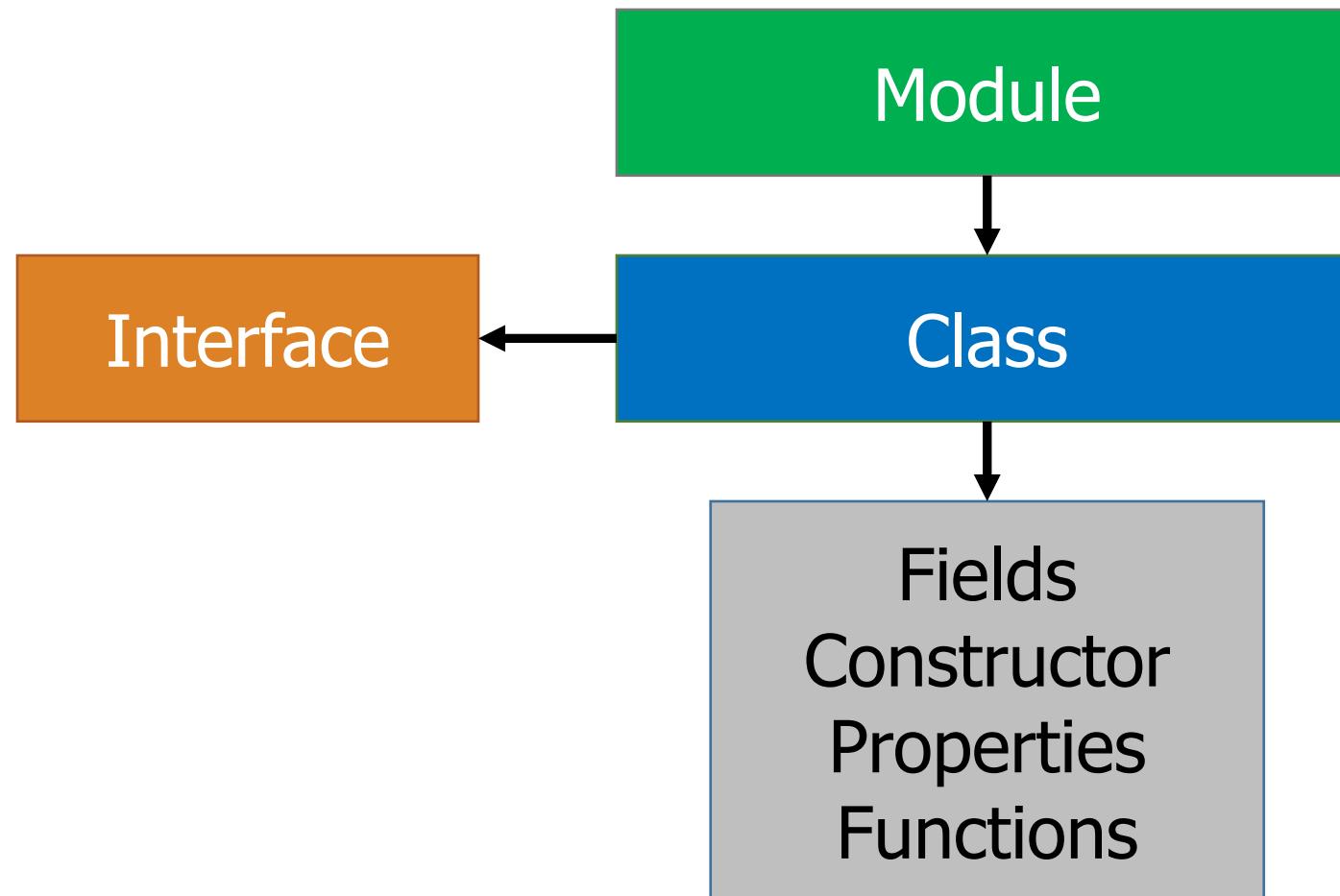
var names: string[] = ['John', 'Dan', 'Aaron', 'Fritz'];

var data: any;
var info;
```

Important Keywords and Operators

Keyword	Description
class	Container for members such as properties and functions
constructor	Provides initialization functionality in a class
exports	Export a member from a module
extends	Extend a class or interface
implements	Implement an interface
imports	Import a module
interface	Defines a code contract that can be implemented by types
namespace	Container for classes and other code
public/private	Member visibility modifiers
...	Rest parameter syntax
=>	Arrow syntax used with definitions and functions
<typeName>	< > characters use to cast/convert between types
:	Separator between variable/parameter names and types

Code Hierarchy



Types in Action

Classes, Properties and Functions

TypeScript Class Members

Fields

Constructors

Properties

Functions

Class Example

```
class Car {  
    engine: string;  
  
    constructor (engine: string) {  
        this.engine = engine;  
    }  
  
    start(): string {  
        return "Started " + this.engine;  
    }  
  
    stop(): string {  
        return "Stopped " + this.engine;  
    }  
}
```

Class members are
public by default

Extending a Class

```
class Auto {  
    engine: Engine;  
    constructor(engine: Engine) {  
        this.engine = engine;  
    }  
}
```

Truck derives from Auto

```
class Truck extends Auto {  
    fourByFour: boolean;  
    constructor(engine: Engine, fourByFour: boolean) {  
        super(engine);  
        this.fourByFour = fourByFour;  
    }  
}
```

Call base class
constructor

Classes in Action

Interfaces

TypeScript Interfaces

- Interfaces act as "code contracts"
- Only used during design time
- Use the **interface** keyword
- Useful for enforcing constraints:
 - Classes
 - Constructors
 - Methods
 - Properties



Defining an Interface

```
interface IEngine {  
    engineType: string;  
    start() : void;  
    stop() : void;  
}
```



IEngine Interface
defines 3 members

Implementing an Interface

```
class Engine implements IEngine {  
    engineType: string;  
  
    constructor(engineType: string) {  
        this.engineType = engineType;  
    }  
  
    start() : void {  
        ...  
    }  
    stop() : void {  
        ...  
    }  
}
```

Interfaces provide a
way to enforce a
"contract"

Interfaces in Action

Namespaces and Modules

Defining Namespaces

```
namespace Shapes {  
  
    export class Rectangle {  
  
        constructor (public height: number, public width: number) {  
  
        }  
    }  
  
    var myRectangle = new Shapes.Rectangle(2,4);
```

Accessible because it
was exported

Creating Modules

calculator.ts

```
export class Calculator {  
    add(x: number, y: number) {  
        return x + y;  
    }  
}
```

main.ts

