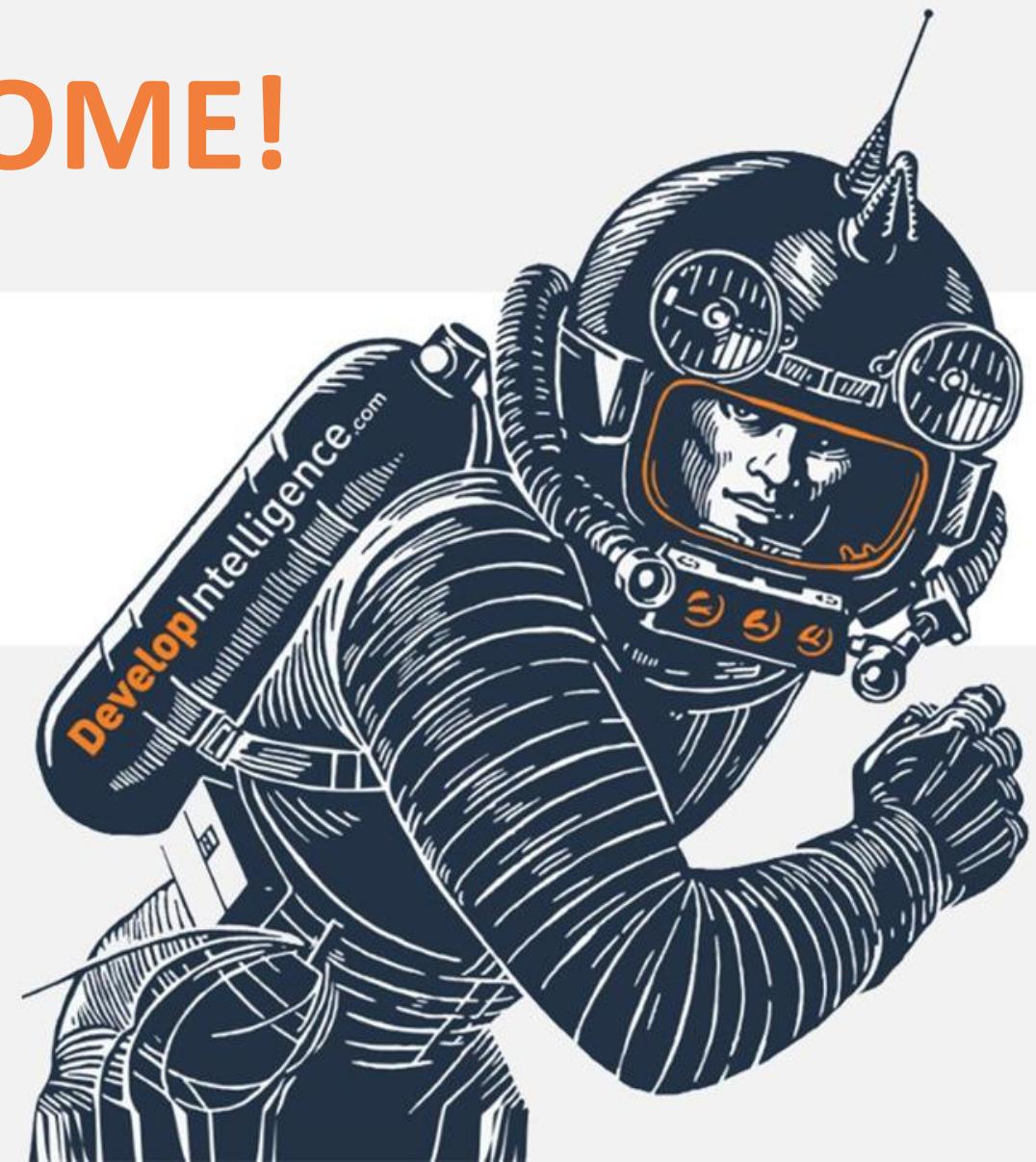


Angular Mechanics

WELCOME!



Allen R. Sanders
Senior Technology Instructor





Introduction



Angular Features



- TypeScript-based open-source web application framework
- Application structure built around a hierarchy of modules and components
- Supports SPAs (Single Page Applications) using both template-driven and reactive forms
- “Separation of concerns” architectural pattern



Angular Features



- Feature-rich application scaffolding support via Angular CLI
- Includes several types of programmatic elements for use in our apps
- Integrates well with many different types of IDEs
- Uses Node.js (for JavaScript runtime), npm (for package management) and Webpack “under the hood” (for bundling)



Semantic Versioning



- Angular versions have included AngularJS and Angular
- AngularJS (aka Angular 1) did not use semantic versioning
- Angular 2 introduced the semantic versioning concept
- Now, instead of referring to Angular 2 vs. Angular 4 vs. Angular 10, we just call it Angular



Semantic Versioning



- Built around the concept of patch, minor and major change tracking
- Correspond to 3 different types of possible change to the framework
- Format:



Changelog available at <https://github.com/angular/angular/blob/master/CHANGELOG.md>



Patch Change



- Rightmost digit in version number incremented
- Usually implies a bug fix
- When upgrading, represents relatively low risk



Minor Change

- Middle digit in version number incremented
- More significant change or new functionality added
- However, should remain backward compatible on upgrade



Major Change



- Leftmost digit in version number incremented
- Represents most significant type of change
- Potentially incompatible with previous versions



Semantic Versioning & Testing



- Whether a major, minor or patch change, it's still a change
- Changes require testing to assess regression
- Points to the need for a robust set of automated test suites
- Which we'll be discussing when we get to the unit testing topic...



Angular – Up & Running





- JavaScript runtime built on top of Chrome's v8 JS engine
- To install, navigate to <https://nodejs.org> (for Windows or macOS)
- For Linux, you can use *sudo apt-get install nodejs*
- Run *node -v* in your terminal to confirm (requires current or LTS version)

```
sysadmin@DESKTOP-QJENT2P:~$ node -v
v10.19.0
sysadmin@DESKTOP-QJENT2P:~$
```



Node Package Manager (npm)



- Software package manager included by default in Node.js
- Run *npm -v* in your terminal to confirm
- Provides scripting access to packages in a central npm registry
- “package.json” and “package-lock.json” list runtime and dev dependencies (with versions) for our Angular app

```
sysadmin@DESKTOP-QJENT2P:~$ npm -v  
6.14.10  
sysadmin@DESKTOP-QJENT2P:~$ -
```



Node Package Manager (npm)



- *npm install <packagename>[@version]* to download and install a package
- -g flag used for global install; otherwise, installs to local “node_modules” folder
- --save-dev flag used to indicate dev dependency
- *npm install* used to install all “package.json” packages



Angular CLI (ng)



- Key part of the Angular development and deployment toolset
- Used to scaffold a new Angular app
- Also used for scaffolding new Angular artifacts
- Includes commands for building, launching and deploying apps



Angular CLI (ng)



- Run *npm install -g @angular/cli* to install
- Run *ng version* to confirm
- General command format is *ng <command> [options]*
- *ng help* provides list of available commands and *ng <command> --help* provides help for a specific command

```
sysadmin@DESKTOP-QJENT2P:~$ ng version

Angular CLI: 11.0.5
Node: 10.19.0
OS: linux x64

Angular:
...
Ivy Workspace:

Package          Version
-----
@angular-devkit/architect    0.1100.5 (cli-only)
@angular-devkit/core         11.0.5 (cli-only)
@angular-devkit/schematics   11.0.5 (cli-only)
@schematics/angular          11.0.5 (cli-only)
@schematics/update           0.1100.5 (cli-only)

sysadmin@DESKTOP-QJENT2P:~$
```



Angular CLI (ng) – Common Commands



- *ng new* (or *ng n*) – Creates/scaffolds a new Angular app
- *ng build* (or *ng b*) – Compiles Angular app and outputs generated/bundled files to predefined folder
- *ng generate* (or *ng g*) – Creates/scaffolds a new Angular artifact



Angular CLI (ng) – Common Commands



- *ng serve* (or *ng s*) – Builds and serves Angular app from a preconfigured web server
- *ng test* (or *ng t*) – Runs unit tests for an Angular app
- *ng deploy* – Optimizes, bundles and deploys Angular app to web hosting provider



Angular CLI (ng) – Common Commands



- *ng add* – Installs a 3rd party library to an Angular app
- *ng update* – Updates an Angular app along with its dependencies



Visual Studio Code (IDE)





Visual Studio Code



- While not the only option, VS Code is a great IDE for Angular development
- Cross-platform and extensible
- With the right extensions installed, can greatly facilitate Angular dev
- Access <https://code.visualstudio.com/Download> to download & install



Angular Essentials Extension Pack



- Represents a pack that includes several useful extensions
- Integrates intellisense, linting and formatting into HTML, CSS and TypeScript
- Provides debugging support for multiple browsers
- Provides code snippets that can be used to “jump start” your coding

Extension Pack (9)

 Angular Snippets (Version 18) Angular version 18 snippets by John Papa John Papa	 Angular Language Service Editor services for Angular templates Angular
 ESLint Integrates ESLint JavaScript into VS Code. Microsoft	 Winter is Coming Theme Preferred dark/light themes by John Papa John Papa
 Peacock Subtly change the workspace color of your w... John Papa	 Prettier - Code formatter Code formatter using prettier Prettier



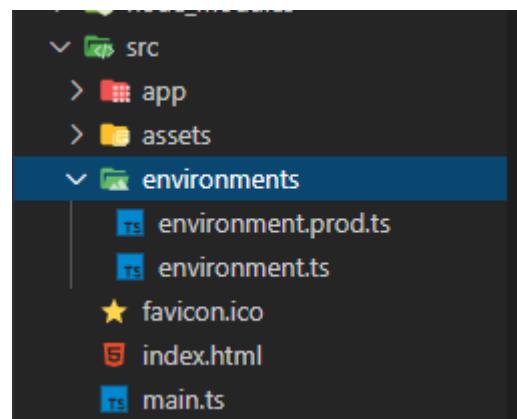
Environments



Environment-Specific Configuration



- Angular allows definition of environment-specific configuration
- At build time, a specific configuration can be applied
- Enables definition of things like back-end API URLs that vary by environment
- Defined as config objects in the “./src/environments/” directory with environment-specific file naming



```
src > environments > environment.prod.ts > environment > BASE_API_URL
1  export const environment = [
2    production: true,
3    BASE_API_URL: 'https://prod-server/api/my-api/',
4  ];
5
```



Environment-Specific Build



- To target an environment, run *ng build --configuration <envname>*
- Possible values for <envname> can be found in “./angular.json”
- Used to specify config file that corresponds to the target environment
- *ng build* will target dev (by default)

ng build --configuration production

```
  "fileReplacements": [
    {
      "replace": "src/environments/environment.ts",
      "with": "src/environments/environment.prod.ts"
    }
  ],
  "optimization": true,
  "outputHashing": "all",
  "sourceMap": false,
  "extractCss": true,
  "namedChunks": false,
  "extractLicenses": true,
  "vendorChunk": false,
```



Angular Components





Angular Components



- The building blocks for our Angular application UI
- Control different pages or different parts of a page called “views”
- As previously mentioned, components in an Angular app compose a hierarchy (called the “component tree”)
- There are facilities for communicating data to and from a component



Creating Components



- To create a new component in Angular, we use *ng generate component <name>*
- Angular will append the word “Component” to the associated class name
- Angular will generate four files (by default):
 - <name>.component.html
 - <name>.component.css
 - <name>.component.ts
 - <name>.component.spec.ts



<name>.component.ts



- The TypeScript class for the component
- Where we place our view management and processing logic
- Houses properties used to populate data placeholders in the view
- Can include input and output signals (for cross-component communication)



<name>.component.ts



- Uses `@Component()` decorator with a configuration object
- Includes a selector (the element name used to place the component)
- Includes a file reference (pointing to the HTML for the view) or inline HTML
- Includes a file reference (pointing to the CSS for the view) or inline styling



<name>.component.ts

```
order-header.component.ts X
src > app > ordering > order-header > order-header.component.ts > OrderHeaderComponent
1 import { Component, EventEmitter, Input, OnInit, Output } from '@angular/core';
2 import { DataService } from 'src/app/services/data.service';
3
4 @Component({
5   selector: 'app-order-header',
6   templateUrl: './order-header.component.html',
7   styleUrls: ['./order-header.component.css']
8 })
9 export class OrderHeaderComponent implements OnInit {
10
11   status = 'Open';
12   @Input() orderNumber: string;
13   @Output() shipped = new EventEmitter<boolean>();
14
15   constructor(private dataService: DataService) { }
16
17   ngOnInit(): void {
18   }
19
20 }
21
```



<name>.component.ts

```
order-header.component.ts X
src > app > ordering > order-header > order-header.component.ts > OrderHeaderComponent
  1 import { Component, EventEmitter, Input, OnInit, Output } from '@angular/core';
  2 import { DataService } from 'src/app/services/data.service';
  3
  4 @Component({
  5   selector: 'app-order-header',
  6   template: `
  7     <div>
  8       <p>Order number is {{orderNumber}}</p>
  9       <p>Order status is {{status}}</p>
 10       <button (click)="shipped.emit(true)">Ship It</button>
 11     </div>
 12   `,
 13   styles: [
 14     `p:first-child {
 15       color: red;
 16     }`
 17   ]
 18 })
 19 export class OrderHeaderComponent implements OnInit {
 20
 21   status = 'Open';
 22   @Input() orderNumber: string;
 23   @Output() shipped = new EventEmitter<boolean>();
 24
 25   constructor(private dataService: DataService) { }
 26
 27   ngOnInit(): void {
 28   }
 29
 30 }
 31
```



Creating Components



- By default, *ng generate component <name>* creates a folder for the component and component files in the path where run
- Alternatively, specify a target path during generation (e.g., *ng generate component ordering/orderDetail*)
- To use the component, it must be registered with a module



Creating Components



- If target path includes a module, the component will be registered with that module automatically
- Generating in the root folder of the app will register with *AppModule*
- Generating in a folder that does not include a module will register with *AppModule* (by default)



Creating Components



- *ng generate component <name> --module <module-name>* will target the named module for registration
- Angular converts camel-casing to kebab-casing (filenames and selectors) and Pascal-casing for class name
- Remember – include in “exports” of module to make component available outside of module



Displaying Component Data



- Interpolated expressions can be used to display component property values
- For example, `<p>Order number is {{orderNumber}}</p>`
- Alternatively, we can use property binding syntax (square brackets)



Displaying Component Data



- `<button (click)="ship.emit(true)" [innerText]="buttonText"></button>`
- “buttonText” in this example is a public property of the component (not a string literal)
- To set using a string literal, use single quotes within the double quotes -
`[innerText]=""Click Me!""`



Displaying Component Data



- We can also use component property binding with attributes of elements
- To do so, we use *attr.* syntax in the binding
- For example:

```
<button (click)="ship.emit(true)" [innerText]="buttonText" [attr.value] = "buttonText"></button>
```



One-Way Data Binding



- `<p>Order number is {{orderNumber}}</p>`
- `<button (click)="ship.emit(true)" [innerText]="buttonText"></button>`
- Creates a binding in the view – updates to the variables are automatically reflected in the display



One-Way Data Binding



- This binding is one-way though
- User input from the view (e.g., entering data in an `<input>` form field) is not persisted to the component variable

```
one-way.component.ts
src > app > one-way > one-way.component.ts > OneWayComponent > theData
1 import { Component, Input, OnInit } from '@angular/core';
2
3 @Component({
4   selector: 'app-one-way',
5   templateUrl: './one-way.component.html',
6   styleUrls: ['./one-way.component.css']
7 })
8 export class OneWayComponent implements OnInit {
9
10   @Input() theData: string;
11
12   constructor() { }
13
14   ngOnInit(): void {
15   }
16
17   showUpdatedData(): void {
18     console.log(this.theData);
19   }
20
21
22
```

The screenshot shows a browser window with two tabs: 'one-way.component.ts' and 'one-way.component.html'. The 'one-way.component.html' tab displays the following code:

```
src > app > one-way > one-way.component.html > ...
1 <p>Data input is {{theData}}</p>
2 <div>
3   <label>Enter the Data:</label>
4   <input type="text" [value]="theData" />
5 </div>
6 <button (click)="showUpdatedData()">Update</button>
7
```

The browser's DOM inspector shows the rendered HTML. Below the browser window, the developer tools' Console tab is open, displaying the message: "Angular is running in development mode. Call enableProdMode() to enable production mode." and "[WDS] Live Reloading enabled.".



Two-Way Data Binding

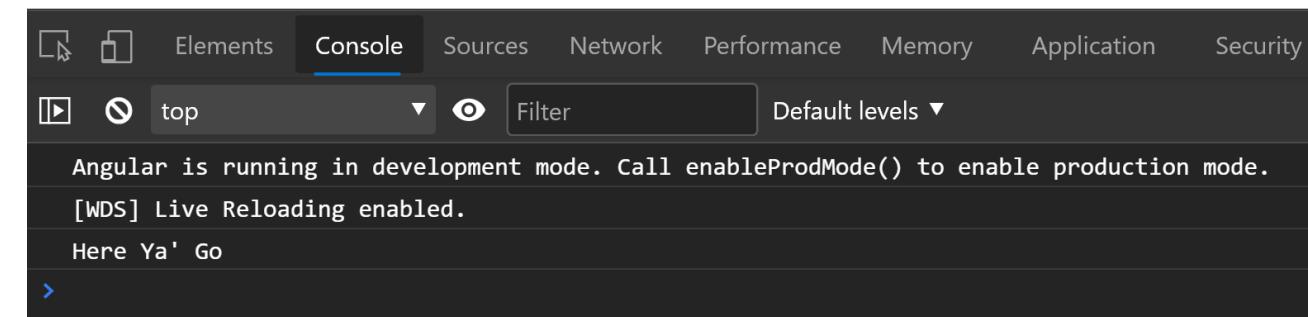


- Can make the binding two-way using template-driven forms and *ngModel*
- *ngModel* is a directive available upon import of *FormsModule*
- Add `[(ngModel)]="<var-name>"` to an input element to enable

```
src > app > one-way > one-way.component.html > ...
1  <p>Data input is {{theData}}</p>
2  <div>
3    <label>Enter the Data:</label>
4    <input type="text" [(ngModel)]="theData" />
5  </div>
6  <button (click)="showUpdatedData()">Update</button>
7
```

Data input is Here Ya' Go

Enter the Data: Update





Template Reference Variables



- Named identifiers that can be added to elements in a component's template
- Prefixed by a hash symbol (#)
- Once assigned to an element, can use that name to reference the element in another part of the view
- Also, can grab a reference to the element in code using the name for programmatic interaction

The screenshot shows a code editor window with a dark theme. The file is named 'one-way.component.html'. The code contains several template reference variables, specifically '#theInput' used as both a selector and a value binding.

```
src > app > one-way > one-way.component.html > ...
1   <p>Data input is {{theData}}</p>
2   <p>Also {{theInput.value}}</p>
3   <div>
4     <label>Enter the Data:</label>
5     <input type="text" [(ngModel)]="theData" #theInput />
6   </div>
7   <button (click)="showUpdatedData()">Update</button>
8 
```



Applying Styles to Component Template



- The template can use class binding or style binding for applying styles
- Class binding supports the `[class.<class-name>]=“<boolean expr>”` syntax
- For example, `...`
- If boolean expression evaluates to true, the `open` class will be applied to the element



Applying Styles to Component Template



- Class binding also supports alternate syntax
- For example, `<p [class] = "targetClasses">...</p>`
- *targetClasses* can be a property of the component and will be one of:
 - A space-delimited string of class names
 - An object with class names for keys and boolean expressions for values – a true value applies the class defined by the key



Applying Styles to Component Template



Component

```
order-header.component.ts X
src > app > ordering > order-header > order-header.component.ts > OrderHeaderComponent > targetClasses
1 import { Component, EventEmitter, Input, OnInit, Output } from '@angular/core';
2 import { DataService } from 'src/app/services/data.service';
3
4 @Component({
5   selector: 'app-order-header',
6   templateUrl: './order-header.component.html',
7   styleUrls: ['./order-header.component.css']
8 })
9 export class OrderHeaderComponent implements OnInit {
10
11   status = 'Open';
12   @Input() orderNumber: string;
13   @Output() shipped = new EventEmitter<boolean>();
14   buttonText = 'Ship it NOW!!';
15   targetClasses = 'open safe';
16
17   constructor(private dataService: DataService) { }
18
19   ngOnInit(): void {
20   }
21
22 }
23
```

Template

```
order-header.component.html X
src > app > ordering > order-header > order-header.component.html > ...
1 <div>
2   <p [class]="targetClasses">Order Number is {{orderNumber}}</p>
3   <p>Order Status is {{status}}</p>
4   <button (click)="shipped.emit(true)" [innerText]="buttonText" [attr.value]="buttonText"></button>
5 </div>
6
```



Applying Styles to Component Template



Component

```
order-header.component.ts X
src > app > ordering > order-header > order-header.component.ts > OrderHeaderComponent > targetClasses
1 import { Component, EventEmitter, Input, OnInit, Output } from '@angular/core';
2 import { DataService } from 'src/app/services/data.service';
3
4 @Component({
5   selector: 'app-order-header',
6   templateUrl: './order-header.component.html',
7   styleUrls: ['./order-header.component.css']
8 })
9 export class OrderHeaderComponent implements OnInit {
10
11   status = 'Open';
12   @Input() orderNumber: string;
13   @Output() shipped = new EventEmitter<boolean>();
14   buttonText = 'Ship it NOW!!';
15
16   targetClasses = {
17     open: this.status === 'Open',
18     safe: true
19   };
20
21   constructor(private dataService: DataService) { }
22
23   ngOnInit(): void {
24
25 }

}
```

Template

```
order-header.component.html X
src > app > ordering > order-header > order-header.component.html > ...
1 <div>
2   <p [class]="targetClasses">Order Number is {{orderNumber}}</p>
3   <p>Order Status is {{status}}</p>
4   <button (click)="shipped.emit(true)" [innerText]="buttonText" [attr.value]="buttonText"></button>
5 </div>
6
```



Applying Styles to Component Template



- Style binding supports the `[style.<style-property>] = "<value>"` syntax
- For example, `<p [style.color]="colorSelection">...</p>`
- Also, supports sub-levels of definition `[style.width.px] = "widthValue"`



Applying Styles to Component Template



- Like class binding, style binding supports alternate syntax as well
- For example, `<p [style] = "targetStyle">...</p>`
- `targetStyle` can be a property of the component and will be one of:
 - Style settings separated by a semicolon
 - An object with style properties for keys and style values for values



Applying Styles to Component Template



Component

```
order-header.component.ts X
src > app > ordering > order-header > order-header.component.ts > OrderHeaderComponent > targetStyles
1 import { Component, EventEmitter, Input, OnInit, Output } from '@angular/core';
2 import { DataService } from 'src/app/services/data.service';
3
4 @Component({
5   selector: 'app-order-header',
6   templateUrl: './order-header.component.html',
7   styleUrls: ['./order-header.component.css']
8 })
9 export class OrderHeaderComponent implements OnInit {
10
11   status = 'Open';
12   @Input() orderNumber: string;
13   @Output() shipped = new EventEmitter<boolean>();
14   buttonText = 'Ship it NOW!';
15   targetClasses = {
16     open: this.status === 'Open',
17     safe: true
18   };
19   targetStyles = 'color: darkred; width: 75px;';
20
21   constructor(private dataService: DataService) { }
22
23   ngOnInit(): void {
24   }
25
26 }
27
```

Template

```
order-header.component.html X
src > app > ordering > order-header > order-header.component.html > div > p
1 <div>
2   <p [class]="targetClasses">Order Number is {{orderNumber}}</p>
3   <p [style]="targetStyles">Order Status is {{status}}</p>
4   <button (click)="shipped.emit(true)" [innerText]="buttonText" [attr.value]="buttonText"></button>
5 </div>
6
```



Applying Styles to Component Template



Component

```
order-header.component.ts X
src > app > ordering > order-header > order-header.component.ts > OrderHeaderComponent > targetStyles
1 import { Component, EventEmitter, Input, OnInit, Output } from '@angular/core';
2 import { DataService } from 'src/app/services/data.service';
3
4 @Component({
5   selector: 'app-order-header',
6   templateUrl: './order-header.component.html',
7   styleUrls: ['./order-header.component.css']
8 })
9 export class OrderHeaderComponent implements OnInit {
10
11   status = 'Open';
12   @Input() orderNumber: string;
13   @Output() shipped = new EventEmitter<boolean>();
14   buttonText = 'Ship it NOW!';
15   targetClasses = {
16     open: this.status === 'Open',
17     safe: true
18   };
19   targetStyles = [
20     color: 'darkred',
21     width: '75px'
22   ];
23
24   constructor(private dataService: DataService) { }
25
26   ngOnInit(): void {
27   }
28
29 }
```

Template

```
order-header.component.html X
src > app > ordering > order-header > order-header.component.html > div > p
1 <div>
2   <p [class]="targetClasses">Order Number is {{orderNumber}}</p>
3   <p [style]="targetStyles">Order Status is {{status}}</p>
4   <button (click)="shipped.emit(true)" [innerText]="buttonText" [attr.value]="buttonText"></button>
5 </div>
6
```



Sandboxing CSS Styles



- On browsers that support Shadow DOM, Angular will scope CSS styles local to the component (by default)
- Values assigned to the “styles” or “styleUrls” property of the `@Component()` decorator will be “sandboxed” to the component
- An additional property of the `@Component()` decorator (“encapsulation”) can be used to change



Sandboxing CSS Styles



- The “encapsulation” property can be set to one of the *ViewEncapsulation* enumeration values:
 - *Emulated* – the default; emulates Shadow DOM by preprocessing and renaming CSS in the rendered HTML
 - *ShadowDOM* – use the browser’s native Shadow DOM implementation (requires a browser that supports)
 - *None* – no view encapsulation; the component’s CSS gets added to global style
- *Emulated* (the default) is recommended



Component Communication





Sharing Data



- Angular offers multiple options for sharing data between components
- Common use cases include:
 - Parent component to child component
 - Child component to parent component
 - Communication between sibling components



Parent to Child



- We can pass data into a component from a parent component host
- Called input binding and is accomplished using the `@Input()` decorator
- In its simplest form, `@Input() <prop-name>: <prop-type>;`
- For example, `@Input() orderNumber: string;`



Parent to Child



- The `@Input()` decorator accepts an optional parameter for alternate name
- Can also break out the associated property into a getter and a setter (for implementing additional logic on read vs. update)
- From the parent component, bind the child component property to a property of the parent (or a literal value)



Parent to Child



Component

```
order-header.component.ts X
src > app > ordering > order-header > order-header.component.ts > ...
1 import { Component, EventEmitter, Input, OnInit, Output } from '@angular/core';
2 import { DataService } from 'src/app/services/data.service';
3
4 @Component({
5   selector: 'app-order-header',
6   templateUrl: './order-header.component.html',
7   styleUrls: ['./order-header.component.css']
8 })
9 export class OrderHeaderComponent implements OnInit {
10
11   status = 'Open';
12   @Input() orderNumber: string;
13   @Output() shipped = new EventEmitter<boolean>();
14   buttonText = 'Ship it NOW!!';
15   targetClasses = {
16     open: this.status === 'Open',
17     safe: true
18   };
19   targetStyles = 'color: darkred; width: 75px;';
20
21   constructor(private dataService: DataService) { }
22
23   ngOnInit(): void {
24
25
26 }
```

Parent Component Template

```
app.component.html X
src > app > app.component.html > app-order-header
1 <app-order-header [orderNumber]="orderNumber" (shipped)="onShip($event)"></app-order-header>
2
```

or

```
app.component.html X
src > app > app.component.html > app-order-header
1 <app-order-header [orderNumber]="'ABC-123'" (shipped)="onShip($event)"></app-order-header>
2
```



- We can push data from a child component to its host (or parent) component
- Called output binding and is accomplished using the `@Output()` decorator
- General syntax is `@Output() <event-name> = new EventEmitter();`
- For example, `@Output() shipped = new EventEmitter();`



Child to Parent



- The *EventEmitter* class supports generics for targeting a specific data type
- By using, indicates the type of data expected for push on event occurrence
- From the child component, raise the event using *emit* (including any data to be communicated along with its occurrence)



- From the parent component, bind the event to a handler method
- To process additional data accompanying the event, use `$event` as an argument in the handler
- The data can be utilized according to its defined type



Child to Parent



Child Component

```
order-header.component.ts X
src > app > ordering > order-header > order-header.component.ts > ...
1 import { Component, EventEmitter, Input, OnInit, Output } from '@angular/core';
2 import { DataService } from 'src/app/services/data.service';
3
4 @Component({
5   selector: 'app-order-header',
6   templateUrl: './order-header.component.html',
7   styleUrls: ['./order-header.component.css']
8 })
9 export class OrderHeaderComponent implements OnInit {
10
11   status = 'Open';
12   @Input() orderNumber: string;
13   @Output() shipped = new EventEmitter<boolean>();
14   buttonText = 'Ship it NOW!';
15   targetClasses = {
16     open: this.status === 'Open',
17     safe: true
18   };
19   targetStyles = 'color: darkred; width: 75px;';
20
21   constructor(private dataService: DataService) { }
22
23   ngOnInit(): void {
24
25   }
26 }
```

Child Component Template

```
order-header.component.html X
src > app > ordering > order-header > order-header.component.html > div > p
1 <div>
2   <p [class]="targetClasses">Order Number is {{orderNumber}}</p>
3   <p [style]="targetStyles">Order Status is {{status}}</p>
4   <button (click)="shipped.emit(true)" [innerText]=buttonText [attr.value]=buttonText></button>
5 </div>
6
```



Child to Parent



Parent Component

```
app.component.ts X
src > app > app.component.ts > ...
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-root',
5   templateUrl: './app.component.html',
6   styleUrls: ['./app.component.css']
7 })
8 export class AppComponent {
9
10   orderNumber = 'DEF-456';
11   title = 'first-app';
12
13   onShip(isShipped: boolean): void {
14     if (isShipped) { console.log('Shipped!'); }
15   }
16 }
17
```

Parent Component Template

```
app.component.html X
src > app > app.component.html > app-order-header
1 <app-order-header [orderNumber]="orderNumber" (shipped)="onShip($event)"></app-order-header>
2
```



Between Siblings



- By siblings, we mean two or more child components hosted by same parent
- One option is a service for transferring data using *BehaviorSubject*
- Enables a publish/subscribe model for moving information between the components



- Create a service and inject into the constructor of each sibling targeted for data sharing
- In the service, create and initialize a private *BehaviorSubject*
- Create a public variable with an observable “wrapper” around the *BehaviorSubject*
- Can then code a public API to enable consumers to push new data through the service (using the *next()* method of the observable)



- One benefit of this approach is looser coupling between the components
- We'll discuss services, injection and observables further in later sections

```
A transfer.service.ts ×  
src > app > behavior-subject > A transfer.service.ts > ...  
1 import { Injectable } from '@angular/core';  
2 import { BehaviorSubject } from 'rxjs';  
3  
4 @Injectable({  
5   providedIn: 'root'  
6 })  
7 export class TransferService {  
8  
9   private transferAgent = new BehaviorSubject<{ valueOne: string, valueTwo: string }>({ valueOne: '', valueTwo: '' });  
10  transferAgent$ = this.transferAgent.asObservable();  
11  
12  constructor() {}  
13  
14  pushData(data: { valueOne: string, valueTwo: string }): void {  
15    this.transferAgent.next(data);  
16  }  
17}  
18
```



- For the component receiving the data, subscribe to the observable
- For the component sending the data, call the service's API to initiate



Between Siblings



“Sending” Component

```
child2.component.ts X
src > app > behavior-subject > child2 > child2.component.ts > Child2Component > dataToTransfer

1 import { Component, OnInit } from '@angular/core';
2 import { TransferService } from '../transfer.service';
3
4 @Component({
5   selector: 'app-child2',
6   templateUrl: './child2.component.html',
7   styleUrls: ['./child2.component.css']
8 })
9 export class Child2Component implements OnInit {
10
11   dataToTransfer = { valueOne: '', valueTwo: '' };
12
13   constructor(private transferService: TransferService) { }
14
15   ngOnInit(): void {
16   }
17
18   sendData(): void {
19     this.transferService.pushData(this.dataToTransfer);
20   }
21
22 }
23
```

“Sending” Component Template

```
child2.component.html X
src > app > behavior-subject > child2 > child2.component.html > ...
1 <div>
2   <input type="text" [(ngModel)]="dataToTransfer.valueOne" />
3   <input type="text" [(ngModel)]="dataToTransfer.valueTwo" />
4   <button (click)="sendData()">Send Data</button>
5 </div>
6
```



Between Siblings



“Receiving” Component

```
child1.component.ts ×  
src > app > behavior-subject > child1 > child1.component.ts > ...  
1  import { Component, OnDestroy, OnInit } from '@angular/core';  
2  import { Subscription } from 'rxjs';  
3  import { TransferService } from '../transfer.service';  
4  
5  @Component({  
6    selector: 'app-child1',  
7    templateUrl: './child1.component.html',  
8    styleUrls: ['./child1.component.css']  
9  })  
10 export class Child1Component implements OnInit, OnDestroy {  
11  
12  private subscription: Subscription;  
13  transferredData: { valueOne: string, valueTwo: string };  
14  
15  constructor(private transferService: TransferService) {}  
16  
17  ngOnInit(): void {  
18    this.subscription = this.transferService.transferAgent$.subscribe(data => {  
19      | this.transferredData = { ...data };  
20    });  
21  }  
22  
23  ngOnDestroy(): void {  
24    if (this.subscription) {  
25      | this.subscription.unsubscribe();  
26    }  
27  }  
28  
29}  
30 |
```

“Receiving” Component Template

```
child1.component.html ×  
src > app > behavior-subject > child1 > child1.component.html > ...  
1  <div>  
2    | <p>First Value: {{ transferredData.valueOne }}</p>  
3    | <p>Second Value: {{ transferredData.valueTwo }}</p>  
4  </div>  
5
```



@ViewChild() Decorator



- Enables query selection against the DOM of a component's view
- Supports referencing by:
 - Component or directive type
 - Template reference variable (as a string)
 - Provider type (defined in child component tree of the component)
 - String token



@ViewChild() Decorator



- Accepts a parameter for target selector (based on type)
- Also accepts an optional parameter for a set of options:
 - *read* – Used to specify a different token for retrieval from the query results
 - *static* – Used to dictate whether query results should be resolved once before change detection (true) or after every change detection (false – the default)



@ViewChild() Decorator



- Using acquired reference, can update the queried element in code (set styles, text content, etc.)
- Can also call methods on the queried element as needed
- For initialization, `@ViewChild()` references are available in `ngAfterViewInit()` vs. `ngOnInit()`
- Query only has access to the component's template – cannot be utilized for referencing elements in its children or parent



@ContentChild() Decorator



- Enables query selection against the DOM of a component's projected content
- Using `<ng-content>` elements, can create placeholders to which content can be projected by the component's parent
- This projected content will not be accessible using `@ViewChild()`



@ContentChild() Decorator



- Accepts same parameter set used with `@ViewChild()`
- Accepts a parameter for target selector (based on type)
- Also accepts an optional parameter for a set of options:
 - `read` – Used to specify a different token for retrieval from the query results
 - `static` – Used to dictate whether query results should be resolved once before change detection (`true`) or after every change detection (`false` – the default)



@ContentChild() Decorator



- Similar capabilities as with `@ViewChild()`, but operates against projected content
- Using acquired reference, can update the queried element in code (set styles, text content, etc.)
- Can also call methods on the queried element as needed



@ContentChild() Decorator



- Target lifecycle hook for initialization should be `ngAfterContentInit()`
- As with `@ViewChild()`, query only has access to the component's content – not its children or parent



@ViewChild() and @ContentChild()



- When building dynamically-generated views from defined templates, programmatic access can be a powerful tool
- Also, enables additional capabilities at runtime vs. only interacting with statically-defined content
- Both support collection-based versions as well - `@ViewChildren()` and `@ContentChildren()`
- Return `QueryList` collections of matching elements



Component Lifecycle





Lifecycle Hooks



- Angular components follow a specific set of lifecycle stages
- Stages include various layers of initialization, change detection and teardown
- Optional “hooks” are provided in a set of interfaces that can be implemented in a component for custom logic