```
import { Calculator } from './calculator';  
  
function run() {  
    let calc = new Calculator();  
    console.log(calc.add(2,2));  
}  
}
```

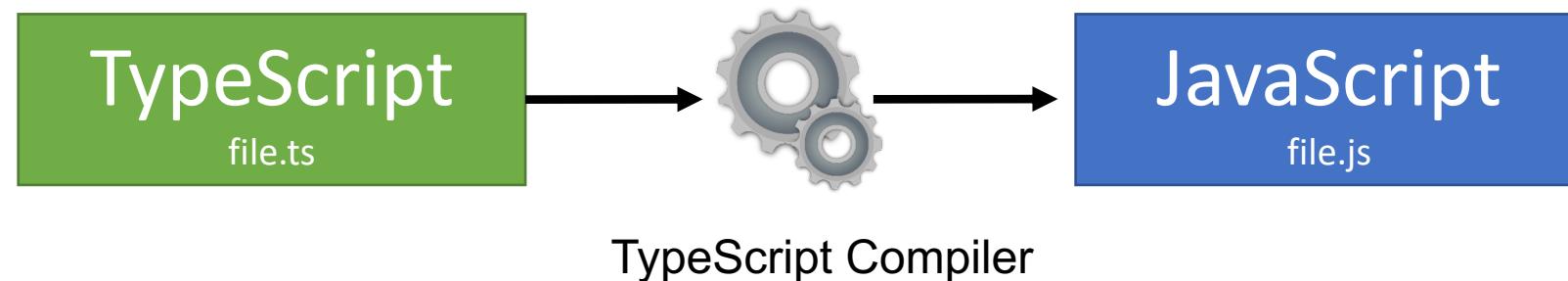


ES2015 modules
require a module
loader

Compiling TypeScript

TypeScript Compilation/Transpilation

- The TypeScript compiler can generate ES3, ES5 or ES6 code
- Runs via the command-line, with build tools (Grunt/Gulp) or dynamically in the browser



Compiling TypeScript

Steps to compile with the command window:

1. Create a package.json file:

```
npm init
```

2. Install the TypeScript npm module:

```
npm install typescript --save-dev
```

3. Update the "scripts" property in package.json:

```
"scripts": { "tsc:w": "tsc --watch" }
```

4. Start the TypeScript compiler in the project folder:

```
npm run tsc:w
```



The tsconfig.json File

- TypeScript compilation settings can be defined in tsconfig.json

tsconfig.json

```
{  
  "compilerOptions": {  
    "target": "es5",  
    "module": "commonjs",  
    "moduleResolution": "node",  
    "sourceMap": true,  
    "noImplicitAny": true  
  },  
  "exclude": [ "node_modules" ]  
}
```

TypeScript Compilation in Action

Summary

- TypeScript is a superset of JavaScript
- Compiled to ES3, ES5 or ES6 compatible code
- Many TypeScript features are from ES6/ES2015
- Support for types, classes, modules and more

Lab

Getting Started with TypeScript





Angular Application Development



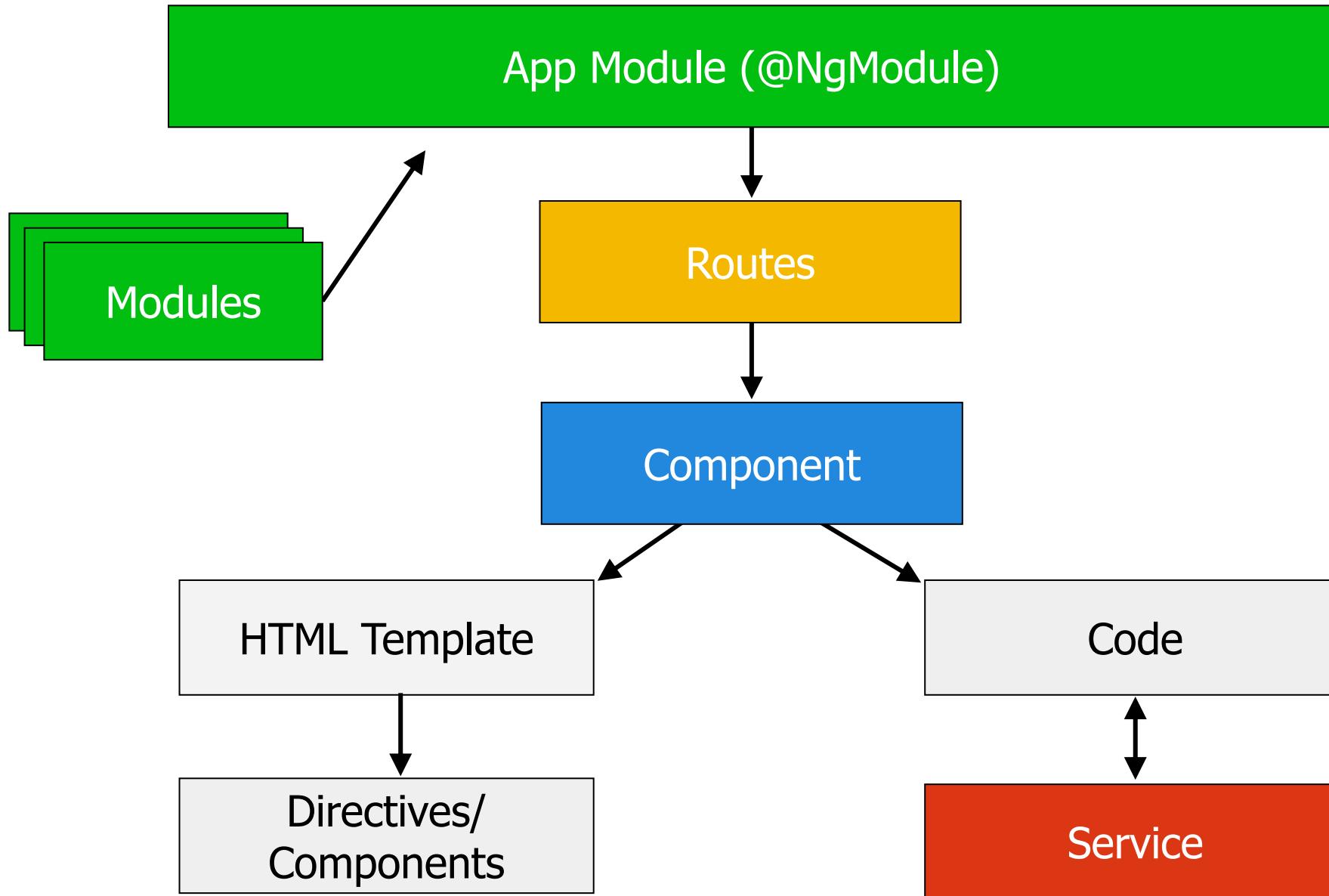
Custom Directives
and Components
(Bonus)

Agenda

- Directives Overview
- Building a Custom Sorting Directive
- Building a Custom FilterTextbox Component
- Building a Custom Mapping Component



The Big Picture



Directives Overview

Directives Overview

- Angular provides three categories of directives:

Components

Attribute
Directives

Structural
Directives

Attribute Directives

- Attribute directives change the appearance or behavior of an element
- Examples:
 - ngModel
 - ngClass
 - ngStyle

Attribute
Directives

Attribute Directive Example

- Attribute directives can change the appearance or behavior of an element

bgChanger.directive.ts

```
import { Directive, ElementRef, OnInit } from '@angular/core';
@Directive({
  selector: '[bgChanger]'
})
export class BgChangerDirective implements OnInit {
  constructor(private el: ElementRef) { }

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

my.component.html

```
<div bgChanger>Hello</div>
```

Structural Directives

- Structural directives change the overall DOM structure by adding or removing nodes
- Examples:
 - ngFor
 - ngIf
 - ngSwitch

Structural
Directives

Structural Directive Example

- Structural directives can change the DOM structure

startStop.directive.ts

```
import {Directive, Input, TemplateRef, ViewContainerRef}
  from '@angular/core';
@Directive({ selector: '[startStop]' })
export class StartStopDirective {
  constructor(private templateRef: TemplateRef,
              private viewContainer: ViewContainerRef) { }
  @Input() set startStop(started: boolean) {
    if (!started) {
      this.viewContainer.createEmbeddedView(this.templateRef);
    } else {
      this.viewContainer.clear();
    }
  }
}
```

my.component.html

```
<p *startStop="started">
  start
</p>

<p *startStop="!started">
  stop
</p>
```

Building a Custom Sorting Directive

The SortBy Directive

- The SortBy directive provides sorting functionality for tabular data

 Customer Manager

Customers

[Card View](#) [List View](#)

	First Name	Last Name	Address	City	State	Order Total	
	Heedy	Wahlin	4651 Tuvo St.	Chandler	Arizona	\$9.99	View Orders
	Tina	Adams	79455 Pinetop Way	Seattle	Washington	\$15.99	View Orders
	Misko	Hevery	9812 Builtway Appt #1	Mountain View	California	\$19.99	View Orders

SortBy Directive Input/Output Properties

- The SortBy directive has the following properties:

Property	Description
sortBy	@Input property used to define the "sort by" column
sorted	@Output property used to notify subscribers when a table header column has been clicked

The sortBy Property

- The **sortBy** @Input property allows the "sort by" column to be passed into the directive

sortby.directive.ts

```
import { Directive, Input, Output, EventEmitter } from '@angular/core';

@Directive({ selector: '[sort-by]' })
export class SortByDirective {
  private sortProperty: string;
  @Input('sort-by')                                "alias" given to property that
  set sortBy(value: string) {                      matches with selector
    this.sortProperty = value;
  }
}
```

The sorted Property

- **sorted** @Output property is used to raise a "sorted" event to any subscribers to notify them when a header column has been clicked:

sortby.directive.ts

```
@Directive({
  selector: '[sort-by]',
  host: { '(click)': 'onClick($event)' }
})
export class SortByDirective {
  ...
  @Output() sorted: EventEmitter<string> = new EventEmitter<string>();
  onClick(event: any) {
    event.preventDefault();
    this.sorted.emit(this.sortProperty);
  }
}
```

host property lets us access events that occur on the directive's host element

Raise the sorted event

Putting it All Together

sortby.directive.ts

```
import { Directive, Input, Output, EventEmitter } from '@angular/core';
@Directive({
  selector: '[sort-by]',
  host: { '(click)': 'onClick($event)' }
})
export class SortByDirective {
  private sortProperty: string;
  @Output() sorted: EventEmitter<string> = new EventEmitter<string>();
  @Input('sort-by')
  set sortBy(value: string) {
    this.sortProperty = value;
  }
  onClick(event: any) {
    event.preventDefault();
    this.sorted.emit(this.sortProperty); //Raise clicked event
  }
}
```

Using the sortBy Directive

- sorted @Output property is used to raise a "sorted" event to subscribers to notify them when a header column has been clicked:

customersGrid.component.ts

```
import { SortByDirective } from '../sortby.directive';
import { SorterService } from '../sorter.service';

@Component({
  selector: 'customers-grid'
})
export class CustomersGridComponent {
  constructor(private sorter: SorterService) { }

  sort(prop: string) {
    this.sorter.sort(this.customers, prop);
  }
}
```

customersGrid.component.html

```
<tr>
  <th>&nbsp;</th>
  <th sort-by="firstName"
       (sorted)="sort($event)">
    First Name
  </th>
</tr>
```

Is the sort-by Directive Necessary?

- A click event could be bound directly to the component's sort() function, but SortBy provides a nice/simple example of a directive using input/output properties

customersGrid.component.ts

```
@Component({  
  ...  
})  
export class CustomersGridComponent {  
  
  sort(prop: string) { ←  
    this.sorter.sort(this.customers, prop);  
  }  
  
}
```

customersGrid.component.html

```
<tr>  
  <th>&nbsp;</th>  
  <th (click)="sort('firstName')">  
    First Name  
  </th>  
</tr>
```

Building a Custom FilterTextbox Component

The FilterTextbox Component

FilterTextbox is a re-useable component that can filter data

Card View List View

Filter:

Ted James   Phoenix, Arizona View Orders	Michelle Thompson   Los Angeles, California View Orders	Zed Bishop   Las Vegas, Nevada View Orders	Tina Adams   Seattle, Washington View Orders
Igor Minar   Chandler, Arizona View Orders	Brad Green   Mountain View, California View Orders	Misko Hevery   Mountain View, California View Orders	Heedy Wahlin   Chandler, Arizona View Orders
John Papa   Orlando, Florida View Orders	Tonya Smith   Atlanta, Georgia View Orders	Ward Bell   San Francisco, California View Orders	

The FilterTextbox Template

- The FilterTextbox component provides a simple template with a label and an input control:

```
@Component({
  selector: 'filter-textbox',
  template: `
    <form>
      Filter: <input type="text" [(ngModel)]="model.filter"
                (keyup)="filterChanged($event)"  />
    </form>
  `,
})
```

The FilterTextbox Code

- The FilterTextbox component code hooks the keyup event to a function and exposes a **changed** Output property:

```
export class FilterTextboxComponent {  
  model: { filter: string } = { filter: null };  
  @Output() changed: EventEmitter<string> = new EventEmitter<string>();  
  
  filterChanged(event: any) {  
    event.preventDefault();  
    this.changed.emit(this.model.filter); //Raise changed event  
  }  
}
```

Enable events to be raised

Raise the Event

Using the FilterTextbox Component

- The FilterTextbox component can be used by including it in the host component's **directives** property and then using its selector in a template:

```
<filter-textbox class="navbar-right" (changed)="filterChanged($event)">
</filter-textbox>
```