Lifecycle Hooks



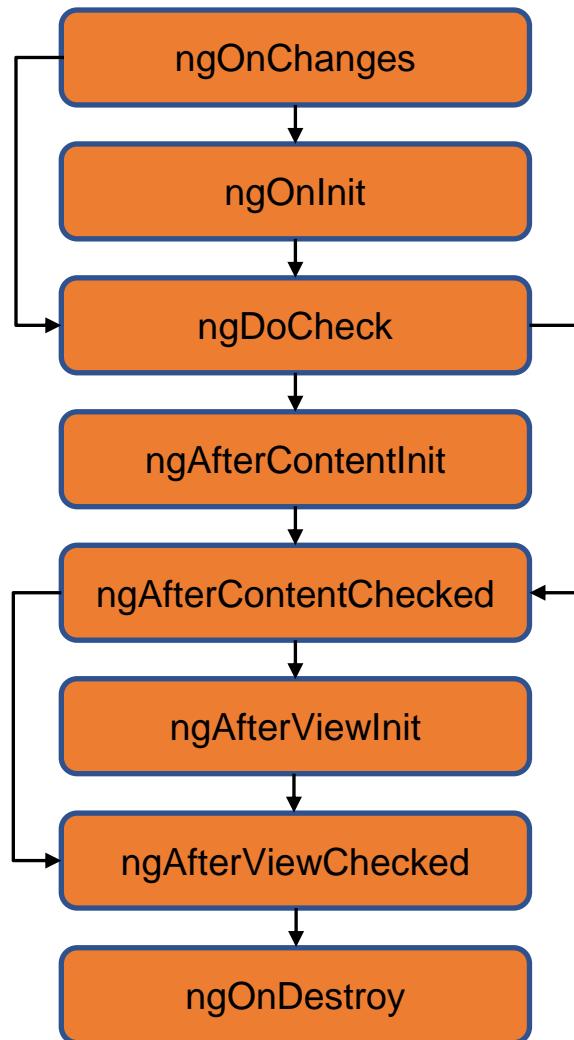
- To utilize, we can implement one (or more) of the interfaces and code the associated method
- Helpful for implementing custom logic on occurrence of the lifecycle event
- For example, setting up a subscription to an observable on initialization and unsubscribing on component teardown



- The interface and its associated lifecycle hook are named similarly - `<Interface Name>` and `ng<InterfaceName>()`
- For example, `OnInit` and `ngOnInit()`
- Technically, the interface does not need to be implemented – the presence of the method is enough
- However, best practice dictates that you should include both

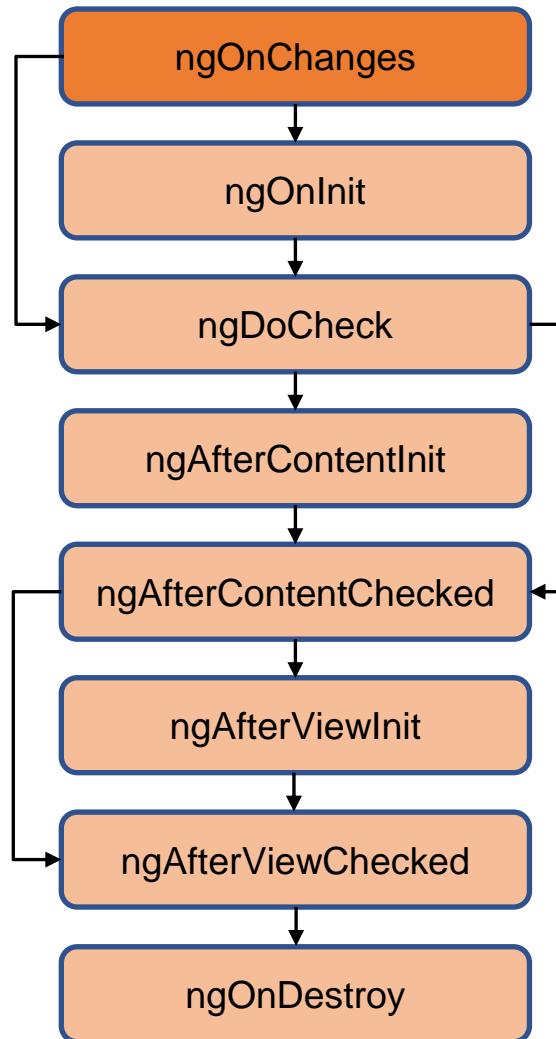


Lifecycle Hooks





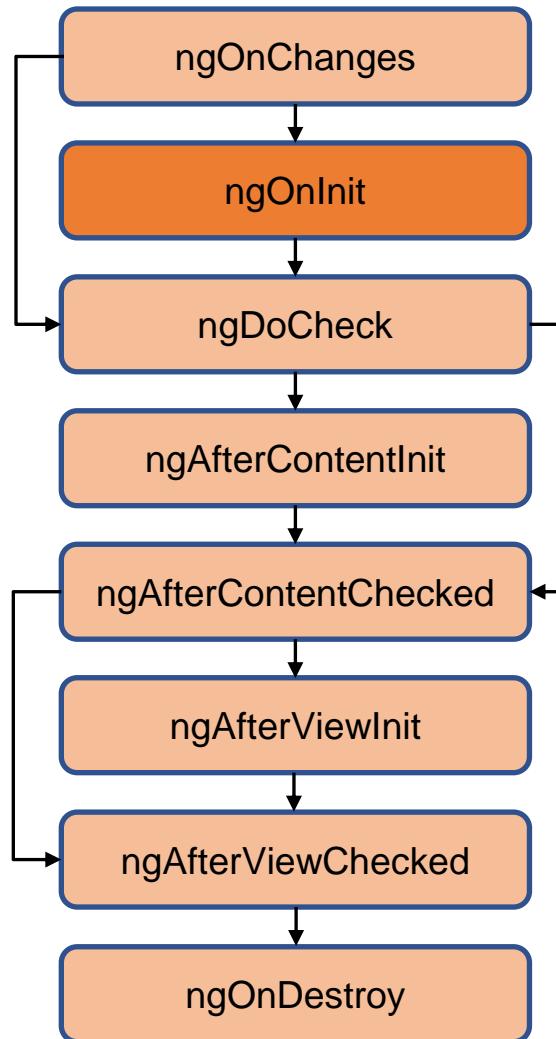
Lifecycle Hooks - ngOnChanges



- Called once before *ngOnInit* and then on every change to one or more data bound input properties
- Receives a *SimpleChanges* object containing a keyed entry for each changed property (by property name)
- Entry includes access to previous value and current value for property
- Also, includes an *isFirstChange()* method



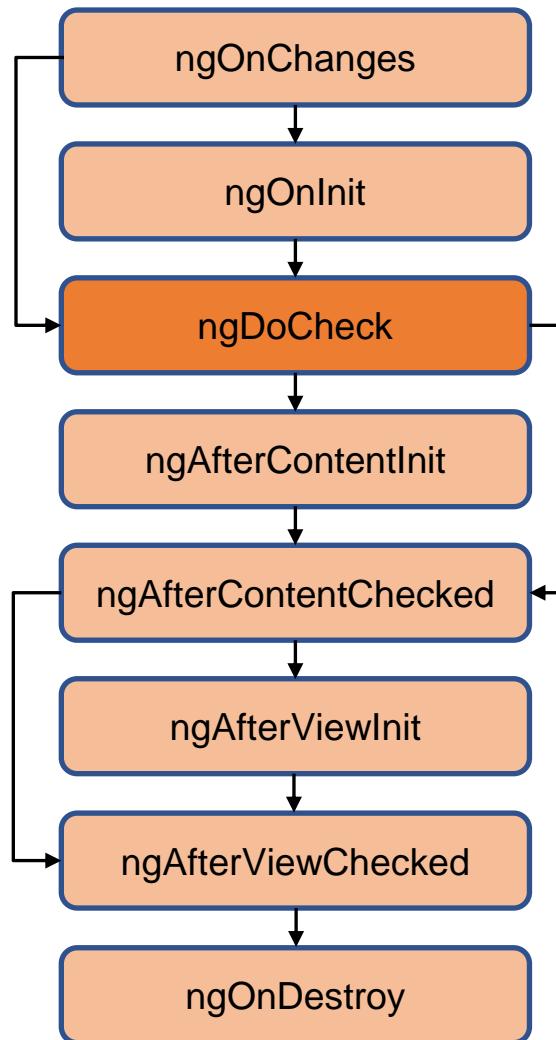
Lifecycle Hooks - ngOnInit



- Called once after first `ngOnChanges` call
- Captures initialization logic after set and first display of data bound properties
- Often used to pull initial data for component from service
- Enables initialization logic using data bound input property values (access not available in constructor)



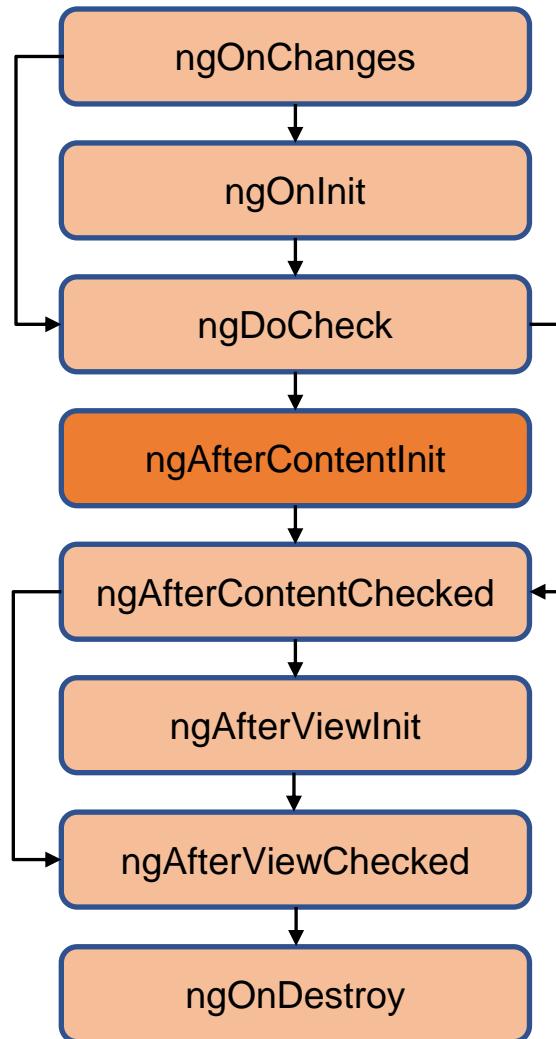
Lifecycle Hooks - ngDoCheck



- Called immediately after `ngOnInit` on component initialization
- Also, called immediately after `ngOnChanges` on every subsequent change detection run
- Enables customized change detection and response
- Best practice dictates that a component should not implement both `ngOnChanges` and `ngDoCheck`



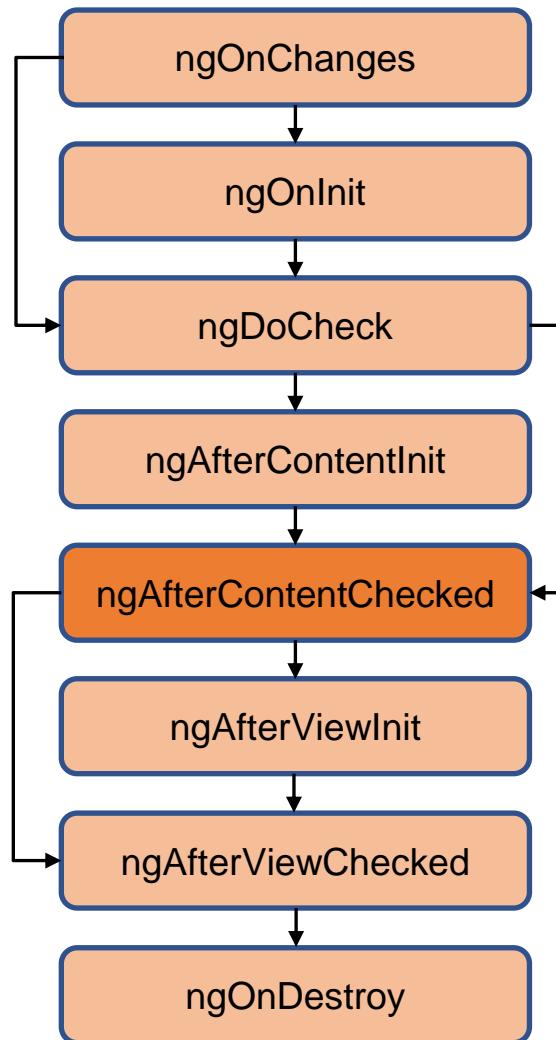
Lifecycle Hooks - ngAfterContentInit



- Called once after first *ngDoCheck()*
- Enables initialization logic against projected content
- Mentioned during *@ContentChild()* review



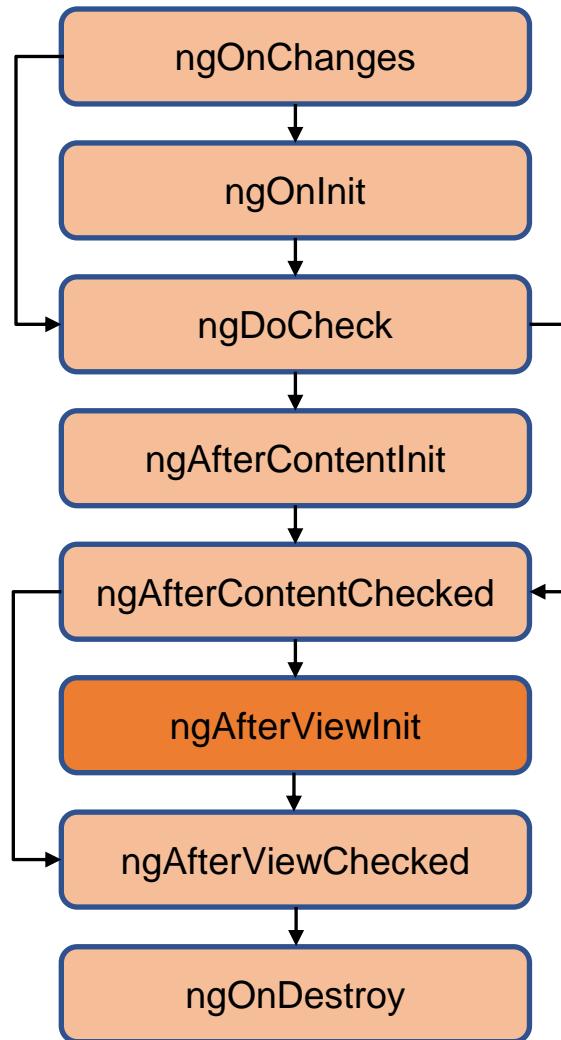
Lifecycle Hooks - ngAfterContentChecked



- Called after *ngAfterContentInit()*
- Also called after every subsequent *ngDoCheck()*
- Enables custom logic in response to projected content changes



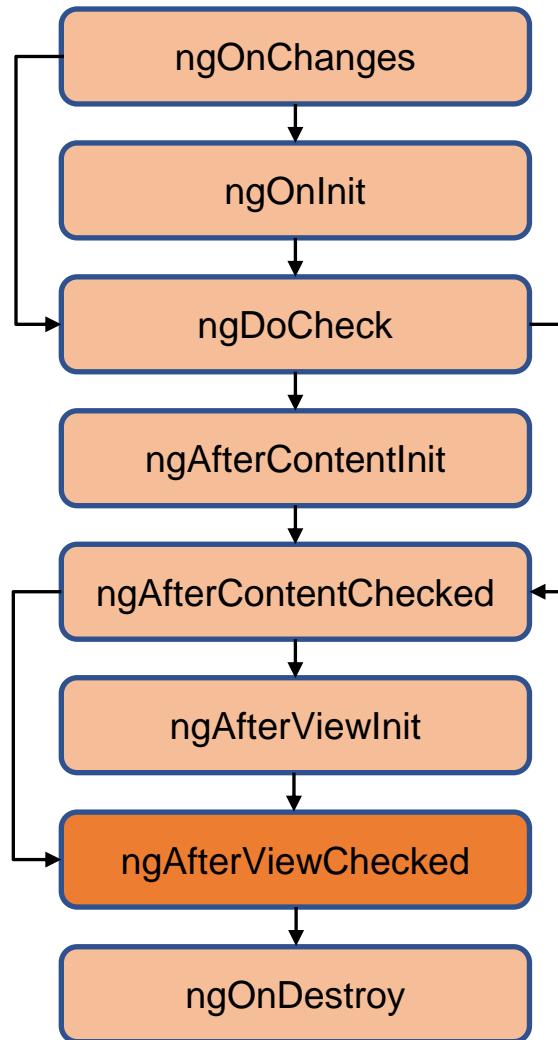
Lifecycle Hooks - ngAfterViewInit



- Called once after first `ngAfterContentChecked()`
- Enables initialization logic after a component's views and child views are setup and ready
- Mentioned during `@ViewChild()` review



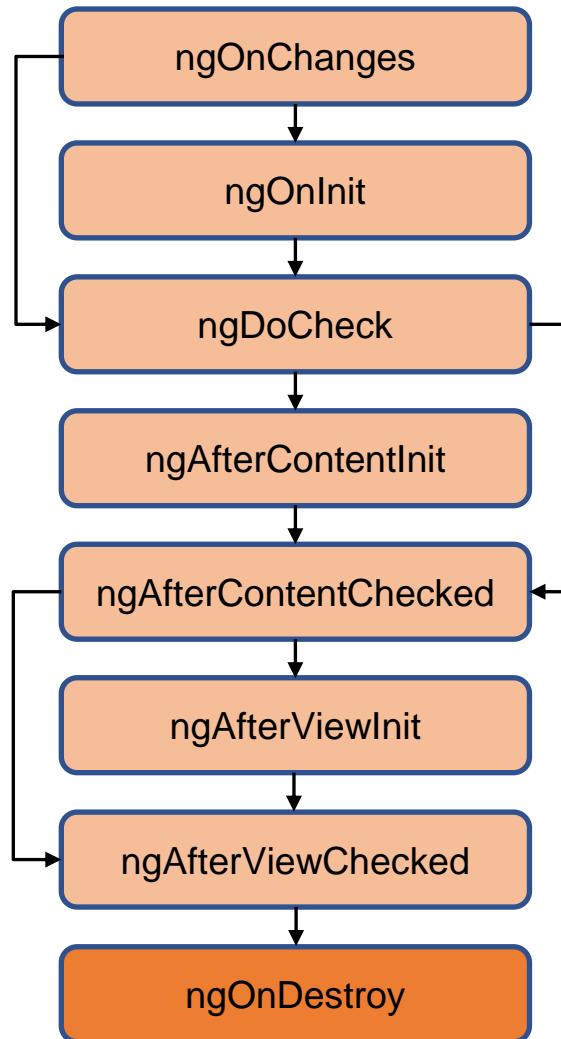
Lifecycle Hooks - ngAfterViewChecked



- Called after *ngAfterViewInit()*
- Also called after every subsequent *ngAfterContentChecked()*
- Enables custom logic in response to changes in the component's view and child views



Lifecycle Hooks - ngOnDestroy



- Called immediately before component is destroyed
- Used for cleanup logic – e.g., unsubscribing from observables, detaching event handlers, etc.
- Important for helping to prevent memory leaks



Component Change Detection





Change Detection Strategies



- Angular's change detection is critical to its ability to serve a dynamic app
- As changes occur (both user-generated and app-generated), application views update to reflect new content and values
- Non-deterministic process triggered in response to specific events



Change Detection Strategies



- Every component has a change detector associated to it
- To detect and apply changes, Angular compares the current value of a component property to the previous value
- If different, the component view is updated accordingly



Change Detection Strategies



- As discussed in the section on component lifecycle, there are hooks in which we can code custom logic to be executed on changes
- However, care should be taken as to amount and complexity of logic
- Otherwise, can result in a performance impact (given that changes likely occur many times)



- Can also see performance impact with applications that include many component instances (e.g., large lists of items)
- With a component that includes object references, change detection will trigger on updates to the object's properties (or sub-properties)
- Depending on the complexity of the component tree, may need to manage differently to optimize user experience



ChangeDetectionStrategy



- For those cases, a different change detection strategy can be set for the component
- The `@Component()` decorator's object parameter includes an optional *changeDetection* designation
- Type of this property is *ChangeDetectionStrategy* (an enum)



- *ChangeDetectionStrategy* supports two possible values:
 - *Default* (which is the default strategy)
 - *OnPush*
- When set to *Default* (or left unset), Angular will leverage standard change detection



- When set to *OnPush*, change detection is triggered only when references for `@Input()` properties change
- This can improve performance but requires some modifications to reflect change in our views
- One option is to use the spread operator (...) for managing updates to objects and arrays in an immutable manner



Spread Operator (...)