Building a Custom Mapping Component

MapComponent

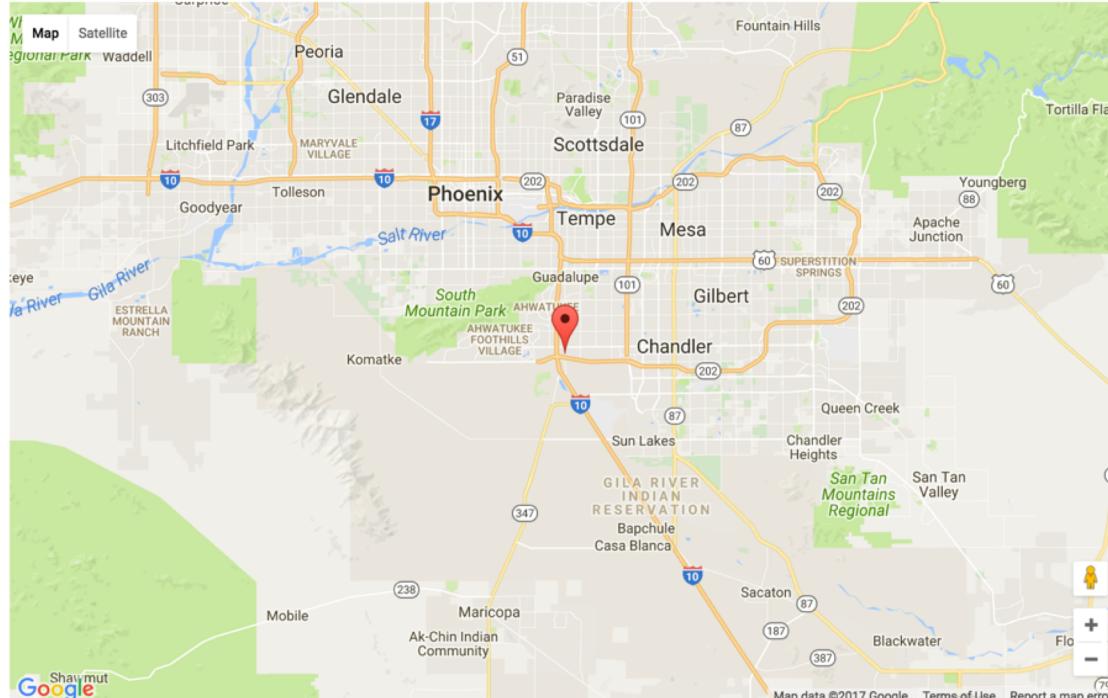
MapComponent is a re-useable component that can convert longitude and latitude into points on a Google map

Customer Manager Customers Orders About Login

Customer Information

Customer Details Customer Orders Edit Customer

Ted James
1234 Anywhere St.
Phoenix, Arizona



A screenshot of a web application titled "Customer Manager". The top navigation bar includes links for "Customer Manager", "Customers", "Orders", "About", and "Login". Below this, a section titled "Customer Information" shows a user profile for "Ted James" with the address "1234 Anywhere St., Phoenix, Arizona". At the bottom of the page is a detailed map of the Phoenix metropolitan area, featuring the Salt River and Gila River, and numerous towns and cities including Surprise, Peoria, Glendale, Phoenix, Tempe, Mesa, Gilbert, Chandler, and Scottsdale. A red marker is placed on the map near the center of the city.

Displaying a Location

MapComponent provides input properties that can be used to set a location on a map:

```
<cm-map [latitude]="customer.latitude"
         [longitude]="customer.longitude"
         [zoom]="10"
         [enabled]="mapEnabled"
         [markerText]="customer.firstName + ' ' + customer.lastName">
</cm-map>
```

Displaying Multiple Locations

Multiple locations can be mapped by defining map-points:

```
<cm-map [zoom]="2" [enabled]="displayMode === displayModeEnum.Map">
  <cm-map-point *ngFor="let customer of filteredCustomers"
    [latitude]="customer.latitude"
    [longitude]="customer.longitude"
    [markerText]="customer.firstName + ' ' + customer.lastName">
  </cm-map-point>
</cm-map>
```

MapComponent Code

```
@Component({
  selector: 'cm-map',
  templateUrl: 'map.component.html',
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class MapComponent implements OnInit, AfterContentInit {  
}
```

MapComponent Template

MapComponent provides a simple template with a single `<div>` that acts as the container for the map:

```
<div #mapContainer [style.height]="mapHeight" [style.width]="mapWidth">  
  Map Loading...  
</div>
```

Local template variable

MapComponent Input Properties

- MapComponent exposes several input properties
- Provides get/set property accessors for the **enabled** property

```
export class MapComponent implements OnInit, AfterContentInit {  
    ...  
    @Input() height: number;  
    @Input() width: number;  
    @Input() latitude: number = 34.5133;  
    @Input() longitude: number = -94.1629;  
    @Input() markerText: string = 'Your Location';  
    @Input() zoom: number = 8;  
    @Input() get enabled(): boolean { return this.isEnabled; }  
    set enabled(isEnabled: boolean) {  
        this.isEnabled = isEnabled;  
        this.init();  
    }  
}
```

Simple way to detect when the
enabled property has changed

Additional MapComponent Properties

- `@ViewChild` can be used to find a local template variable
- `@ContentChildren` can be used to locate children in the component

```
export class MapComponent implements OnInit, AfterContentInit {  
    ...  
    @ViewChild('mapContainer') mapDiv : ElementRef;  
  
    @ContentChildren(MapPointComponent) mapPoints : QueryList<MapPointComponent>;  
}
```

Get to the DOM element that acts
as the container for the map

Locate `MapPointComponent` children
defined in the `MapComponent`

Rendering a Map

MapComponent uses the Google Maps API to render a map

```
private renderMap() {  
    const latlng = this.createLatLong(this.latitude, this.longitude);  
    const options = {  
        zoom: this.zoom,  
        center: latlng,  
        mapTypeControl: true,  
        mapTypeId: google.maps.MapTypeId.ROADMAP  
    };  
  
    this.map = new google.maps.Map(this.mapDiv.nativeElement, options);  
    if (this.mapPoints && this.mapPoints.length) { this.renderMapPoints();  
}  
    else { this.createMarker(latlng, this.map, this.markerText); }  
}
```

Summary

- Structural directives such as ngFor and ngIf manipulate the DOM structure
- Attribute directives are used to change the look of an element or handle different behaviors
- EventEmitter provides a simple way to raise events that other components/directives can subscribe to
- @ViewChild and @ContentChildren can be used to access the component's view and any custom content





Angular Application Development



Introduction to webpack (Bonus)

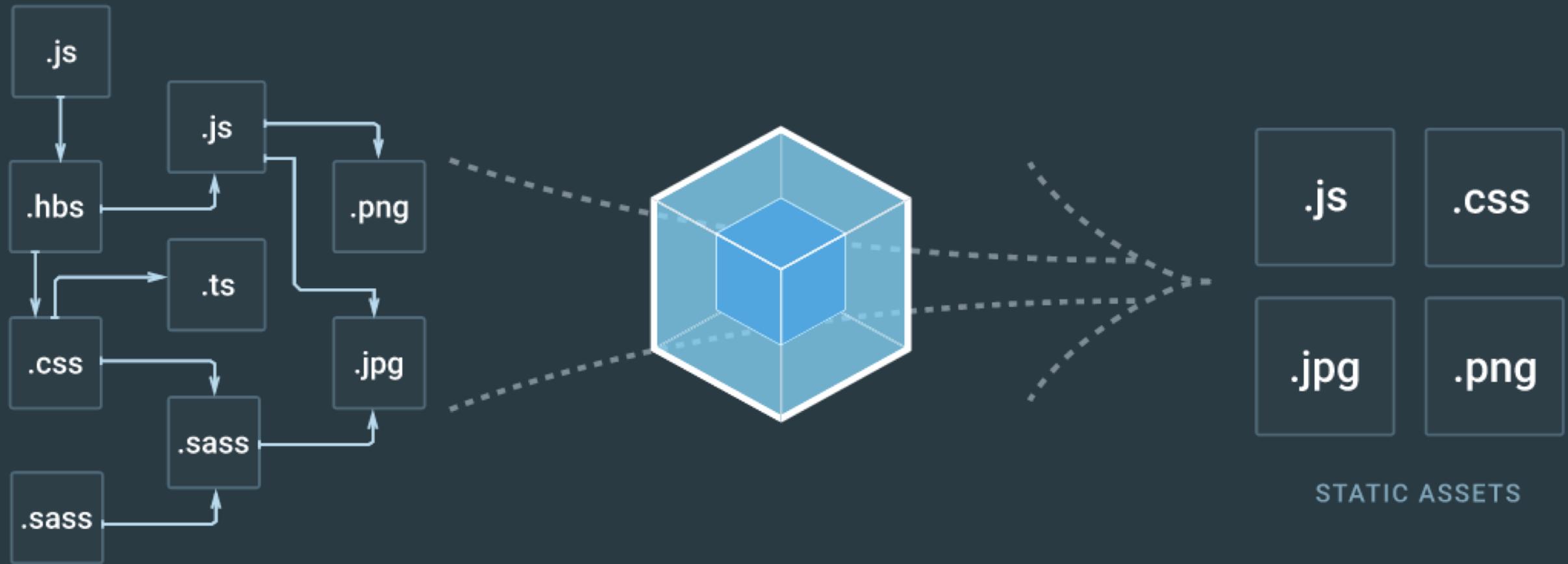
Agenda

- webpack Overview
- webpack Building Blocks
- webpack Files



webpack Overview

bundle your assets

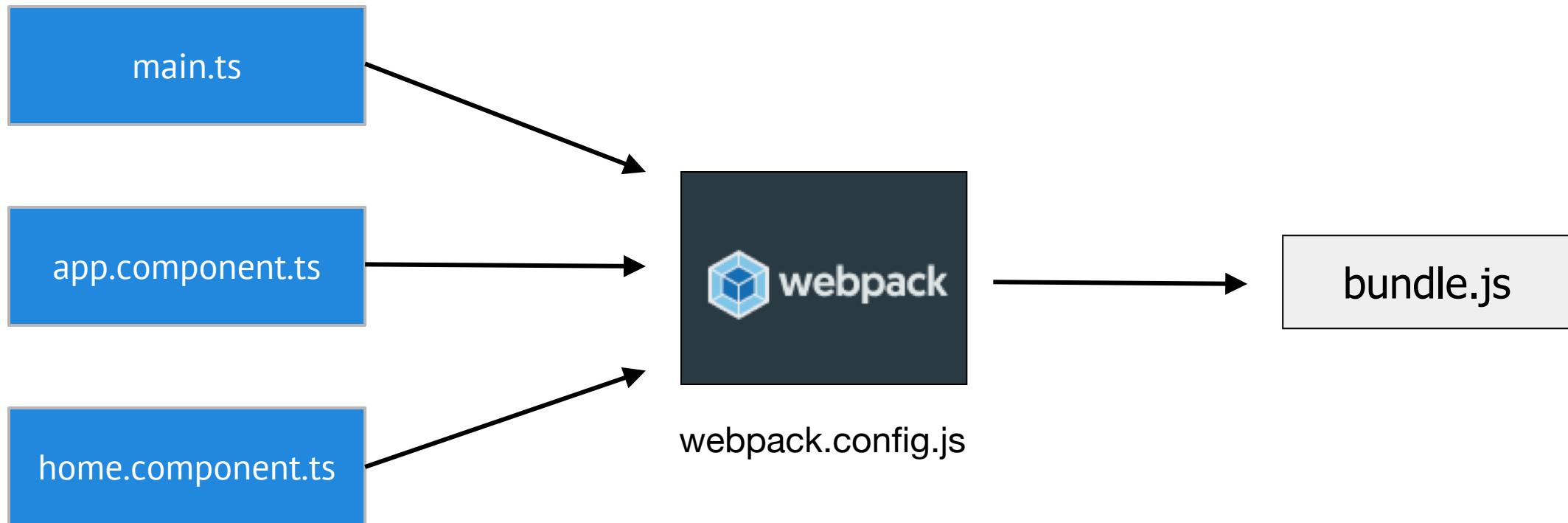


MODULES WITH DEPENDENCIES

What is webpack?

- webpack is a *module bundler* that can be used with JavaScript applications
- Key Features:
 - Creates a graph of application dependencies
 - Loads and transforms files and adds them to the dependency graph
 - Dependencies are converted into a "bundle"
 - Minimize size, HTTP requests and more with bundles

webpack Bundles



Key webpack Features

- Convert from one file format to another
 - ES2015 → ES5
 - TypeScript → ES5
 - SASS/LESS → CSS
- Generate bundles (including code splitting bundles)
- Run a development web server
- `webpack.config.js` provides configuration details

Installing webpack

- webpack can be installed using standard npm commands:

```
npm install webpack --save-dev
```

webpack.config.js

webpack relies on a configuration file to define an application entry point, how to load assets and where to output bundles

webpack.config.js

```
const webpack = require('webpack');

module.exports = {

  //webpack configuration

};


```

webpack Building Blocks

webpack Building Blocks

Entry

Output

Loaders

Plugins

Entry

Entry defines the starting point of the dependency graph

webpack.config.js

```
module.exports = {
  entry: './src/app/main.ts'
};
```

Output

Determines where the output bundle should be created

webpack.config.js

```
module.exports = {
  output: {
    path: path.resolve(rootDir, './src/devDist'),
    filename: '[name].js'
  }
};
```

Loaders

Defines rules that convert an application file to a new source and add it to the dependency graph. Similar to "tasks" in other build tools.

webpack.config.js

```
module.exports = {
  module: {
    loaders: [
      { test: /\.ts$/, loader: 'awesome-typescript-loader' }
    ]
  }
};
```