- For arrays, instead of calling `push()` to add a new element, create a new array from the existing one

```
this.values = [...this.values, 'Added Value'];
```

- Using this approach, the reference will be updated (triggering `OnPush` change detection)



Spread Operator (...)

- For objects, instead of modifying properties of an existing instance, create a new instance

```
this.objectRef = { ...this.objectRef, prop1: 'update' };
```

-or-

```
this.objectRef = { ...this.objectRef };
```

```
this.objectRef.prop1 = 'Update';
```

- As with arrays, the reference will be updated (triggering *OnPush* change detection)



ChangeDetectorRef



- Alternatively, can use *ChangeDetectorRef* to manage change detection
- Includes the following methods:
 - *markForCheck()* – marks the view (and its ancestors up to root) as changed for recheck (at some point in future)
 - *detach()* – detaches the view from change-detection
 - *detectChanges()* – explicitly triggers change detection for the view and its children
 - *reattach()* – reattaches a detached view



Angular Directives





Angular Directives



- Custom HTML attribute applied to another HTML element
- Used when we want to customize HTML layout or how an element behaves or looks
- Angular provides a set of built-in directives
- However, we can create our own custom directives as well



Types of Directives



- There are three types of directives:
 - Components – Directives with a template
 - Structural Directives – Used to add or remove elements from the DOM
 - Attribute Directives – Used to modify appearance or add custom behavior to a DOM element



Built-In Directives

- Include the following:
 - *ngIf* (structural)
 - *ngFor* (structural)
 - *ngSwitch* (structural)
 - *ngClass* and *ngStyle* (attribute)
- Let's take a closer look at each one

- Used to conditionally add or remove an HTML element in the DOM
- Uses a boolean expression to manage
- If expression evaluates to true, element is inserted into the DOM
- If false, the element is removed
- Does not simply manage visibility of element but actual presence in tree

Component

```
order-header.component.ts X
src > app > ordering > order-header > order-header.component.ts > ...
1  import { Component, EventEmitter, Input, OnInit, Output } from '@angular/core';
2  import { DataService } from 'src/app/services/data.service';
3
4  @Component({
5    selector: 'app-order-header',
6    templateUrl: './order-header.component.html',
7    styleUrls: ['./order-header.component.css']
8  })
9  export class OrderHeaderComponent implements OnInit {
10
11    status = 'Open';
12    @Input() orderNumber: string;
13    @Output() shipped = new EventEmitter<boolean>();
14    buttonText = 'Ship it NOW!';
15    targetClasses = {
16      open: this.status === 'Open',
17      safe: true
18    };
19    targetStyles = 'color: darkred; width: 75px;';
20
21    constructor(private dataService: DataService) { }
22
23    ngOnInit(): void {
24
25
26  }
```

Component Template

```
order-header.component.html X
src > app > ordering > order-header > order-header.component.html > div > p
1  <div>
2    <p [class]="targetClasses">Order Number is {{orderNumber}}</p>
3    <p *ngIf="status === 'Open'" [style]="targetStyles">Order Status is {{status}}</p>
4    <button (click)="shipped.emit(true)" [innerText]="buttonText" [attr.value]="buttonText"></button>
5  </div>
6
```

DOM When Status is “Open”

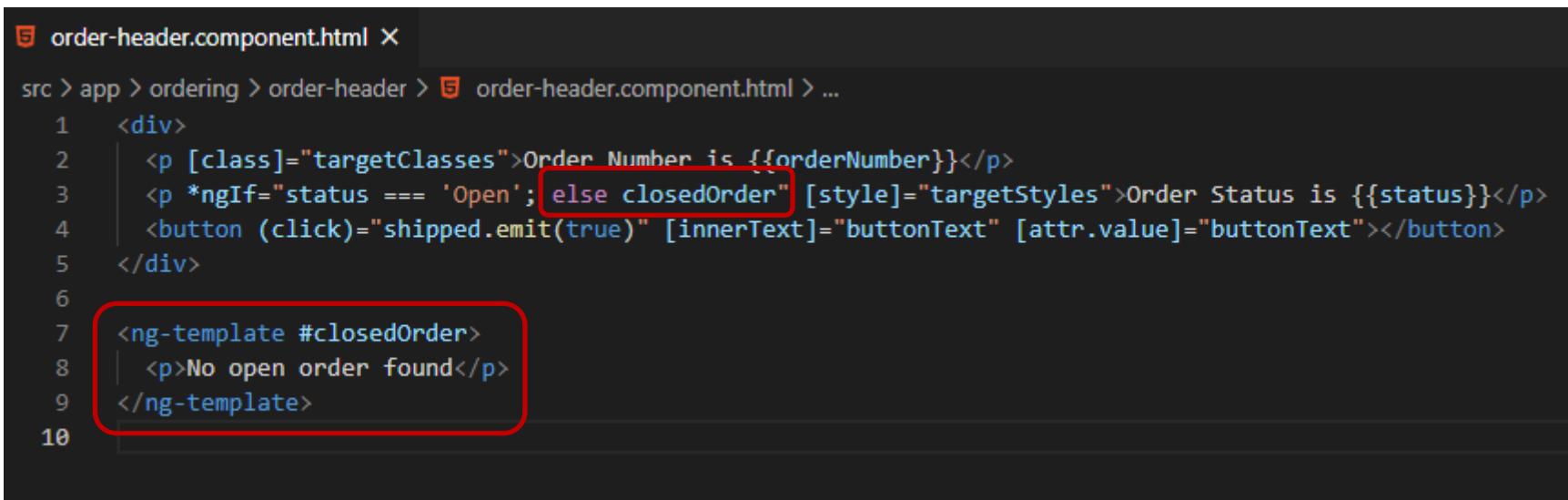
```
...<!DOCTYPE html> == $0
<html lang="en">
  <head>...</head>
  <body>
    <app-root _nghost-dbl-c12 ng-version="11.0.5">
      <app-order-header _ngcontent-dbl-c12 _nghost-dbl-c11 ng-reflect-order-number="DEF-456">
        <div _ngcontent-dbl-c11>
          <p _ngcontent-dbl-c11 class="open safe">Order Number is DEF-456</p>
          <p _ngcontent-dbl-c11 style="color: darkred; width: 75px;">Order Status is Open</p>
          <!--bindings={-->
            <ng-reflect-ng-if: "true" -->
          <!--bindings={-->
            <button _ngcontent-dbl-c11 value="Ship it NOW!!">Ship it NOW!!</button>
          </div>
        </app-order-header>
      </app-root>
      <script src="runtime.js" defer></script>
      <script src="polyfills.js" defer></script>
      <script src="vendor.js" defer></script>
      <script src="main.js" defer></script>
    </body>
  </html>
```

DOM When Status is “Closed”

```
...<!DOCTYPE html> == $0
<html lang="en">
  <head>...</head>
  <body>
    <app-root _nghost-hcc-c12 ng-version="11.0.5">
      <app-order-header _ngcontent-hcc-c12 _nghost-hcc-c11 ng-reflect-order-number="DEF-456">
        <div _ngcontent-hcc-c11>
          <p _ngcontent-hcc-c11 class="safe">Order Number is DEF-456</p>
          <!--bindings={-->
            <ng-reflect-ng-if: "false" -->
          <!--bindings={-->
            <button _ngcontent-hcc-c11 value="Ship it NOW!!">Ship it NOW!!</button>
          </div>
        </app-order-header>
      </app-root>
      <script src="runtime.js" defer></script>
      <script src="polyfills.js" defer></script>
      <script src="vendor.js" defer></script>
      <script src="main.js" defer></script>
    </body>
  </html>
```



- If an alternate element is to be displayed instead, we have a couple of options:
 - Add another `*ngIf` using an opposite boolean expression
 - Use an `else` with an `<ng-template>` element and a template reference variable

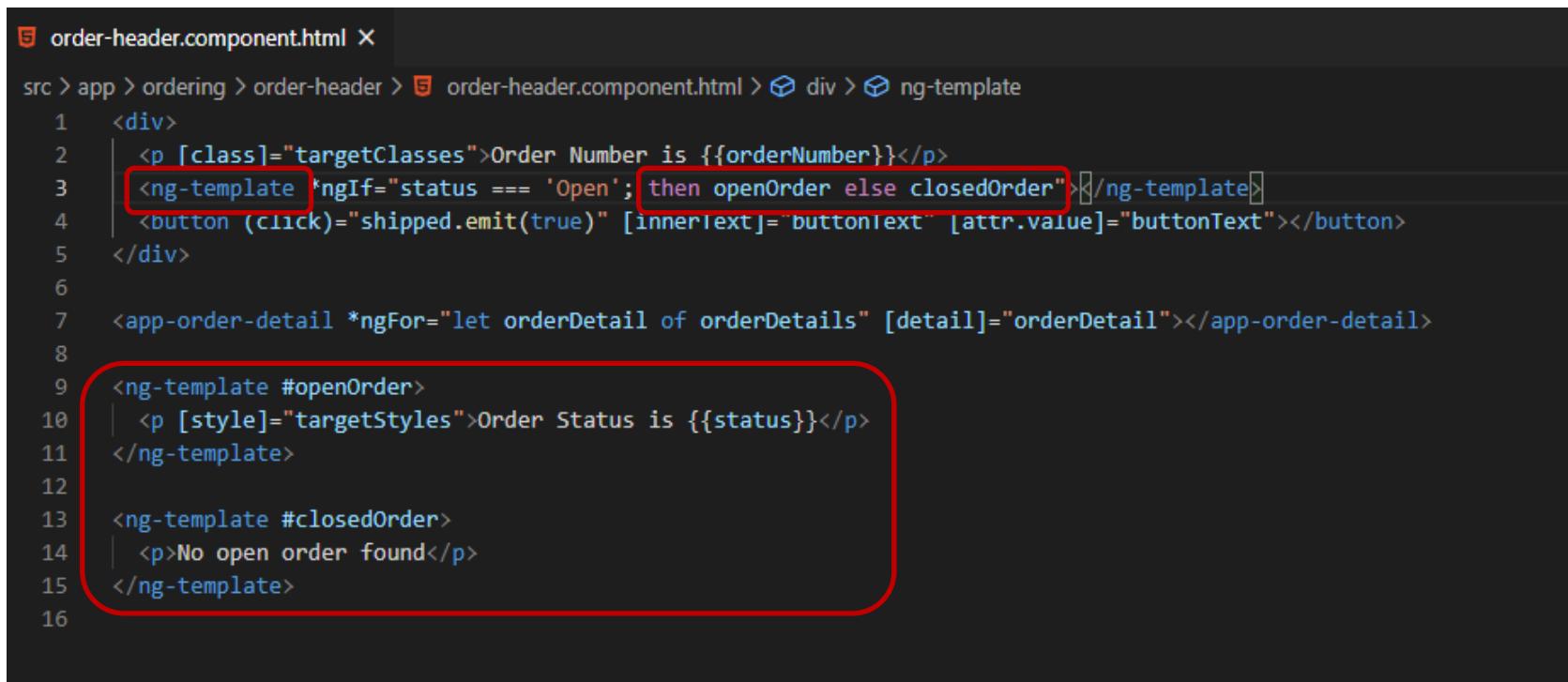


```
order-header.component.html X
src > app > ordering > order-header > order-header.component.html > ...
1  <div>
2    <p [class]="targetClasses">Order Number is {{orderNumber}}</p>
3    <p *ngIf="status === 'Open'; else closedOrder" [style]="targetStyles">Order Status is {{status}}</p>
4    <button (click)="shipped.emit(true)" [innerText]="buttonText" [attr.value]="buttonText"></button>
5  </div>
6
7  <ng-template #closedOrder>
8    <p>No open order found</p>
9  </ng-template>
10
```



ngIf Using <ng-template>

- In fact, an `<ng-template>` can be used for both the `*ngIf` and the `else`
- Additional `*ngIf` statements can be nested within the `<ng-template>` definitions for testing more conditions (if-elseif)



```
order-header.component.html X
src > app > ordering > order-header > order-header.component.html > div > ng-template
1  <div>
2  |   <p [class]="targetClasses">Order Number is {{orderNumber}}</p>
3  |   <ng-template *ngIf="status === 'Open'; then openOrder else closedOrder"></ng-template>
4  |   <button (click)="shipped.emit(true)" [innerText]=buttonText [attr.value]=buttonText></button>
5  </div>
6
7  <app-order-detail *ngFor="let orderDetail of orderDetails" [detail]="orderDetail"></app-order-detail>
8
9  <ng-template #openOrder>
10 |   <p [style]="targetStyles">Order Status is {{status}}</p>
11 </ng-template>
12
13 <ng-template #closedOrder>
14 |   <p>No open order found</p>
15 </ng-template>
16
```



- Used to iterate over a collection of items and render a template for each
- Observes changes in the underlying collection and updates rendered templates as items are added, removed or reordered

```
order-header.component.html X

src > app > ordering > order-header > order-header.component.html > ng-template

1  <div>
2    <p [class]="targetClasses">Order Number is {{orderNumber}}</p>
3    <p *ngIf="status === 'Open'; else closedOrder" [style]="targetStyles">Order Status is {{status}}</p>
4    <button (click)="shipped.emit(true)" [innerText]="buttonText" [attr.value]="buttonText"></button>
5  </div>
6
7  <app-order-detail *ngFor="let orderDetail of orderDetails" [detail]="orderDetail"></app-order-detail>
8
9  <ng-template #closedOrder>
10   <p>No open order found</p>
11 </ng-template>
12
```

Ship it NOW!!

DDA1334 at a count of 25
DAE5543 at a count of 100
WQA4590 at a count of 2
TYR0003 at a count of 45



ngFor – Additional Properties



- *ngFor supports creation of a local reference variable for each item in the collection
- This variable can be used in the template as well
- Format is `*ngFor="let item of items; <property> as <var-name>"`
- Property can be:
 - `index` – 0-based index of item in collection
 - `first/last` – boolean indicating whether item is first or last item
 - `even/odd` – boolean indicating whether `index` of item is even or odd



ngFor – Additional Properties



- Multiple can be chained together separated by semi-colons

```
simple-list.component.html X
src > app > simple-list > simple-list.component.html > ...
1   <ul>
2     <li *ngFor="let item of items; index as indexVal; odd as isIndexOdd" [class.highlight]="isIndexOdd">
3       {{indexVal + 1}}. {{item}}
4     </li>
5   </ul>
6
```

- 1. Widgets
- 2. Hammers
- 3. Screwdrivers
- 4. Televisions
- 5. PS5 Consoles



ngFor – trackBy



- By default, *ngFor uses object identity to track items in a collection
- Assume a method that loads a collection of objects
- Can use *ngFor to display that collection
- By default, when the collection is reset (even with same data), *ngFor is re-rendered in whole



ngFor – trackBy



- Using trackBy, can specify property of items in the collection to be used to determine need for re-render
- Still supports robust change detection for item property changes but manages collection render more performantly

```
order-list.component.ts ×
src > app > ordering > order-list > order-list.component.ts > OrderListComponent > loadAll
  1 import { Component, OnInit } from '@angular/core';
  2 import { OrderService } from 'src/app/services/order.service';
  3 import { OrderHeader } from '../order-header.model';
  4
  5 @Component({
  6   selector: 'app-order-list',
  7   templateUrl: './order-list.component.html',
  8   styleUrls: ['./order-list.component.css']
  9 })
10 export class OrderListComponent implements OnInit {
11
12   orders: Array<OrderHeader>;
13
14   constructor(private orderService: OrderService) { }
15
16   ngOnInit(): void {
17   }
18
19   loadAll(): void {
20     this.orders = this.orderService.getOrders();
21   }
22
23   trackOrderId(index: number, order: OrderHeader): number {
24     return order.id;
25   }
26
27 }
28
```

```
order-list.component.html ×
src > app > ordering > order-list > order-list.component.html > ...
  1 <button (click)="loadAll()">Get All Orders</button>
  2 <table>
  3   <tr>
  4     <th>Order Number</th>
  5     <th>Description</th>
  6     <th>Total</th>
  7   </tr>
  8   <tr *ngFor="let order of orders; trackBy: trackOrderId">
  9     <td [attr.data-id]="order.id">{{order.orderNumber}}</td>
10     <td>{{order.description}}</td>
11     <td>{{order.total | currency }}</td>
12   </tr>
13 </table>
14
```



ngSwitch

- Used to switch between templates based on the value of a variable or expression
- Akin to a switch-case statement in code
- *[ngSwitch]* is used to specify the variable or expression



ngSwitch

- **ngSwitchCase* is used to define the template to be added if its value matches
- **ngSwitchDefault* is used to define a default template if none of the values match
- Unlike switch-case, there is no fallthrough or break



ngSwitch

```
order-header.component.html X
src > app > ordering > order-header > order-header.component.html > div
  1  <div>
  2  |  <p [class]="targetClasses">Order Number is {{orderNumber}}</p>
  3  |  <ng-container [ngSwitch]="status">
  4  |    <p *ngSwitchCase="'Open'" [style]="targetStyles">Order Status is {{status}}</p>
  5  |    <p *ngSwitchCase="'Closed'">No open order found</p>
  6  |    <p *ngSwitchDefault>Order status is unknown</p>
  7  |  </ng-container>
  8  |  <button (click)="shipped.emit(true)" [innerText]="buttonText" [attr.value]="buttonText"></button>
  9  </div>
10
11 <app-order-detail *ngFor="let orderDetail of orderDetails" [detail]="orderDetail"></app-order-detail>
12
```



ngClass and ngStyle



- *ngClass* is used to add or remove CSS classes on an HTML element
- *ngStyle* is used update styles on an HTML element



- *ngClass* accepts the following types of expressions:
 - Space-delimited string of class names to be applied
 - An array of string class names to be applied
 - An object with class names for keys and boolean expressions for values – a true value applies the class defined by the key
- In the object expression, multiple space-delimited class values can be managed by the same boolean expression



```
order-header.component.html X
src > app > ordering > order-header > order-header.component.html > ...
1  <div>
2  |  <p [ngClass]="{'open safe': status === 'Open'}">Order Number is {{orderNumber}}</p>
3  |  <ng-container [ngSwitch]="status">
4  |  |  <p *ngSwitchCase="'Open'" [style]="targetStyles">Order Status is {{status}}</p>
5  |  |  <p *ngSwitchCase="'Closed'">No open order found</p>
6  |  |  <p *ngSwitchDefault>Order status is unknown</p>
7  |  </ng-container>
8  |  <button (click)="shipped.emit(true)" [innerText]="buttonText" [attr.value]="buttonText"></button>
9  </div>
10
11 <app-order-detail *ngFor="let orderDetail of orderDetails" [detail]="orderDetail"></app-order-detail>
12
```



ngStyle



- *ngStyle* accepts one or more style identifiers with corresponding expressions
- If the expression evaluates to a non-null value, the style is applied
- Otherwise, the style is removed



- Where applicable, supports an optional unit suffix
- Also supports object expressions containing a set of key-value pairs

```
order-header.component.html X
src > app > ordering > order-header > order-header.component.html > ...
1  <div>
2    <p [ngClass]="{'open safe': status === 'Open'}">Order Number is {{orderNumber}}</p>
3    <ng-container [ngSwitch]="status">
4      <p *ngSwitchCase="Open" [ngStyle]="{'background-color': colorValue, 'width.%': widthValue}">Order Status is {{status}}</p>
5      <p *ngSwitchCase="Closed">No open order found</p>
6      <p *ngSwitchDefault>Order status is unknown</p>
7    </ng-container>
8    <button (click)="shipped.emit(true)" [innerText]="buttonText" [attr.value]="buttonText"></button>
9  </div>
10
11 <app-order-detail *ngFor="let orderDetail of orderDetails" [detail]="orderDetail"></app-order-detail>
12
```



Creating Custom Directives



- To create a custom directive, we use *ng generate directive <name>*
- Angular creates a TypeScript file for the directive called `<name>.directive.ts`
- Be aware – directive will be created in the folder where *ng generate* is executed
- It will be registered with a module local to the folder (or *AppModule* if none exists there)



@Directive() Decorator



- Directive is defined by TypeScript class decorated with `@Directive()`
- Accepts an object parameter that includes a property called `selector`
- Property is required and represents named placeholder applied to target host



Data Bindings in Directives



- As with components, `@Input()` properties can be applied to directives
- Creates input bindings for the directive
- Defined input bindings are specified on the host where directive is applied
- For example, `<p myDirective [inputVal] = "value">...</p>`



Data Bindings in Directives



- Can also use the directive as an input binding as well (subject to some rules)
- Requires an `@Input()` property with same name as directive selector
- Attribute directives support additional inputs in the standard way

```
1 import { Directive, HostBinding, HostListener, Input } from '@angular/core';
2
3 @Directive({
4   selector: '[appNumbersOnly]'
5 })
6 export class NumbersOnlyDirective {
7
8   @Input('appNumbersOnly') numbersOnly: boolean;
9   @Input() validClass: string;
10  @Input() invalidClass: string;
11}
```

```
1 <input [appNumbersOnly]="true" [validClass]="'valid'" [invalidClass]="'invalid'" />
2
```



Data Bindings in Directives



- Structural directives require additional input names to be prefixed with the directive selector
- Host syntax is different as well

```
1 import { Directive, Input, TemplateRef, ViewContainerRef, OnInit } from '@angular/core';
2
3 @Directive({
4   selector: '[appViewManager]'
5 })
6 export class ViewManagerDirective implements OnInit {
7
8   @Input() appViewManager: string[];
9   @Input('appViewManagerSecondaryRoles') secondaryRoles: string[] = [];
10
11   private currentRole = 'supervisor'.
```

```
1 <input [appNumbersOnly]="" [validClass]="" [invalidClass]="" />
2 <div *appViewManager="['supervisor', 'plant manager']; secondaryRoles: ['admin']">
3   <p>Super-secret display area!!</p>
4 </div>
5
```



Data Bindings in Directives



- Means that structural directives require an input property with name matching selector if additional inputs are required
- Attribute directives do not
- Additionally, not required to have any input bindings if intent is to use presence of directive only



Bindings Directive Details to Host



- `@HostBinding()` decorator can be used to bind a directive value to property of host element
- `@HostListener()` decorator can be used to bind to an event of host element
- Enables implementation of custom logic in directive tied to details of the host
- For example, binding to the `class` attribute for programmatically updating styles on the host



Angular Pipes



Angular Pipes



- Pipes allow the filtering and transformation of data at the view level
- The applied pipe does not persist any changes to the underlying data
- General format is `<expression> | <pipe name>`
- Also support chaining for additional view transformation (i.e., `<expression> | <pipe1> | <pipe2>`)



- Several built-in pipes are available for use:
 - *uppercase/lowercase* – transforms string to designated casing
 - *percent* – formats a number as a percentage
 - *currency* – formats a number as local currency; can override with different currency code
 - *slice* – retrieves a subset of elements from an array; accepts starting index and optional ending index
 - *date* – formats expression as a date; accepts an optional format string
 - *json* – outputs an object in JSON format; useful for debugging purposes



Built-In Pipes

```
1 <p>{{'this is some text' | uppercase }}</p>
2 <p>{{'THIS IS SOME OTHER TEXT' | lowercase}}</p>
3 <p>{{0.23239543 | percent:'0.2'}}</p>
4 <p>{{945.32344 | currency:'EUR'}}</p>
5 <p>{{[1, 2, 3, 4] | slice:1:3}}</p>
6 <p>{{[1, 2, 3, 4] | slice:1}}</p>
7 <p>{{currentDate | date: 'fullDate'}}</p>
8 <p>{{currentDate | date: 'yyyy-MM-dd'}}</p>
9 <p>{{obj | json}}</p>
10
```

THIS IS SOME TEXT

this is some other text

23.24%

€945.32

2,3

2,3,4

Sunday, January 3, 2021

2021-01-03

{ "prop1": "One", "prop2": "Two", "prop3": "Three", "prop4": "Four" }



Creating Custom Pipes



- To create a custom pipe, we use `ng generate pipe <name>`
- Angular creates a TypeScript file for the pipe called `<name>.pipe.ts`
- Be aware – pipe will be created in the folder where `ng generate` is executed
- It will be registered with a module local to the folder (or `AppModule` if none exists there)



@Pipe() Decorator



- Pipe is defined by TypeScript class decorated with `@Pipe()`
- Accepts an object parameter that includes a property called `name`
- Property is required and represents name used with `|`
- The pipe class will implement the `PipeTransform` interface



PipeTransform



- Includes a single method called *transform* that will contain pipe's transformation logic
- Format of initial scaffolded method will look like this:

```
transform(value: unknown, ...args: unknown[]): unknown {
  return null;
}
```

- We modify input types, output type and implement pipe logic, returning results of transformation



PipeTransform



- *value* parameter will be the value to which the pipe is being applied
- Specific parameters can be added to the pipe signature (if required)
- When applying the pipe, parameters will be separated with a colon (:)

transform(value: Person[], primarySort: string, secondarySort?: string) : Person[]

*<li *ngFor="let person of people | sorter:'lastName'">...*

-or-

*<li *ngFor="let person of people | sorter:'lastName':'firstName'">...*



Angular Pipes



- By default, a custom pipe is re-executed only when the reference of the value changes
- These are known as “pure” pipes (the default)
- If items added to array or properties of object change, use immutable data management techniques to modify
- Resets reference (causing pipe to re-execute)



Angular Routing



Angular Routing



- Facilitates navigation between different components (e.g., routing from list view to detail view)
- Defines a set of paths and the component that should be rendered for each
- When user hits that path in the browser, app will render the component



Angular Routing



- Enables a more dynamic navigation experience
- Instead of static component tags in our HTML, we use a generic routing placeholder
- Supports definition of a “wildcard path” as a catch-all if none of the route paths match
- Also enables designation of route-based parameters



Adding Angular Routing



- When creating a new app using `ng new`, the Angular CLI will ask if routing should be enabled
- Creates the necessary scaffolding for routing on app creation
- However, can also add routing to an existing app not scaffolded for it



Adding Angular Routing

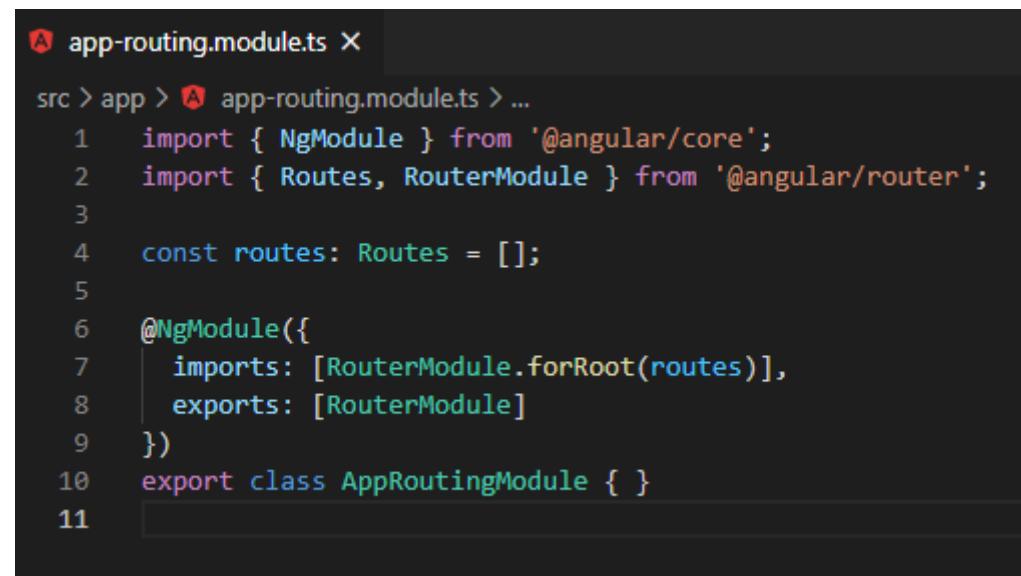


- Check “package.json” to confirm “@angular/router” is included
- If not, run *npm install @angular/router* to add
- Execute *ng generate module appRouting --flat --module=app* from app root
- This will generate a new module called app-routing.module.ts



Adding Angular Routing

- Verify that *AppRoutingModule* has been added to the *AppModule* imports
- Replace the *AppRoutingModule* code with the following base content



```
app-routing.module.ts ×

src > app > app-routing.module.ts > ...
1 import { NgModule } from '@angular/core';
2 import { Routes, RouterModule } from '@angular/router';
3
4 const routes: Routes = [];
5
6 @NgModule({
7   imports: [RouterModule.forRoot(routes)],
8   exports: [RouterModule]
9 })
10 export class AppRoutingModule { }
```



Adding Angular Routing



- Add a `<router-outlet></router-outlet>` element to the template for `AppComponent`
- Verify the presence of a `<base href="/">` element in `./src/index.html`
- To test, create a new component (or use an existing one)
- In `app-routing.module.ts`, add a new entry to the `routes` collection (called a route configuration)



Adding Angular Routing



- Entry will be an object with two properties:
 - A string path value (called *path*)
 - The component that should be rendered for the defined path (called *component*)
- For example, `{ path: 'landing', component: LandingComponent }`
- If all is set up correctly, on launch and navigation to the configured path, the specified component should be rendered



Adding Angular Routing

- Can add a default path using an empty string for *path* and a redirect
- You can add a “wildcard path” using “`**`” for path
- Routing will use the first match it finds – because of that, order of the routes is important (more specific routes listed first)

```
5
6  const routes: Routes = [
7    { path: 'landing', component: LandingComponent },
8    { path: '', redirectTo: '/landing', pathMatch: 'full' },
9    { path: '**', component: PageNotFoundComponent }
10];
11
```



Navigating to a Route



- Couple of options:
 - Add the *routerLink* directive to `<a>` elements
 - Use the *navigate()* method of *Router* service in code
- *routerLink* associates a string path to the hyperlink for navigating to a route
- For example, `Home`



Navigating to a Route



- routerLinkActive is another directive used for styling the active link
- For example,

```
<a routerLink="/landing" routerLinkActive="active">Home</a>
<a routerLink="/orders" routerLinkActive="active">Access Orders</a>
```

- Used to apply a CSS class to a link to style when route selected



Navigating to a Route



- Router service (exported by *RouterModule*) can be used for routing in code
- Includes a method called `navigate`
- Method accepts an array containing destination route path and any route parameters

```
8   }
9   export class PageNotFoundComponent implements OnInit {
10
11   constructor(private router: Router) { }
12
13   ngOnInit(): void {
14   }
15
16   goHome(): void {
17     this.router.navigate(['/landing']);
18   }
19
20 }
21
```



Feature Routing Modules



- Instead of using *AppRoutingModule* for all route configuration, can localize routing to our feature modules
- *AppRoutingModule* continues to house application-level routing (e.g., home page, page not found, etc.)
- Each feature module can also define its own routing



Feature Routing Modules



- Routing can be added at time of module creation using the *routing* flag
- For example, *ng generate module <mod-name> --routing*
- Creates <mod-name>-routing.module.ts along with <mod-name>.module.ts

```
ordering-routing.module.ts ✘
src > app > ordering > ordering-routing.module.ts > ...
1 import { NgModule } from '@angular/core';
2 import { Routes, RouterModule } from '@angular/router';
3
4 const routes: Routes = [];
5
6 @NgModule({
7   imports: [RouterModule.forChild(routes)],
8   exports: [RouterModule]
9 })
10 export class OrderingRoutingModule { }
```



Feature Routing Modules



- To add routing to an existing feature module, execute `ng generate module <mod-folder>/<mod-name>Routing --flat --module=<mod-name>`
- This will generate a new module called `<mod-name>-routing.module.ts`

```
ordering-routing.module.ts ✘
src > app > ordering > ordering-routing.module.ts > ...
1  import { NgModule } from '@angular/core';
2  import { Routes, RouterModule } from '@angular/router';
3
4  const routes: Routes = [];
5
6  @NgModule({
7    imports: [RouterModule.forChild(routes)],
8    exports: [RouterModule]
9  })
10 export class OrderingRoutingModule { }
```



Feature Routing Modules



- Feature routing module will have its own route configuration
- Instead of `forRoot`, the feature routing module uses `forChild` to import
- As with `AppRoutingModule`, the `RouterModule` is an import and export



Feature Routing Modules

- The feature module will import the feature routing module
- *AppModule* will import the feature module (enabling the feature routing by extension)
- With routing, the feature module should be placed above the *AppRoutingModule*

```
ordering.module.ts X
src > app > ordering > ordering.module.ts > ...
1  import { NgModule } from '@angular/core';
2  import { CommonModule } from '@angular/common';
3
4  import { OrderingRoutingModule } from './ordering-routing.module';
5
6
7  @NgModule({
8    declarations: [],
9    imports: [
10      CommonModule,
11      OrderingRoutingModule
12    ]
13  })
14  export class OrderingModule { }
```

```
4  import { AppComponent } from './app.component';
5  import { AppRoutingModule } from './app-routing.module';
6  import { LandingComponent } from './landing/landing.component';
7  import { PageNotFoundComponent } from './page-not-found/page-not-found.component';
8  import { OrderingModule } from './ordering/ordering.module';
9
10 @NgModule({
11   declarations: [
12     AppComponent,
13     LandingComponent,
14     PageNotFoundComponent
15   ],
16   imports: [
17     BrowserModule,
18     OrderingModule,
19     AppRoutingModule
20   ],
21   providers: [],
22   bootstrap: [AppComponent]
23 })
24 export class AppModule { }
```



Using Route Parameters



- Route parameters can be used to send additional data to inform a route
- For example, sending the id of a specific order on the order route to enable dynamic display of that order's detail
- The component for the detail route could use the id to look up order details using a service



Using Route Parameters



- The parameter can be added to the *path* value of the route
- For example, `{ path: 'order/:id', component: OrderDetailComponent }`
- The colon (:) denotes a route parameter
- If multiple parameters are required, separate each with “/”



Using Route Parameters

- The parameter value can be included in the *routerLink* directive
- For example, `<a [routerLink]=“[‘/order’, order.id]”>...`
- Can also be included in the call to the Router service’s `navigate` method (in code)

```
goToOrder(): void {
  this.router.navigate(['/order', this.order.id]);
}
```



Using Route Parameters



- In the route component, the *ActivatedRoute* service can be used to get information for the currently active route (including parameters)
- *ActivatedRoute* includes a couple of observables for this purpose
- *paramMap* and *queryParamMap* provide access to route parameters and query string parameters respectively