Plugins

Perform actions and custom functionality on bundled modules

webpack.config.js

```
module.exports = {
  plugins: [
    new webpack.optimize.UglifyJsPlugin({
      compress: {
        warnings: false
      },
      output: {
        comments: false
      },
      sourceMap: true
    })
  ]
};
```

webpack Files

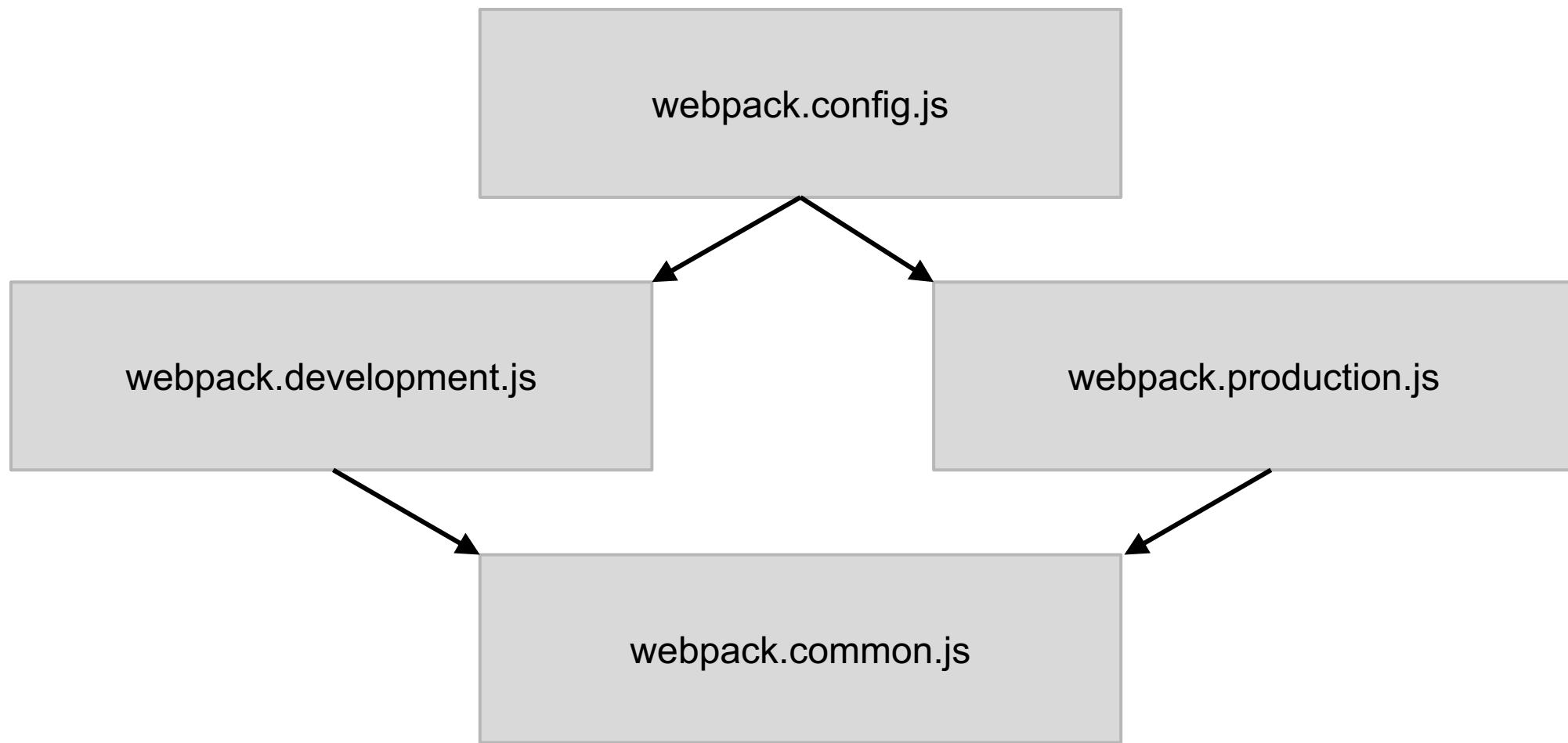
webpack Files

- webpack configuration can be defined in a single file or multiple files
- The Angular CLI "hides" webpack functionality by default
- To see the Angular CLI webpack configuration file you can run:

`ng eject`

- The content that follows shows how a custom webpack solution can be created

Organizing webpack Files



webpack.config.js

Loads development or production config based on NODE_ENV setting

webpack.config.js

```
const env = process.env.NODE_ENV || 'development';
console.log('Running Webpack in ' + env + ' mode.');

if (env === 'development') {
  module.exports = require('./config/webpack.development');
} else {
  module.exports = require('./config/webpack.production');
}
```

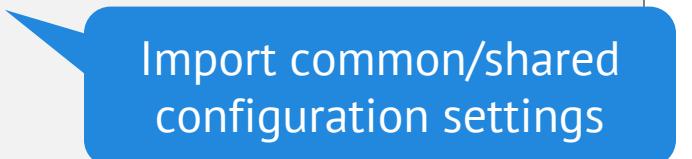
webpack.development.js

Produces development environment bundles

webpack.development.js

```
const webpack = require('webpack'),
  webpackMerge = require('webpack-merge'),
  ExtractTextPlugin = require('extract-text-webpack-plugin'),
commonConfig = require('./webpack.common.js'),
  path = require('path'),
  rootDir = path.resolve(__dirname, '..');

module.exports = webpackMerge(commonConfig, {
  devtool: 'source-map',
  output: { ... },
  module: { loaders: [ ... ] }
});
```



Import common/shared configuration settings

webpack.production.js

Produces production environment bundles

webpack.production.js

```
const webpack = require('webpack'),
      webpackMerge = require('webpack-merge'),
      ExtractTextPlugin = require('extract-text-webpack-plugin'),
      commonConfig = require('./webpack.common.js'),
      ngToolsWebpack = require('@ngtools/webpack')
      path = require('path'),
      rootDir = path.resolve(__dirname, '..');

module.exports = webpackMerge(commonConfig, {
  entry { app: './src/app/main.aot.ts' },
  output: { ... },
  module: { loaders: [ { test: /\.ts$/, loader: '@ngtools/webpack' } ] },
  plugins: [ //AOT, UglifyJS
});
```

Import ngtools AOT module

Define AOT entry file location

webpack.common.js

Defines common/shared functionality used in development and production builds

webpack.common.js

```
const webpack = require('webpack'),
  HtmlWebpackPlugin = require('html-webpack-plugin'),
  ExtractTextPlugin = require('extract-text-webpack-plugin'),
  path = require('path'),
  rootDir = path.resolve(__dirname, '..');

module.exports = {
  resolve: { //resolve imports },
  entry { // main.ts, vendor.ts and polyfills.ts },
  module: { loaders: [ //html, image and css loaders ] },
  plugins: [ //ExtractTextPlugin, CommonsChunkPlugin, HtmlWebpackPlugin ]
};
```