Using Route Parameters



- An alternative to the observables is use of a route snapshot
- Contains a point-in-time snapshot of the current route detail
- The snapshot property of `ActivatedRoute` includes `paramMap` and `queryParamMap` properties
- Represent key-value collections housing parameter values



- Can be used for route orchestration in a container component
- Utilizes a separate `<router-outlet>` element for placement of child routes
- Child route information is defined in the route configuration for the container component using the `children` property

```
5
6  const routes: Routes = [
7    {
8      path: 'orders',
9      component: OrdersContainerComponent,
10     children: [
11       { path: ':id', component: OrderDetailComponent }
12     ]
13   }
14 ];
15
```



Route Guards



- In some cases, may want to manage access to or from a given route
- Examples include preventing unauthorized access to a route or verifying with a user before navigating away from a route
- To do so, we leverage an Angular service called a guard



Route Guards



- To create a new guard use *ng generate guard <name>*
- At creation, you will be prompted for the set of interfaces (i.e., guard actions) to be implemented
- Angular will scaffold a new TypeScript class for the guard accordingly



Guard Types



- Includes:
 - *CanActivate* – Controls access to a route
 - *CanActivateChild* – Controls access to a route's child routes
 - *CanDeactivate* – Controls navigation away from a route
 - *CanLoad* – Controls access to a lazy-loaded route



General Guard Functionality



- Parameters provide information about current and target routes (route parameters, query string parameters, etc.)
- If all assigned guards of a specific type return true, routing type proceeds
- If any assigned guard returns false, routing type will be cancelled



General Guard Functionality



- Guards can use the *Router* service to navigate to an alternate route
- Or can return a *UrlTree* which will cancel current route and begin a new one
- Both synchronous and asynchronous versions are available



General Guard Functionality



- To assign a guard to a route, update the route configuration
- In addition to *path* and *component* properties, there are properties for each guard type (named after type)
- Properties are of array type – allows assignment of multiple guards of a given type



Lazy-Loaded Routes



- As application complexity grows, bundle size can grow as well
- If large enough, can impact load or startup performance
- For routes that may be accessed with less frequency, lazy-loading can help



Lazy-Loaded Routes



- Modules for lazy-loaded routes are bundled separately, reducing main application bundle size
- Those modules can get loaded by request – once requested, will remain loaded for the session
- To lazy-load a module, we do not include it in the *imports* property of another module



Lazy-Loaded Routes



- We load in code as part of the route configuration
- Uses an additional property in the config called *loadChildren*
- Leverages a method called *import* that returns a *Promise*
- On completion of the *Promise*, the module is loaded (in response to route request)

```
{  
  path: 'contactus',  
  loadChildren: () => import('./contact-us/contact-us.module').then(m => m.ContactUsModule),  
  ...  
}
```



Guarding Lazy-Loaded Routes



- To guard lazy-loaded routes, we use a guard that implements *CanLoad*
- Operates similarly to the other guards:
 - Provides parameters with details about the route
 - Returns either a boolean or a UrlTree
 - Provides both sync and async options
- In route configuration, applied in the *canLoad* property (an array)



Angular Services



Purpose



- Angular services are TypeScript classes with specific purposes
- Intended to house and provide shared functionality for reuse across the app
- No requirement to use services in your Angular app
- However, considered a best practice



Purpose



- Used for tasks like pulling data from an API, specialized validation or logging
- Logic could be coded directly in components
- However, components should focus solely on view management



Purpose

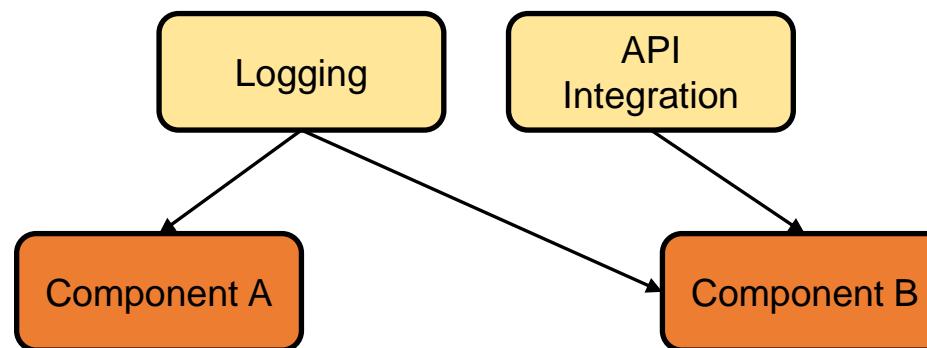
- Capturing logic in service:
 - Keeps components smaller and easier to manage
 - Enables support for multiple components
 - Facilitates testing
 - Promotes modularity



Dependency Management



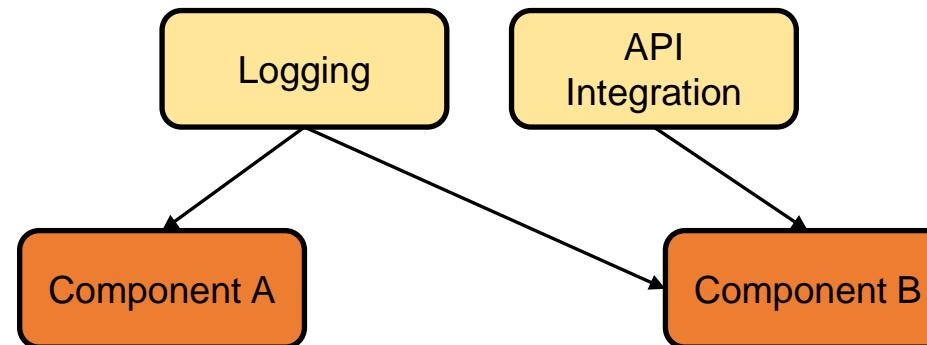
- To keep components modular, leverage reusable logic from other classes
- Classes with reusable logic are known as dependencies





Dependency Management

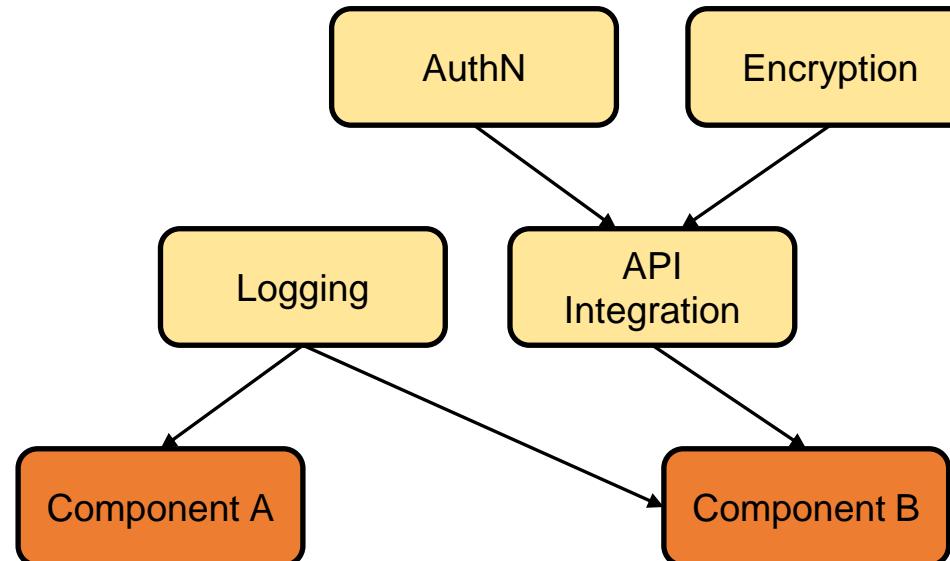
- One option is to explicitly create new instances of dependencies as needed
- Requires extra code
- Could waste memory if new instance created in every component where logic is required





Dependency Management

- Exacerbated if dependency also has downstream dependencies
- When testing, requires creation and configuration of the dependency tree
- And a dependency used for API access requires access to the API in the test
- Difficult to test the component logic directly and in isolation (true unit test)





Dependency Injection



- Instead of explicitly managing dependencies, Angular allows us to delegate
- Provides a rich Dependency Injection mechanism to help automate management
- Greatly loosens coupling between components and dependencies



Dependency Injection



- Dependency reference added to constructor of consuming class
- When consuming class is created, injector checks for existing dependency instance
- If found, reuses; otherwise, creates and returns new singleton instance
- Adds newly created instance to pool of instances for future use

```
orders-container.component.ts X
src > app > ordering > orders-container > orders-container.component.ts > ...
1 import { Component, OnInit, ViewChild } from '@angular/core';
2 import { Order } from 'src/app/page-not-found/page-not-found.component';
3 import { DataService } from 'src/app/services/data.service';
4 import { OrderDetailComponent } from '../order-detail/order-detail.component';
5
6 @Component({
7   selector: 'app-orders-container',
8   templateUrl: './orders-container.component.html',
9   styleUrls: ['./orders-container.component.css']
10 })
11 export class OrdersContainerComponent implements OnInit {
12   @ViewChild('orderDetail') orderDetail: OrderDetailComponent;
13
14   constructor(private dataService: DataService) { }
15
16   ngOnInit(): void {
17   }
18
19   onShip(isShipped: boolean): void {
20     if (isShipped) {
21       window.alert(`Shipped order number ${this.orderDetail.orderNumber}`);
22     }
23   }
24
25   getOrders(): Order[] {
26     return this.dataService.getOrders();
27   }
28
29 }
30
```



Creating Custom Services



- To create a custom service, we use `ng generate service <name>`
- Angular creates a TypeScript file for the service called `<name>.service.ts`
- Be aware – service will be created in folder where `ng generate` is executed
- Not registered with a module – instead, registered with an injector



@Injectable() Decorator



- Service is defined by TypeScript class decorated with `@Injectable()`
- Accepts an optional object parameter that includes property called *providedIn*
- By default, value for property will be ‘root’



@Injectable() Decorator



- Indicates service registered with root injector
- When registered with root injector, available across app
- `@NgModule` decorator includes *providers* property
- Can also register with root injector by adding to *providers* in `AppModule`

```
5
4  @Injectable({
5    providedIn: 'root'
6  })
7  export class DataService {
8
9    constructor() { }
10
11   getOrders(): Order[] {
12     // Logic to retrieve orders
13     return [] as Order[];
14   }
15 }
16 }
```



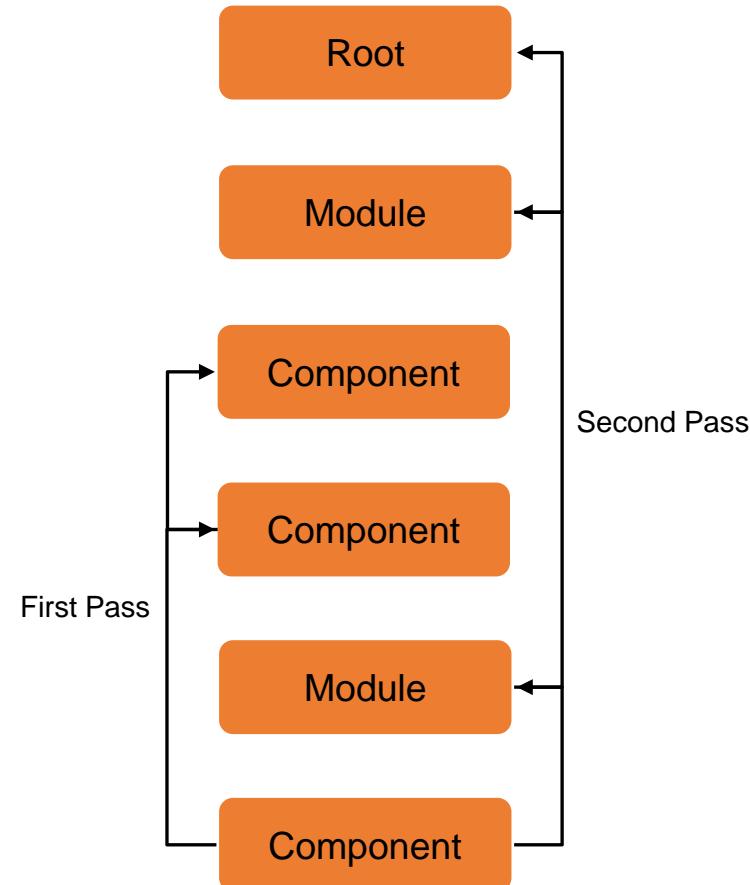
The Injector Tree



- Components & modules also provide injectors
- The set of available injectors represents a hierarchy used during resolution
- When encountering a dependency in a constructor, Angular searches this tree looking for a provider to handle it



The Injector Tree





The Injector Tree

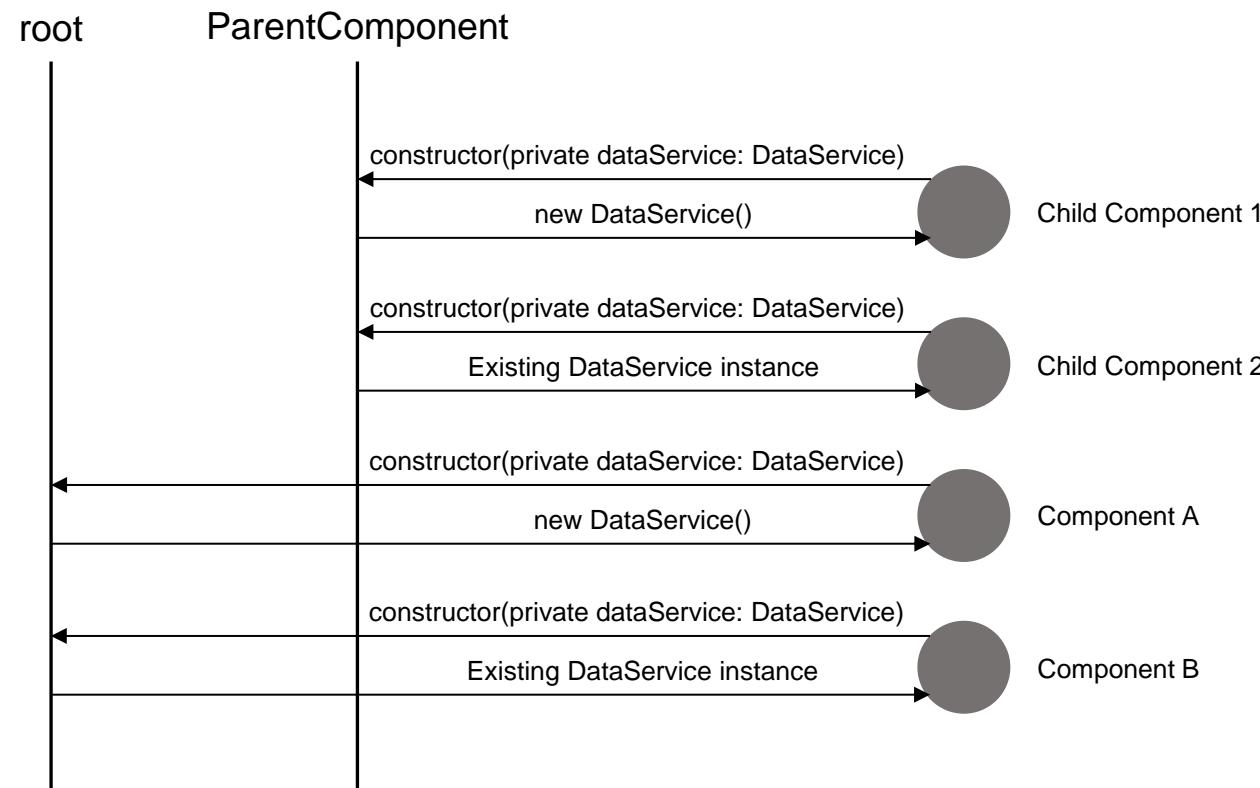


- We've already seen the *providers* property in `@NgModule()`
- There is also a *providers* property available in `@Component()`
- With `@Component()`, service registered in *providers*:
 - Can be shared with child components
 - Can create multiple copies every time component is rendered – scoped to instance



The Injector Tree

- Same service can be registered with a component and root injectors
- Angular uses closest ancestor for dependency resolution





Restricting Dependency Injection



- `@Host()` decorator restricts injector lookup to local and parent only
- If unable to find a match, will not bubble up – will throw exception instead
- `@Optional()` decorator will return null for dependency instead of exception
- Obviously, service calls against a null reference would fail



Restricting Dependency Injection



- `@Self()` decorator restricts injector lookup to local component only
- `@Self()` can also be combined with `@Optional()`
- `@SkipSelf()` decorator skips local injector and searches up the hierarchy
- For example:

```
constructor(@Host() @Optional() private dataService: DataService)
```



Overriding Providers



- As discussed, the *providers* property can also be used for service injection
- Accepts one or more service classes
- Also, accepts an object used to designate how service instances are created by the injector



Overriding Providers



providers: [DataService]

is equivalent to

providers: [{ provide: DataService, useClass: DataService }]



Overriding Providers



- *provide* property defines what consumers use in their constructor
- Second property defines what will be provided to consumers as implementation on dependency resolution
- Second property can be one of:
 - *useClass*
 - *useFactory*
 - *useValue*



Overriding Providers



- *useClass* can be used to override implementation with a different class
- For example,

providers: [{ provide: DataService, useClass: MockDataService }]



Overriding Providers



- *useFactory* can be used to override implementation with a dynamically generated service instance
- *deps* can be used to inject downstream dependencies for instance types returned from factory method
- For example,

```
providers: [
    provide: DataService,
    useFactory: dataServiceFactory(),
    deps: [HttpClient]
]
```



Overriding Providers



- *useValue* can be used to provide a value instead of class instance
- Value returned could be a simple value or a dynamically-defined object
- Allows return of constants as part of dependency resolution



Asynchronous Data





Working with Data



- Often, an Angular app will make calls to an API for data operations
- The API is likely off box and out of process
- Because of that, API operations can take some time to complete



Working with Data



- Rather than block the UI thread, the calls are made asynchronously
- App can move on to other processing/interaction with user
- When data operation completes, can be notified for results processing



Asynchronous Calls



- Traditionally, we've had 3 options for handling:
 - Callbacks
 - Promises
 - *async/await*



Callbacks



- Function reference passed to a method executing asynchronous activity
- When the async processing finishes, calls the provided method to notify
- Callback method can then be used to process results or move to next step in sequence



Callbacks



- When working with a single async operation, this approach is OK
- With multiple async operations in sequence, the code can get ugly quickly
- Often results in several nested levels of calls that can be hard to read and maintain



Promises



- Provide an improvement over callbacks
- Multiple asynchronous operations can be logically chained together
- Also, operations can be split off and results can be combined and resolved together as a unit



Promises

- To use, create a new instance of the *Promise* type
- Constructor accepts *resolve* and *reject* method references
- The *resolve* method reference is a callback executed on success
- The *reject* method reference is a callback executed on failure



- While still technically using callbacks, the code becomes much cleaner
- Multiple async operations are chained using `then()`
- Set of chained operations might look like this:

```
this.getOrder(orderNumber)
  .then((o: Order) => Promise.all([this.getShippingInformation(o), this.getTaxInformation(o)]))
  .then((result: [ShippingInformation, TaxInformation]) => combine(result));
```



async/await



- “Syntactic sugar” for Promises
- Code can call asynchronous methods in sequence (like other methods)
- *await* is only valid in a method defined as *async*

```
async getAllOrderInformation(orderNumber): Order {  
    const order = await this.getOrder(orderNumber);  
    order.ShippingInformation = await this.getShippingInformation(order);  
    order.TaxInformation = await this.getTaxInformation(order);  
    return order;  
}
```



Issues with Promises



- Promises can work for some use cases
- However, there are some limitations:
 - Cannot be canceled
 - Are immediately executed – executed at point of call
 - Represent one-time operations only – no easy way to retry or handle a “stream” of results



Observables



- Observables are defined in the RxJS library
- Support asynchronous and event-based interaction
- Offer similar capabilities as Promises with additional benefits



Observables

- An observable is an object that maintains a list of dependent observers
- Informs observers about state changes through async emit of events
- Observers subscribe to be notified and react to the changed state
- Subscribed observers continue to “listen” and be notified until either:
 - An unsubscribe
 - Observable is destroyed



Observables



- Observables return a stream of events
- Observers receive prompt notification of a new occurrence
- The stream can be a combination of different async operations
- Operators exist to enable things like sequencing vs. parallel, filtering, etc.