Provides configuration shared between development and production

Summary

- webpack provides module bundling functionality for JavaScript applications
- Install from npm
- Configuration is defined in webpack.config.js
- Relies on several key building blocks:
 - Entry
 - Output
 - Loaders
 - Plugins





Angular Application Development



Unit Testing
(Bonus)

Agenda

- Angular Unit Testing Features
- Unit Testing Players
- Test Suites, Specs and Expectations Overview
- Creating an Angular Service Test Suite and Spec
- Creating an Angular Component Test Suite and Spec
- Mocking Objects



Angular Unit Testing Features

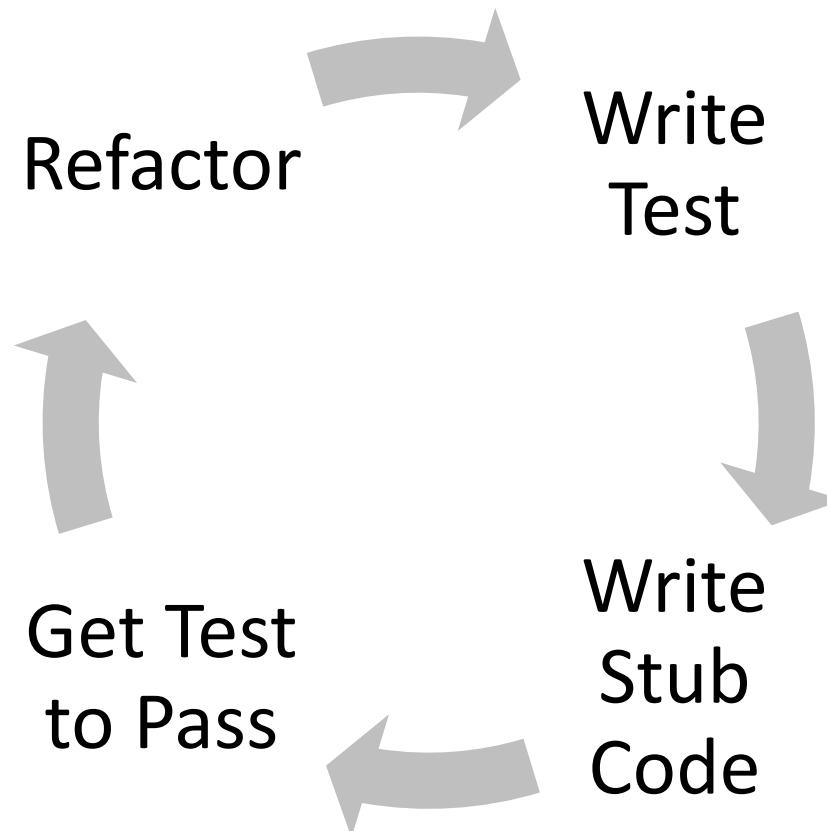
Angular Unit Testing Features

- Framework written with testability in mind
- Drives Separation of Concerns (SOC)
- Relies on dependency injection
- Natively supports object "mocking"
- Can use different test frameworks (jasmine, mocha, etc.)
- Angular CLI supports creating and running tests directly

Unit Testing Benefits

- Forces more thought about the application's design
- Simplifies code integration from team members
- Catch bugs early in the development lifecycle
- Leads to more modular code with less dependencies
- Easier to change and maintain code
- Tests provide a simple way to view an application's API

Test Driven Development (TDD) Process



Unit Test Example

my-component.spec.ts

```
describe('my-component tests', () => {  
  it('true is true', () => {  
    expect(true).toBe(true);  
  });  
});
```

What we're testing (it always
passes in this example ☺)

Unit Testing Players

Angular Unit Testing Players

Karma

Jasmine

Angular Testing
Utilities

* Note that other testing libraries and utilities (Mocha, Sinon, Chai, etc.) can be used as well

What is Karma?

- Karma is a test runner that can be installed using npm
- Key Features
 - Test on real browsers and devices
 - Tests can automatically be re-run when a file changes
 - Control through the command line or IDE
 - Supports multiple test frameworks (Jasmine, Mocha, Qunit, etc.)
 - Debug tests directly in Chrome
- Used by the Angular CLI
- <http://karma-runner.github.io>

What is Jasmine?

- Jasmine is a test framework for JavaScript code:
 - **describe()** – Used to define a test suite
 - **it()** - Used to define test "specs"
 - **expect()** - Used to perform a test assertion
- Provides built-in support for expectations, mocking and spies
- Karma can run tests/specs defined with Jasmine
- <http://jasmine.github.io>



What are Angular Testing Utilities?

- Adds Angular specific functionality into the test environment
- Provides **TestBed** and **ComponentFixture** classes to initialize Angular modules and create components
- Provides testing functions:
 - `async()`
 - `fakeAsync()`
 - `inject()`
 - More...

The Big Picture

Jasmine Test Suite

beforeEach

Spec

Spec

Test Suites, Specs and Expectations Overview

Test Suites and Specs

Jasmine allows test suites and specs to be defined

```
describe('A suite is just a function', () => {  
  var a;  
  
  it('is a spec', function () {  
    a = true;  
    expect(a).toBe(true);  
  });  
});
```

Test Suite

Test Spec

Grouping Multiple Specs in a Suite

```
describe('A test suite', () => {

  it('is just a function, so it can contain any code', () => {
    var foo = 0;
    foo += 1;
    expect(foo).toEqual(1);
  });

  it('can have more than one expectation', () => {
    var foo = 0;
    foo += 1;
    expect(foo).toEqual(1);
    expect(true).toEqual(true);
  });

});
```

Examples of Expectations

```
var a = 10, b = a, message = 'foo bar baz', status = true;  
  
expect(a).toEqual(10);  
expect(a).toBe(b);  
expect(a).not.toBe(null);  
expect(a).not.toBeNull();  
expect(pi).toBeGreaterThan(5);  
expect(message).toMatch(/bar/);  
  
expect(message).toMatch('bar');  
expect(message).not.toMatch(/quux/);  
expect(a.bar).not.toBeDefined();  
expect(a.bar).toBeUndefined();  
expect(status).toBeTruthy();  
expect(status).not.toBeFalsy();
```

Creating an Angular Service Test Suite and Spec

Creating an Angular Service Test Suite and Spec

1. Create a test suite
2. Add a beforeEach()
3. Use TestBed to configure the module
4. Add a test spec with one or more expectations

Step 1: Create a Test Suite

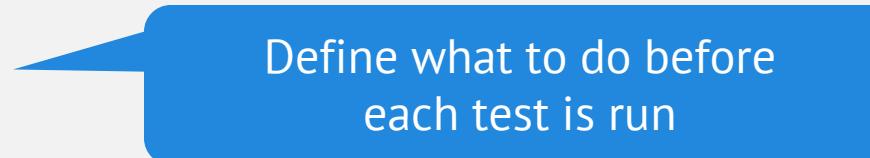
data.service.spec.ts

```
describe('DataService Tests', () => {  
});
```