Reactive Functional Programming



- Requires the following:
 - An observable
 - One or more observers
 - A timeline
 - A stream of events
 - A set of composable operators
- Using RxJS and Observables, can apply async subscriptions and transformations to an observable stream of events or state changes



RxJS Operators



- RxJS includes extensive set of operators (<https://rxjs.dev/guide/operators>)
- Categories include creation, transformation, filtering, join, error handling and utility operators
- Enable rich composition in the logic used to handle state changes or events

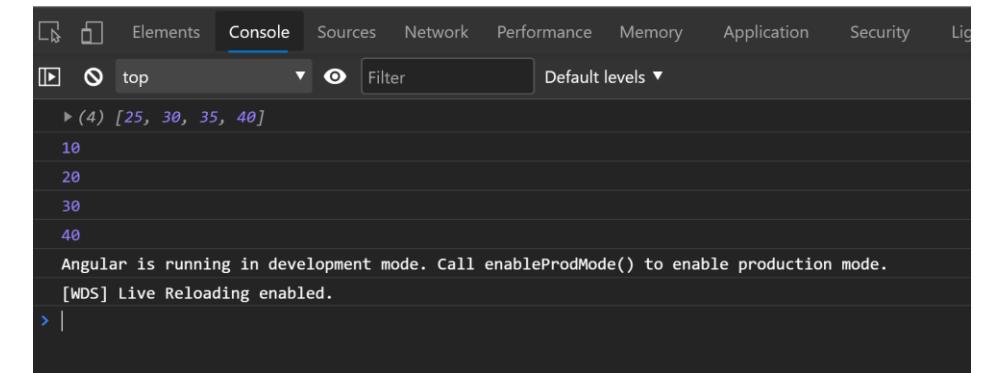
- Wraps a set of items into an observable sequence
- Emits each argument passed to it and then completes
- Each argument is treated as an implicit *next()* – the observable's way of sending a new notification to subscriber(s)



of

```
observables.component.ts X
src > app > observables > observables.component.ts > ...
1 import { Component, OnDestroy, OnInit } from '@angular/core';
2 import { of, Subscription } from 'rxjs';
3
4 @Component({
5   selector: 'app-observables',
6   templateUrl: './observables.component.html',
7   styleUrls: ['./observables.component.css']
8 })
9 export class ObservablesComponent implements OnInit, OnDestroy {
10
11   sequence$ = of(10, 20, 30, 40);
12   subscription: Subscription;
13
14   constructor() { }
15
16   ngOnInit(): void {
17     this.subscription = this.sequence$.subscribe(value => console.log(value));
18   }
19
20   ngOnDestroy(): void {
21     this.subscription?.unsubscribe();
22   }
23
24 }
25
```

observables works!





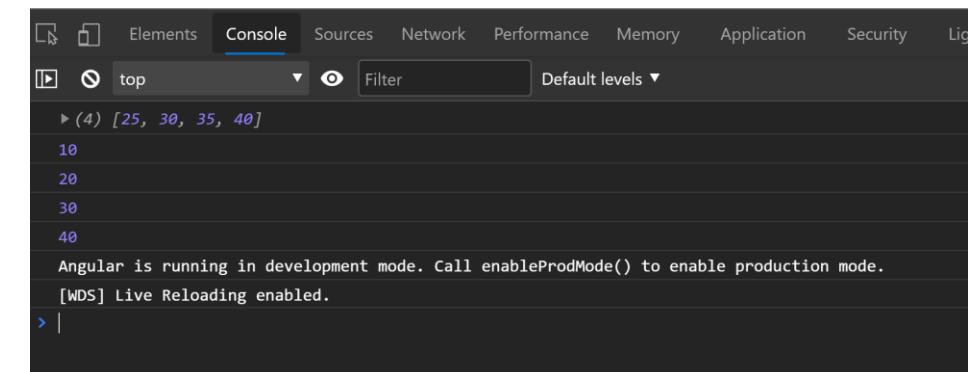
- Converts almost anything to an Observable
- Accepts an *Array*, a *Promise*, an iterable object, etc.
- As with *of()*, each argument is treated as an implicit *next()*
- Unlike *of()*, *from()* flattens – e.g., emits a *next()* separately for each element of an *Array*



from

```
observables.component.ts X
src > app > observables > observables.component.ts > ...
1 import { Component, OnDestroy, OnInit } from '@angular/core';
2 import { from, Subscription } from 'rxjs';
3
4 @Component({
5   selector: 'app-observables',
6   templateUrl: './observables.component.html',
7   styleUrls: ['./observables.component.css']
8 })
9 export class ObservablesComponent implements OnInit, OnDestroy {
10
11   sequence$ = from([10, 20, 30, 40]);
12   subscription: Subscription;
13
14   constructor() { }
15
16   ngOnInit(): void {
17     this.subscription = this.sequence$.subscribe(value => console.log(value));
18   }
19
20   ngOnDestroy(): void {
21     this.subscription?.unsubscribe();
22   }
23
24 }
```

observables works!



The screenshot shows the Chrome DevTools Console tab with the title 'top'. The console output displays the values 10, 20, 30, and 40, each on a new line, indicating that the Observable sequence has been correctly subscribed to and logged.

```
▶ (4) [25, 30, 35, 40]
10
20
30
40
Angular is running in development mode. Call enableProdMode() to enable production mode.
[WDS] Live Reloading enabled.
```

- Method that can be called on an Observable
- Combines a sequence of operations against event or notification data
- Able to “pipe” results of one operator to another (depending on the operator)
- Final `subscribe()` will receive result from all operators being applied



map

- Projects data received into a new version or value
- Returns an Observable that wraps the updated result
- Can be used in a *pipe()* to transition event data as it flows through a sequence of steps



- Filters data received in Observable stream according to a provided predicate
- Values that meet provided predicate condition are included (boolean)
- Returns an Observable that wraps the set of filtered values



Multiple Operators



```
ngAfterViewInit(): void {
  const logger = fromEvent(this.keyInput.nativeElement, 'keyup');
  this.subscription = logger.pipe(
    map((evt: KeyboardEvent) => evt.key.charCodeAt(0)),
    filter(code => {
      return !(code < 48 || code > 57);
    }),
    tap(digit => this.keys += String.fromCharCode(digit))
  ).subscribe(_ => {
    const value = +this.keys;
    if (value > 999999) {
      console.log('Whoa - that is way too high!!');
    }
  });
}
```



- Joins and flattens a set of Observables into a sequence executed serially
- Subscribes to each “inner” Observable only after previous has completed
- Merges all results into a single Observable

```
ngOnInit(): void {
  this.subscription = fromEvent(document, 'click')
    .pipe(
      map(_ => interval(1000).pipe(take(4))),
      concatAll()
    ).subscribe(x => console.log(x));
}
```



- Executes an array of Observables in parallel
- Does not complete until all Observables in the array complete
- Amount of time required to complete will be the duration of the longest Observable

```
ngOnInit(): void {
  this.subscription = forkJoin([
    this.service.makeRequest('Request One', 8000),
    this.service.makeRequest('Request Two', 1000),
    this.service.makeRequest('Request Three', 3000)
  ]).subscribe(results => {
    this.result1 = results[0];
    this.result2 = results[1];
    this.result3 = results[2];
  });
}
```



- Used in our Angular services for integrating with a REST API
- Available through import of *HttpClientModule* (usually in *AppModule*)
- Includes methods to support standard CRUD operations



HTTP Client



- Injected as a dependency in our service constructor
- Methods supported include:
 - *get()* – performs a GET operation to retrieve data
 - *post()* – performs a POST operation to add data
 - *put()* – performs a PUT operation (typically for updating data)
 - *delete()* – performs a DELETE operation to delete data



HTTP Client



- All return an Observable for async handling
- Support providing a request body (where applicable) in the form of an object
- Also support specification of custom headers (as required)



Angular In-Memory Web API



- In some cases, the API may be developed separately and in parallel
- In-memory Web API enables work to continue on the UI while in progress
- Provides a fake server that mimics all CRUD operations of a REST API
- Backed with a transient collection to provide “database”



Angular In-Memory Web API



- First install using `npm install angular-in-memory-web-api --save-dev`
- Create a service that implements the `InMemoryDbService` interface
- Implement the `createDb()` method to build out in-memory “database”



- `createDb()` returns an object with one or more key-value pairs
- Each key represents a “table”
- Each value is a collection of objects representing “table” rows
- Row object structure should match up with application models



Angular In-Memory Web API



```
data.service.ts X
src > app > data.service.ts > ...
1  import { Injectable } from '@angular/core';
2  import { InMemoryDbService } from 'angular-in-memory-web-api';
3
4  @Injectable({
5    providedIn: 'root'
6  })
7  export class DataService implements InMemoryDbService {
8
9    constructor() { }
10
11   createDb(): any {
12     return {
13       orders: [
14         { id: 1, customerId: 1, total: 199.99 },
15         { id: 2, customerId: 2, total: 37.55 },
16         { id: 3, customerId: 1, total: 54.89 },
17         { id: 4, customerId: 3, total: 119.00 },
18         { id: 5, customerId: 4, total: 1045.87 }
19       ],
20       orderDetails: [
21         { id: 1, orderId: 1, productId: 2, quantity: 18 },
22         { id: 2, orderId: 1, productId: 110, quantity: 1 },
23         { id: 3, orderId: 1, productId: 1, quantity: 14 },
24         { id: 4, orderId: 2, productId: 99, quantity: 187 }
25       ]
26     };
27   }
28 }
29 }
```



Angular In-Memory Web API



- Import `HttpClientInMemoryWebApiModule`
- On import, call `forRoot()`, passing in service name that provides mock data
- For example, `HttpClientInMemoryWebApiModule.forRoot(DataService)`



Angular In-Memory Web API



- *HttpClientModule* must be imported before in-memory API module
- Can use environments to dynamically manage between dev and prod
- Enables specification of a delay in milliseconds (to mimic latency)

```
  ],
  imports: [
    BrowserModule,
    HttpClientModule,
    !environment.production ? HttpClientInMemoryWebApiModule.forRoot(DataService, { delay: 1000 }) : [],
    HeroesModule
  ],
  providers: [
```



Angular In-Memory Web API



- URL for mock API will be in form of “*api/<entity>*”
- *<entity>* corresponds to keys used in *createDb()* method (e.g., *orders*)
- All other service code for API integration can be written normally



HTTP Interceptors



- Usually, API's require an “Authorization” header (probably in the form of “Bearer <JWT>”)
- Sent on request and used by API to verify authorization to make call
- This custom header can be added to every API method call



HTTP Interceptors



- However, can lead to an extensive amount of repeat code
- Instead, can use an HTTP interceptor to automate
- Intercepts HTTP requests/responses originating from *HttpClient*
- Enables designation of configuration or activity that should occur for each



HTTP Interceptors



- Use *ng generate interceptor <name>* to create
- Provides a service that implements *HttpInterceptor*
- Provides implementation for the *intercept()* method to enable

```
authorization.interceptor.ts ×

src > app > authorization.interceptor.ts > ...
  1 import { Injectable } from '@angular/core';
  2 import { HttpRequest, HttpHandler, HttpEvent, HttpInterceptor } from '@angular/common/http';
  3 import { Observable } from 'rxjs';
  4
  5 @Injectable()
  6 export class AuthorizationInterceptor implements HttpInterceptor {
  7
  8   constructor() {}
  9
 10  intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
 11    const authorizationReq = request.clone({ setHeaders: { Authorization: 'TOKEN' } });
 12    return next.handle(authorizationReq);
 13  }
 14}
 15
```



HTTP Interceptors



- *intercept()* accepts two parameters:
 - *HttpRequest* object representing current request
 - *HttpHandler* object denoting next interceptor in chain (if applicable)
- Enables chaining together of multiple interceptors (e.g., authorization + logging)
- Last interceptor in the chain is *HttpBackend*



HTTP Interceptors



- To use, must be registered with a module (usually `AppModule`)
- Likely will not leverage `providedIn` property of `@Injectable()`
- Instead, will use `providers` property of `@NgModule()`



HTTP Interceptors



- The object passed to *providers* includes valuable config options
- This config is what enables the chaining capability
- Also note – because of chaining, order of import is important

```
  ],
  providers: [
    { provide: HTTP_INTERCEPTORS, useClass: AuthorizationInterceptor, multi: true }
  ],
  bootstrap: [AppComponent]
```



Unsubscribing from Observables



- Good practice to unsubscribe from Observables
- Helps prevent memory leaks
- Couple of options:
 - Explicitly unsubscribing
 - Using `async pipe`



Unsubscribing from Observables



- When subscribing, can store a reference to the subscription
- In *ngOnDestroy()*, can use the *unsubscribe()* method to unsubscribe

```
        export class ObservablesComponent implements OnInit, OnDestroy {  
  
    result1: string;  
    result2: string;  
    result3: string;  
    subscription: Subscription;  
  
    constructor(private service: MyService) { }  
  
    ngOnInit(): void {  
        this.subscription = forkJoin([  
            this.service.makeRequest('Request One', 8000),  
            this.service.makeRequest('Request Two', 1000),  
            this.service.makeRequest('Request Three', 3000)  
        ]).subscribe(results => {  
            this.result1 = results[0];  
            this.result2 = results[1];  
            this.result3 = results[2];  
        });  
    }  
  
    ngOnDestroy(): void {  
        this.subscription?.unsubscribe();  
    }  
}
```



Unsubscribing from Observables



- async pipe provides alternative to subscribing to an Observable
- In template, can pipe the Observable using `<observable> | async`
- This handles both subscription and unsubscription for us



Forms in Angular



Web Forms



- Web applications use forms to gather input from users
- When building apps, remember the adage “if a user can do it, they’ll try”
- Need to implement a rich validation strategy for data input (both client-side and server-side)



Web Forms



- Angular offers two options for building our forms:
 - Template-driven – what we've been using so far
 - Reactive



Template-Driven vs. Reactive



Feature	Template-Driven	Reactive
Form model setup	Implicit, created by directives	Explicit, created in code
Data model	Unstructured and mutable	Structured and immutable
Predictability	Asynchronous	Synchronous
Form validation	Directives	Functions
Testability	More difficult	Less difficult
Scalability	Considered worse	Considered better
Ease of use	Simpler	More complex



Template-Driven and ngModel



- Already discussed *FormsModule* and the *ngModel* directive
- We saw that *ngModel* is one option for adding two-way data binding to our forms



Reactive Forms



- Will leverage the *ReactiveFormsModule*
- Will build out our forms in code, in the component class
- Will link our HTML elements in the template to the coded form objects
- Will utilize reactive techniques to manage state changes and validation



- Key classes used in Reactive forms approach include:
 - *FormControl* – represents individual form control (e.g., input element)
 - *FormGroup* – represents collection of form controls (static)
 - *FormArray* – represents collection of form controls (dynamic)
- Available in the `@angular/forms` npm package



- In code, can build out a set of controls as a group using *FormGroup*
- Constructor takes a set of key-value pairs for the *FormControl* objects
- Key will be a unique name for form control
- Value will be the associated *FormControl* object



- *formGroup* used to associate a `<form>` element in component HTML to `FormGroup` object
- *FormControlName* used to associate a form control in the HTML to a `FormControl` in the group



- Can reference *FormControl* object in HTML expressions and directives using `<form-group-name>.controls.<form-control-name>.<property-name>`
- For example, `orderForm.controls.description.value`
- Alternatively, create getter properties for controls to allow shortened syntax

```
order-form.component.html X
src > app > order-form > order-form.component.html > ...
1   <h1>Enter New Order</h1>
2   <form [formGroup]="orderForm" (ngSubmit)="submit()">
3     <div>
4       <label>Order Description:</label>
5       <input type="text" name="description" placeholder="description" formControlName="description" />
6     </div>
7     <div>
8       <p>Entered description: {{orderForm.controls.description.value}}</p>
9     </div>
10    </form>
11
```



Reactive Forms



order-form.component.ts X

```
src > app > order-form > order-form.component.ts > ...
2   import { AbstractControl, FormControl, FormGroup } from '@angular/forms';
3
4   @Component({
5     selector: 'app-order-form',
6     templateUrl: './order-form.component.html',
7     styleUrls: ['./order-form.component.css']
8   })
9   export class OrderFormComponent implements OnInit {
10
11   orderForm = new FormGroup({
12     description: new FormControl('Default')
13   });
14
15   constructor() { }
16
17   ngOnInit(): void {
18   }
19
20   get description(): AbstractControl {
21     return this.orderForm.controls.description;
22   }
23
24   submit(): void {
25
26   }
27
28 }
```

order-form.component.html X

```
src > app > order-form > order-form.component.html > ...
1   <h1>Enter New Order</h1>
2   <form [FormGroup]="orderForm" (ngSubmit)="submit()">
3     <div>
4       <label>Order Description:</label>
5       <input type="text" name="description" placeholder="description" formControlName="description" />
6     </div>
7     <div>
8       <p>Entered description: {{description.value}}</p>
9     </div>
10    </form>
11
```



Providing Feedback on Input Status



- Angular automatically applies the following classes to communicate control status (or a combination of them):

Class	Purpose
ng-untouched	User has not yet interacted with control
ng-touched	User has interacted with control
ng-dirty	Control has a value
ng-pristine	Control does not yet have a value
ng-valid	Value of control is valid (according to defined validation rules)
ng-invalid	Value of control is not valid



Adding Form Validation



- *FormControl* constructor accepts a parameter for default value
- It accepts a second parameter for one or more validators to be applied to the control
- Possible options include required input, minimum/maximum length, minimum/maximum value
- Also supports email validation as well as *RegExp* pattern validation

```
})
export class OrderFormComponent implements OnInit {

  orderForm = new FormGroup({
    description: new FormControl('Default', [Validators.required, Validators.minLength(10), Validators.maxLength(25)])
  });

  constructor() { }

  ngOnInit(): void {
  }
}
```



Adding Form Validation



- With definition in place, can see immediate effects on interaction with form

Enter New Order

Order Description:

Entered description:

Enter New Order

Order Description:

Entered description: First quart

Enter New Order

Order Description:

Entered description: First

Enter New Order

Order Description:

Entered description: First quarter order with 1



Providing User with Visual Cues



- *FormGroup* and *FormControl* objects have status properties (like status classes previously mentioned)
- Invalid status for one or more controls rolls up to invalid status for form
- Can use to prevent form submission if any input is invalid

```
order-form.component.html X
src > app > order-form > order-form.component.html > form > button
1  <h1>Enter New Order</h1>
2  <form [formGroup]="orderForm" (ngSubmit)="submit()">
3    <div>
4      <label>Order Description:</label>
5      <input type="text" name="description" placeholder="description" formControlName="description" />
6    </div>
7    <div>
8      <p>Entered description: {{description.value}}</p>
9    </div>
10   <button type="submit" [disabled]="!orderForm.valid">Create Order</button>
11 </form>
12
```

Enter New Order

Order Description: First

Entered description: First

Create Order



Providing User with Visual Cues



- *FormControl* objects can use status properties to add helper messages
- Can use valid or invalid for general “something’s wrong” message
- Or can use the errors collection on the input to fine tune messaging

```
order-form.component.html X
src > app > order-form > order-form.component.html > ...
1  <h1>Enter New Order</h1>
2  <form [formGroup]="orderForm" (ngSubmit)="submit()">
3  |   <div>
4  |     <label>Order Description:</label>
5  |     <input type="text" name="description" placeholder="description" formControlName="description" />
6  |     <span class="help-block" *ngIf="description.touched && description.errors?.required">Description is required</span>
7  |     <span class="help-block" *ngIf="description.touched && description.errors?.minlength">Description must be at least 10 characters long</span>
8  |     <span class="help-block" *ngIf="description.touched && description.errors?.maxlength">Description must no more than 25 characters long</span>
9  |   </div>
10 |   <div>
11 |     <p>Entered description: {{description.value}}</p>
12 |   </div>
13 |   <button type="submit" [disabled]="!orderForm.valid">Create Order</button>
14 </form>
15
```



- Using *FormArray*, can add controls dynamically to our forms
- Can create as part of *FormGroup* using `<prop-name>: new FormArray([])`
- If referenced via property and cast as *FormArray*, can use standard array methods like `push()` to manage



- *FormArray* elements can contain individual *FormControl* objects
- Can also contain *FormGroup* objects (e.g., row-level edits in dynamic table)
- Will look at how to implement in next exercise

Enter New Order

Order Description: 

Line Items:

Product Number	Quantity
ABC123	14
DEF456	8
GHI999	100.4
Product Number	0

[Add Line Item](#)

Entered description: First quarter order

[Create Order](#)



Manipulating Form Data



- *FormGroup* provides two methods for changing form data programmatically:
 - *setValue* – replaces all controls
 - *patchValue* – updates a specific subset of the controls
- Methods take key-value pairs representing control name and target value
- To pull entire set of form data (e.g., on submit), use `<form-name>.value`



Manipulating Form Data



```
populateForm(): void {
  if (this.orderDetails.length < 2) {
    this.orderDetails.push(this.fb.group({
      productNumber: [''],
      quantity: [0]
    }));
  }
  this.orderForm.setValue({
    description: 'Auto-populated',
    orderDetails: [
      { productNumber: 'AAA111', quantity: 13 },
      { productNumber: 'BBB222', quantity: 99 }
    ]
  });
}

resetOrderDetailQuantities(): void {
  this.orderForm.patchValue({
    orderDetails: [
      { quantity: 100 },
      { quantity: 100 }
    ]
  });
}

submit(): void {
  console.log(this.orderForm.value);
}
```

Enter New Order

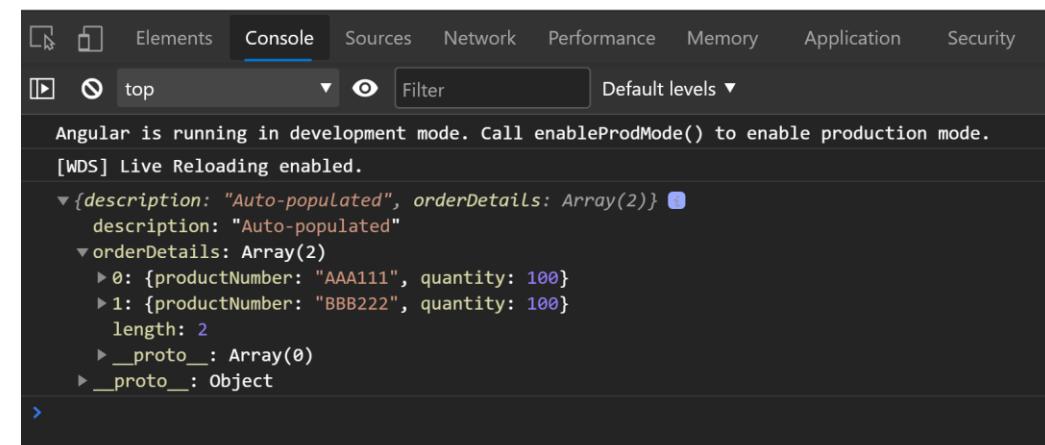
Order Description:

Line Items:

Product Number	Quantity
AAA111	100
BBB222	100
Add Line Item	

Entered description: Auto-populated

[Create Order](#) [Auto-Populate](#) [Reset Order Quantities](#)





- *FormControl* objects provide two Observable properties:
 - *statusChanges* – notifies when status of control changes (e.g., invalid to valid)
 - *valueChanges* – notifies when value of control changes
- Could be used to proactively display hints for validation rules as entered
- Could also be used to check back-end for duplicate input
- Be aware of potential performance implications of complex logic on change



Creating Custom Validator



- Can create a custom validator for your specific model
- To do so, create a class with a static method that accepts form control
- Return either *ValidationErrors* or null



Creating Custom Validator



- *ValidationErrors* represents key-value collection of validation errors
- Apply as validator in *FormBuilder* code
- Optionally look for validation key in collection of control errors for error message display

```
description.validator.ts ×  
src > app > description.validator.ts > ...  
1 import { AbstractControl, ValidationErrors } from '@angular/forms';  
2  
3 export class DescriptionValidator {  
4   static cannotContainMeanWords(control: AbstractControl): ValidationErrors | null {  
5     if ((control.value as string).toLowerCase().includes('stupid')) {  
6       return { cannotContainMeanWords: true };  
7     }  
8     return null;  
9   }  
10 }  
11
```



Creating Custom Validator



```
private buildForm(): void {
  this.orderForm = this.fb.group({
    description: ['Default', [
      Validators.required,
      Validators.minLength(10),
      Validators.maxLength(25),
      DescriptionValidator.cannotContainMeanWords
    ]],
    orderDetails: this.fb.array([
      this.fb.group({
        productNumber: [''],
        quantity: [0]
      })
    ])
  });
}
```

```
<span class="help-block" *ngIf="description.touched && description.errors?.maxLength">Description must no more than 25 characters long</span>
<span class="help-block" *ngIf="description.touched && description.errors?.cannotContainMeanWords">Whoa, watch the sailor mouth!</span>
<span *ngIf="showDescriptionHint">Description should be at least 10 and no more than 25 characters long</span>
</div>
```



Testing in Angular



Types of Testing



- There are multiple types of testing for an Angular app
 - Unit testing
 - Integration testing
 - E2E (End-to-end) or system testing
 - Performance testing
 - User acceptance testing
 - ...



Philosophy of Testing



- Whenever possible, we automate
- Takes time and extra coding to build out the test suites
- However, in the long run, often leads to higher quality and efficiency gains



Philosophy of Testing



- Automation allows testing to be wired into the software delivery pipeline
- Quality gates at various stages help to ensure defects are discovered earlier
- The earlier a defect is discovered, the cheaper it is to fix



Philosophy of Testing



- As mentioned, our ability (or inability) to test can speak to quality of design
- If something is difficult to test, it might be because its design is not optimal
- Perhaps there is tight coupling between elements of the system that will ultimately lead to brittleness



Philosophy of Testing



- Automated testing can help us protect against regression
- Tests can also represent an excellent source of system documentation
- Whether “test first”, “test after” or “test as you go”, testing is a valuable use of time



Discussion – So Why Don't We Test More?





Testing an Angular App



- Unit testing is often performed using tools like Karma/Jasmine or Jest
- E2E (end-to-end) testing may be performed using tools like Protractor
- Often, a logical grouping of tests will be organized in a “test suite”
- Each suite can represent tests for a specific feature



Testing an Angular App



- Karma/Jasmine are very popular tools used for testing Angular apps
- Jasmine is the testing framework
- Karma is the test runner (browser-based)
- Takes a BDD (Behavioral Driven Development) approach to testing



Organization of .spec.ts Files



- *describe* method used to define a test suite (grouping of tests for feature)
- *it* method(s) used to define individual tests or test specs
- Unit tests often follow what's called AAA (Arrange, Act, Assert) methodology



Organization of .spec.ts Files



- General format is `describe('<suite name>', () => { ... })`; to define suite
- Contained within will be one or more `it('<test name>', () => { ... })`; tests



Organization of .spec.ts Files



```
⚠ app.component.spec.ts ×
src > app > ⚠ app.component.spec.ts > ...
1 import { TestBed } from '@angular/core/testing';
2 import { RouterTestingModule } from '@angular/router/testing';
3 import { AppComponent } from './app.component';
4
5 describe('AppComponent', () => {
6   beforeEach(async () => {
7     await TestBed.configureTestingModule({
8       imports: [
9         RouterTestingModule
10      ],
11      declarations: [
12        AppComponent
13      ],
14    }).compileComponents();
15  });
16
17  it('should create the app', () => {
18    const fixture = TestBed.createComponent(AppComponent);
19    const app = fixture.componentInstance;
20    expect(app).toBeTruthy();
21  });
22
23  it(`should have as title 'unit-test-app'`, () => {
24    const fixture = TestBed.createComponent(AppComponent);
25    const app = fixture.componentInstance;
26    expect(app.title).toEqual('unit-test-app');
27  });
28
29  it('should render title', () => {
30    const fixture = TestBed.createComponent(AppComponent);
31    fixture.detectChanges();
32    const compiled = fixture.nativeElement;
33    expect(compiled.querySelector('.content span').textContent).toContain('unit-test-app app is running!');
34  });
35});
36
```



Setup & Teardown



- In some cases, tests require a particular starting state
- `beforeEach(() => {});` can be used to run common setup before each test
- `afterEach(() => {});` can be used to run common teardown after each test
- Jasmine also supports `beforeAll()` and `afterAll()` forms for setup/teardown before & after all tests



Configure, Run & Debug in VS Code



- Verify configuration in `./karma.conf.js` and `./tsconfig.spec.json`
- May need to set `$CHROME_BIN` environment variable
- Can set inline (for current session) or add to `~/.bashrc` for persistent use
- Encapsulates path to `chrome.exe` (similar setup if using Firefox or Edge)



Configure, Run & Debug in VS Code



- Add launch configuration (.vscode/launch.json)
- Can setup config for debugging through browser and through Karma
- Config becomes selectable at bottom of VS Code window



Configure, Run & Debug in VS Code



- To debug app through browser, run `ng serve`, select debug config for browser, set breakpoint and choose Run | Start Debugging
- `ng test` will be used to run unit tests
- To debug test, run `ng test`, select debug config for tests, set breakpoint and choose Run | Start Debugging (or use “Debug” link in Karma display)



Configure, Run & Debug in VS Code



- To debug app through browser, run `ng serve`, select debug config for browser, set breakpoint and choose Run | Start Debugging
- `ng test` will be used to run unit tests
- To debug test, run `ng test`, select debug config for tests, set breakpoint and choose Run | Start Debugging (or use “Debug” link in Karma display)



Configure, Run & Debug in VS Code



- Use `ng test --no-watch --code-coverage` to run tests and report on code coverage
- Code coverage can help confirm amount of code (and branches of code) being exercised by unit tests



@angular/core/testing Namespace



- Provides access to the *TestBed* class
- Allows us to create a “testing” module that mimics a standard Angular module
- Uses *configureTestingModule()* method to accomplish
- Essentially can detach Angular artifact under test from its host module and attach to testing module



configureTestingModule()



- Uses object with same properties as `@NgModule` decorator
- Allows us to register and configure the test module as we would a live module in the app
- Call `compileComponents()` on the configured testing module to complete setup



compileComponents()



- Compiles components configured in the testing module
- Inlines external CSS files and templates
- Angular CLI will handle this inlining automatically but supports testing with other tools besides the CLI



Testing Components



- @angular/core/testing provides access to *ComponentFixture*
- Wrapper class around an Angular component instance
- Enables interaction with the component and its corresponding selector for test in isolation



Testing Components



- Can use `TestBed.createComponent()` to create an instance of a target component for testing
- With that instance, can test public properties and methods (like any other class)
- Can also use `detectChanges()` to do content tests against rendered view



Testing Components with Dependencies



- In many cases, component under test will have service dependencies
- Goal is to test component in isolation
- Can use the following to help isolate testing to component directly:
 - Stubbing
 - Spying



Stubbing vs. Spying



- Stubbing is utilizing Dependency Injector to inject a stub (or mock) of the dependency vs. the real thing
- Allows more controlled interaction
- Spying is injecting the actual dependency but attaching a “spy” to a method called on component
- Provides options – return mock data or let real call through



Stubbing



- Can use *providers* property of test module and *useValue* version of dependency definition to stub a given service token
- Test setup pulls service reference from root injector which gets mapped to stub
- Can also use *overrideComponent()* if unable to retrieve service from root injector (dependency provided by component injector)



Spying



- Can use the `spyOn()` method to spy on specific method(s) of a service
- Allows verification of arguments passed to actual calls
- Alternatively, can use `createSpyObj()` to stand up a hook for intercepting calls and controlling results



Testing Asynchronous Service Dependencies



- Can use `async` with `whenStable()` or `fakeAsync` with `tick()`
- Can make body of `it` spec method `async` and process assertions using `whenStable()` Promise
- Call `detectChanges()` to trigger `ngOnInit` lifecycle



Testing Asynchronous Service Dependencies



- Can also call `detectChanges()` on async service call completion to trigger data binding and DOM query
- Approach using `fakeAsync` and `tick()` very similar

```
it('should get data from async service dep operation', waitForAsync(() => {
  fixture.detectChanges();

  fixture.whenStable().then(() => {
    fixture.detectChanges();
    expect(fixture.nativeElement.querySelectorAll('tr').length).toBe(7);
  });
}));

it('should get data from async service dep operation diff method', fakeAsync(() => {
  fixture.detectChanges();
  tick(1000);
  fixture.detectChanges();
  expect(fixture.nativeElement.querySelectorAll('tr').length).toBe(7);
}));
```

```



# Testing Component Inputs/Outputs



- As part of test, will build a “dummy” component that will act as host for component under test
- Includes utilization of inputs/outputs of component under test
- Tests execute against an instance of the host component with assertions against expected properties of component under test
- Trigger events (e.g., *buttonRef.click()*) from test to exercise output bindings



# Testing Services



- Get service instance from injector and test public API
- Goal is testing service in isolation – will mock downstream dependencies
- Common types of service testing:
  - Test synchronous ops
  - Test asynchronous ops
  - Test services with downstream dependencies



# Testing Services – Synchronous Ops



- Use `TestBed.inject()` to retrieve instance of service under test
- If provided by root injector, no additional config required in `TestBed.configureTestingModuleTestingModule()`
- Otherwise, can add to providers property of testing module to access
- Exercise method and verify expected results



# Testing Services – Asynchronous Ops



- Use same techniques as with synchronous ops to retrieve service instance
- If method to be tested is Observable, subscribe and verify
- Use `done()` at end to signify to Karma that Observable has completed, and expected results can be verified



# Testing Services – Downstream Dependencies



- As with components, can mock downstream dependencies to ensure isolation
- However, when testing service that wraps calls to API using *HttpClient*, Angular offers some additional support
- Import *HttpClientTestingModule* in `@angular/common/http/testing` namespace



# Testing Services – Downstream Dependencies



- Get an instance of *HttpTestingController* from root injector (defined in same namespace)
- Will be used to “mock out” *HttpClient* calls
- Call service method and subscribe to Observable in test (as normal)



# Testing Services – Downstream Dependencies



- Set up expected API calls using `HttpTestingController` (to replace real calls via `HttpClient`)
- Can exercise assertions against the HTTP calls as well
- Execute `flush()` on each mocked HTTP call, optionally providing data to be returned from the call in the service



# Testing Pipes



- Create instance of Pipe class in test
- Exercise *transform()* method with test data, verifying expected results



# Testing Directives



- Applied to a host component so testing often executed in that context
- Can build test host component and set it plus directive up in test module
- Then test directive using host element, verifying that expected modifications are in place from directive presence
- Also, use defined inputs on directive to test each possible condition