Step 2: Add a beforeEach()

data.service.spec.ts

```
describe('DataService Tests', () => {  
  beforeEach(() => {  
    });  
});
```



Define what to do before
each test is run

Step 3: Use TestBed to Configure the Module

data.service.spec.ts

```
describe('DataService Tests', () => {  
  beforeEach(() => {  
    TestBed.configureTestingModule({  
      providers: [ DataService ]  
    });  
  });  
});
```

Configure the Angular module
for the service test

Step 4: Add a Test Spec

```
data.service.spec.ts
```

```
describe('DataService Tests', () => {  
  beforeEach(() => {  
    TestBed.configureTestingModule({  
      providers: [ DataService ]  
    });  
  });  
  
  it('should be created', inject([DataService], (service: DataService) => {  
    expect(service).toBeTruthy();  
  }));  
});
```

Define the spec (test) and
inject service to be tested

Define spec expectation

Exploring a Service Test Suite and Specs

Explore the following features:

- Imports included at the top of the spec file
- Test Suite definition
- The beforeEach() code (mocking will be covered later)
- TestBed code
- The first spec

[Samples/Unit-Testing/customers-orders/src/app/core/services/data.service.spec.ts](#)

Creating an Angular Component Test Suite and Spec

Creating an Angular Component Test Suite and Spec

1. Create a test suite
2. Add a beforeEach()
3. Use TestBed to configure the module
4. Use TestBed to create the component fixture
5. Add a test spec with one or more expectations

Step 1: Create a Test Suite

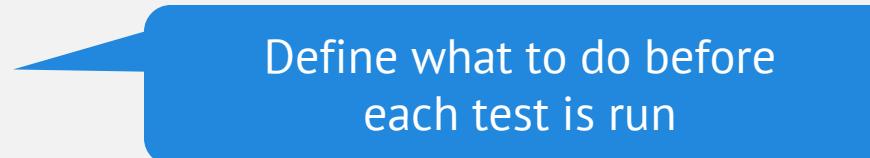
```
customers.component.spec.ts
```

```
describe('CustomersComponent Tests', () => {  
});
```

Step 2: Add a beforeEach()

customers.component.spec.ts

```
describe('CustomersComponent Tests', () => {  
  beforeEach(() => {  
    });  
});
```



Define what to do before
each test is run

Step 3: Use TestBed to Configure the Module

customers.component.spec.ts

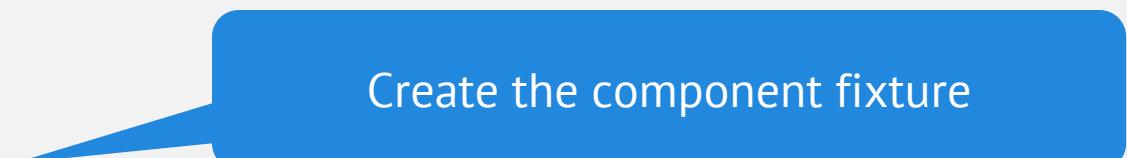
```
describe('CustomersComponent Tests', () => {  
  beforeEach(() => {  
    TestBed.configureTestingModule({  
      declarations: [ CustomersComponent ],  
      providers: [ DataService ],  
      imports: [ RouterTestingModule ]  
    });  
  });  
});
```

Configure the Angular module
for the component test

Step 4: Use TestBed to Create the Component

customers.component.spec.ts

```
describe('CustomersComponent Tests', () => {
  let fixture: ComponentFixture<CustomersComponent>;
  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [ CustomersComponent ],
      providers: [ DataService ],
      imports: [ RouterTestingModule ]
    });
    fixture = TestBed.createComponent(CustomersComponent);
    TestBed.compileComponents();
  });
});
```



Create the component fixture

Step 5: Add a Test Spec

customers.component.spec.ts

```
describe('CustomersComponent Tests', () => {  
  let fixture: ComponentFixture<CustomersComponent>;  
  beforeEach(() => {  
    TestBed.configureTestingModule({  
      ...  
    });  
    fixture = TestBed.createComponent(CustomersListComponent);  
    TestBed.compileComponents();  
  });  
  
  it('should be created', inject([DataService], (service: DataService) => {  
    expect(fixture.componentInstance).toBeTruthy();  
  }));  
});
```

Define the spec (test)

Exploring a Component Test Suite and Specs

Explore the following features:

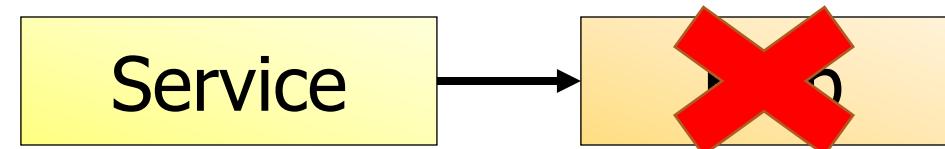
- Imports included at the top of the spec file
- Test Suite definition
- The beforeEach() code (mocking will be covered later)
- TestBed code
- The spec

[Samples/Unit-Testing/customers-orders/src/app/customers/customers.component.spec.ts](#)

Mocking Objects

Mocking Objects

- Unit tests should focus on testing specific object functionality:
 - Components
 - Services
 - Classes
 - More...
- Unit tests **SHOULD NOT** test external dependencies of an object



- Many tests will rely on "mock" or "fake" objects

Mocking Http

```
data.service.spec.ts
```

```
describe('DataService Tests', () => {
  let mockBackend: MockBackend;
  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [ DataService, MockBackend, BaseRequestOptions,
      {
        provide: Http,
        deps: [ MockBackend, BaseRequestOptions ],
        useFactory: (backend: XHRBackend, defaultOptions: BaseRequestOptions) => {
          return new Http(backend, defaultOptions);
        }
      }
    ],
    imports: [ HttpClientModule ]      }));
  mockBackend = TestBed.get(MockBackend);
});
```



Mock Http

Making Http Calls in Specs

data.service.spec.ts

```
function provideCustomersToMockBackend() {
  mockBackend.connections.subscribe((connection: MockConnection) => {
    connection.mockRespond(new Response(
      new ResponseOptions({
        body: customers
      })
    ));
  });
}

it('should get customers async',
  async(inject([DataService], (service) => {
    provideCustomersToMockBackend();
    service.getCustomers().subscribe(data) => {
      expect(data.length).toBe(4);
    });
  )));
```

Provide test data

Ensure service returns
expected data

Summary

- Unit testing helps minimize bugs and leads to higher confidence in code
- Karma simplifies the process of running tests using real browsers and/or devices
- Jasmine is a test framework that provides test suites, specs, expectations and more
- Angular services and other objects can be "mocked" as necessary in test suites

